Composable Abstractions for Synchronization in Dynamic Threading Platforms

by

Jim Sukha

S.B. (Mathematics), Massachusetts Institute of Technology (2004)
S.B. (Electrical Engineering and Computer Science), Massachusetts Institute of Technology (2004)
M.Eng. (Electrical Engineering and Computer Science), Massachusetts Institute of Technology (2005)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2011

Author ....
Department of Electrical Engineering and Computer Science
August 26, 2011

Certified by ....
Charles E. Leiserson
Professor
Thesis Supervisor

Accepted by ....
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Composable Abstractions for Synchronization in Dynamic Threading Platforms

by

Jim Sukha

Submitted to the Department of Electrical Engineering and Computer Science
on August 26, 2011, in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Abstract

High-level abstractions for parallel programming simplify the development of efficient parallel applications. In particular, ***composable*** abstractions allow programmers to construct a complex parallel application out of multiple components, where each component itself may be designed to exploit parallelism. This dissertation presents the design of three composable abstractions for synchronization in dynamic-threading platforms, based on ideas of task-graph execution, helper locks, and transactional memory. These designs demonstrate provably efficient runtime scheduling for programs with synchronization.

For applications that use task-graph synchronization, I demonstrate provably efficient execution of task graphs with arbitrary dependencies as a library in a fork-join platform. Conventional wisdom suggests that a fork-join platform can execute an arbitrary task graph only with special runtime support or by converting the graph into a series-parallel computation which has less parallelism. By implementing ***Nabbit***, a Cilk++ library for arbitrary task-graph execution, I show that one can in fact avoid introducing runtime modifications or additional constraints on parallelism. Nabbit achieves an asymptotically optimal completion-time bound for task graphs with constant degree.

For applications that use lock-based synchronization, I introduce ***helper locks***, a new synchronization abstraction that enables programmers to exploit asynchronous task parallelism inside locked critical regions. When a processor fails to acquire a helper lock, it can help complete the parallel critical region protected by the lock instead of simply waiting for the lock to be released. I also present ***HELPER***, a runtime for supporting helper locks, and prove theoretical performance bounds which imply that HELPER achieves linear speedup on programs with a small number of highly parallel critical regions.

For applications that use transaction-based synchronization, I present ***CWSTM***, the first design of a transactional memory (TM) system that supports transactions with nested parallelism and nested parallel transactions of unbounded nesting depth. CWSTM demonstrates that one can provide theoretical bounds on the overhead of transaction conflict detection which are independent of nesting depth. I also introduce the concept of ***ownership-aware TM***, the idea of using information about which memory locations a software module owns to provide provable guarantees of safety and correctness for open-nested transactions.

Thesis Supervisor: Charles E. Leiserson
Title: Professor

Acknowledgments

Over the past *n* years, I have spent a lot of time writing. Between papers, problem sets, proposals, and this dissertation, I have spent more time writing than I ever would have imagined when I first started my PhD. Of all the documents I have worked on, however, these acknowledgments are perhaps the most difficult to compose. Words seem insufficient to convey my gratitude to everyone who has contributed to the creation of this dissertation. Due to the constraints of space and time, however, I'm afraid they will have to suffice.

Thanks to Charles Leiserson, my advisor, for his guidance and support during my time at MIT. I am grateful to Charles for many things, but in particular, I'd like to thank Charles for imparting upon me two significant lessons. First, having Charles as a mentor has improved my writing and presentation skills. Second, working with Charles has taught me the importance of time management and managing deadlines. I could spend several pages going into detail about these points or trying to recount the myriad of other nuggets of wisdom I have learned from Charles or all the ways that he has helped me over the years. I believe, however, that he would agree that this document is long enough as it is. It is hard for me to believe that it has been over a decade since I first met Charles as a student in 6.046 and a summer UROP student. It has been a long journey but a wonderful experience.

This dissertation would not exist without the help of Kunal Agrawal, my friend and collaborator. Kunal seems to have an uncanny ability to interpret and improve upon my half-baked ideas and ramblings, even when I have trouble making sense of them myself. I could not have asked for a better collaborator or coauthor. The days when we could bounce ideas off each other at the whiteboard are long gone now. But I hope that there will still be opportunities for us to work together in the future.

Thanks to Bradley Kuszmaul for introducing me to the MIT Cilk code. I remember fondly one of my early summer UROP projects working with the source-to-source translator for the MIT Cilk compiler. That project was my first experience working with Cilk. It seems that I've been a convert ever since. More generally, Bradley has always been a source of both helpful advice and fun trivia. I am constantly amazed by the breadth and depth of his knowledge. Every conversation with Bradley always teaches me something new.

I would like to thank Saman Amarasinghe and Bradley Kuszmaul for serving on my thesis committee and for giving me helpful comments on how to improve my thesis. I apologize to the readers of my thesis — Saman, Bradley, and Charles — I'm afraid I didn't have enough time to make my thesis shorter.

Thanks to all the members of the SuperTech research group, both past and present. I'm sure I would have quit school long ago if I had not had such excellent colleagues and amazing people to work with each day.

- Thanks to Angelina Lee for collaborations on ownership-aware TM and the work on PR-Cilk. More generally, I owe Angelina a great thanks for all the helpful comments she has provided on my work over the years. I have always been able to count on her to provide constructive but critical feedback on my ideas.
- Thanks to Jeremy Fineman for collaborations on CWSTM. Jeremy's talk on race detection in Cilk was one of the first technical talks I ever attended, and it provided the inspiration for much of the design of CWSTM. Jeremy's expertise was crucial in

putting together the whole CWSTM design.

- Thanks to TB Schardl for collaborations on pedigrees and DPRNG. Although this material didn't make the final draft, TB has my gratitude for doing all the heavy lifting on this other work while I was writing my thesis.
- Thanks to all the administrative staff that have worked with SuperTech: Leigh Deacon, Alissa Cardone, Kyle Lagunas, and Marcia Davidson. A special thanks to Marcia for helping me take care of all the many tasks required for graduation this year. Also, thanks to Mary McDavitt, who although not officially affiliated with SuperTech, has helped me on a number of occasions over the years.
- Thanks to all the other individuals who have been a part of SuperTech over the years, either as students, staff, or visitors. In particular, I would like to thank Rezaul Chowdhury, Will Hasenplaugh, Edya Ladan-Mozes, Aamir Shafi, Yuan Tang, and Justin Zhang, who have all been subjected to multiple versions of my jobtalk this year.

Thanks to David Ferry and Kunal Agrawal at Washington University in St. Louis for their work on Nabbit's data-block interface and asynchronous Cholesky factorization. I am also grateful to everyone who has worked on Cilk projects (Cilk, Cilk++, or Cilk Plus) in the past or present. The Cilk project has provided me with an excellent platform for research and experimentation for my PhD thesis.

Many other people have helped me along in the official process of receiving a PhD. Thanks to Arvind and Albert Meyer for serving on my RQE committee. Thanks to Erik Demaine, Madhu Sudan, and Charles Leiserson for giving me the opportunity to TA for 6.046. Thanks to everyone in the EECS Graduate Office; my time at MIT has run smoothly, and I'm sure I'm unaware of all the ways they have helped me over the years.

Thanks to everyone I met during my internships at Intel, Google, and AMD, as well as all the individuals in industry I met this year. I would also like to thank the various funding organizations and companies which have funded my research. These funding sources include NSF Grants (ACI-0324974, CNS-0305606, CNS-0540248, CNS-0615215), Intel Corporation, an Akamai/MIT Presidential Fellowship, and the Siebel Scholars program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of any of these organizations.

Many people have helped me in an official capacity, but it is equally important for me to thank everyone who has supported me outside of work. Although research may have taken up the majority of my time at MIT, it is the time I have spent with everyone outside the lab that has been the most rewarding.

- A special thanks goes to Karen Lee, my good friend from Ashdown. Karen has always been there to keep me sane and connected to the outside world, even despite my best efforts to spend my life in the office. I would not have survived these years without her.
- Thanks to Chung Chan, my friend from undergrad who has accompanied me on this trek from undergrad through PhD at MIT.
- Thanks also to my friends Mark Harris and Katherine Yiu. Although they don't live anywhere near MIT, their virtual presence has always been here to keep me motivated through the years.
- Thanks to everyone I've met while living in Ashdown over the years. Between coffee

hour, brunch, and all the numerous social events, I will always remember my time there fondly. A special thanks to Terry and Ann Orlando, the housemasters of both old and new Ashdown, who have made Ashdown a wonderful place to call home.

- Thanks to everyone who has indulged me in my foosball obsession these past few years. In particular, I am grateful to Matt, TL, Bryan and Andy for helping field an IM foosball team these past two years. Thanks also to the many others I have had the pleasure of playing against — Alan, Fanny, Justin, and even Charles. This list should be much longer, but I'm afraid I've played too many games these past two years to remember...

Many other individuals should also be thanked for their friendship and help. A partial list includes Alfred, Alice, Chih-yu, Dave, Jelani, Jennifer, John, Lawson, Pei-Lan, Stephen, Tin, Wentao, Winnie, Yang, and Yun. Unfortunately, time and space constraints prevent me from making this list complete or thanking everyone individually. I wish I could say that *n* years ago, I was organized enough to create a list of people I should thank, and that I've been industrious enough to consistently maintain this list ever since. Unfortunately, since I was not, I am constructing it *a posteriori.* Thus, I have most likely have omitted someone who deserves to be thanked. To anyone who reads these acknowledgments and believes their name should be included in this list — you are most likely correct with high probability, and I offer my deepest apologies. ☺

Finally, above all, I would like to thank my family — my mother, father, and brother. They have been infinitely patient with me throughout this entire PhD process. I would not have made it this far without their love and support through all these years, and for that I am eternally grateful.

Contents

Chapter 1

Introduction

With the growing availability of multicore CPUs, parallel computing has emerged as a mainstream topic relevant to all programmers who wish to fully exploit the capabilities of modern computers. Unfortunately, realizing the performance benefits of multicores requires programmers to write parallel programs, a job traditionally left only to experts because of the challenges faced when writing parallel code. For example, when tasks can execute in parallel, programmers must sometimes consider the issue of ***scheduling*** — deciding on which processor to execute each task. When two parallel tasks try to modify the same piece of shared data, programmers must also consider the issue of ***synchronization*** — coordination between processors to ensure that the tasks do not perform any concurrent modifications to shared data that interfere with each other. Managing the low-level implementation details of scheduling and synchronization poses a significant challenge to programmers who wish to write efficient parallel code.

To tackle these challenges and facilitate parallel programming, industry and academia have been actively developing ***concurrency platforms*** — software platforms that provide users with parallel-programming abstractions and runtime support for scheduling. In particular, many of these platforms provide the abstraction of ***dynamic threading***, i.e., a parallel program creates many lightweight dynamic threads, with threads corresponding to tasks that are allowed to execute in parallel. Programmers do not map dynamic threads to processors; instead, the platform's runtime scheduler decides which threads to execute on which processors dynamically, as the program is being executed. Many state-of-the-art concurrency platforms, such as Cilk++ [93], Cilk Plus [73], Fortress [13], Habenero [21], Hood [31], Java Fork/Join Framework [90], OpenMP 3.0 [106], Task Parallel Library (TPL) [92], Threading Building Blocks (TBB) [110], and X10 [37,44], support dynamic threading because it frees programmers from the burden of explicit task scheduling. Modern dynamic-threading platforms contain runtime schedulers which are modeled after the provably efficient scheduler in MIT Cilk [51]. Cilk's scheduler, which is based on randomized work-stealing, has both an efficient implementation and theoretical bounds on time and space usage.

Existing dynamic-threading platforms however, generally lack equally high-level abstractions for dealing with the challenge of synchronization. Typically, programmers use locks for synchronization. When a ***critical section*** is protected by a lock L, a thread must acquire L before it can execute the code in the critical section. The platform guarantees

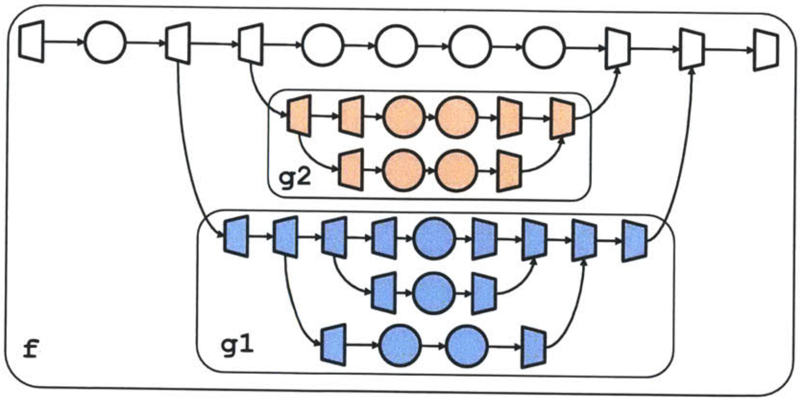


Figure 1-1: A parallel function f that spawns two parallel functions, g1 and g2. A platform with composable parallelism can simultaneously exploit the parallelism in all three functions.

that only one critical section can hold a given lock L at a time. Thus, critical sections protected by the same lock cannot execute concurrently. Unfortunately, the provable bounds on time and space for Cilk do not hold for programs that utilize locks or other forms of synchronization. An arbitrary use of locks imposes complex constraints for scheduling a computation, making it difficult for a runtime to provide strong, provable guarantees on performance.

Furthermore, existing dynamic-threading platforms typically require that all critical sections execute sequentially, because the schedulers for these platforms are not designed to interact with locks. Said differently, in these platforms, locking fails to exhibit the property of composable parallelism. For instance, consider a function f that spawns calls to two nested parallel functions g1 and g2, as shown in Figure 1-1. We say that a dynamic-threading platform exhibits ***composable parallelism*** if its scheduler is able to simultaneously and efficiently exploit parallelism in all functions, f, g1, and g2, even if they use synchronization, e.g., even if g1 acquires a lock to synchronize with some other code running in parallel with f. In most existing platforms, g1 can execute in parallel with g2 if these sections are protected by different locks, but the platform cannot exploit parallelism inside g1 or g2. Thus, these platforms fail to provide good performance even for computations which use a single lock, because calling a parallel function inside a critical section must serialize the execution of the function.

The lack of composable abstractions for synchronization creates several obstacles for programmers trying to write efficient parallel code. First, the lack of composable parallelism prevents users from reusing existing library code that has already been parallelized inside critical sections. Also, because platforms cannot exploit nested parallelism inside critical sections, to achieve good performance in a program that requires synchronization, programmers may be forced to use ***fine-grained locks*** — many locks that each protect a small critical section. Unfortunately, fine-grained locking is often beyond the capabilities of nonexpert programmers, since it requires programmers to correctly reason about all the possible ways that the execution of tasks might interleave when run concurrently on different processors. Allowing nested parallelism inside critical sections can simplify the development of some parallel applications, because it facilitates the reuse of existing library code and enables developers to improve program performance by using a few large but parallel critical sections.

Contributions

In this dissertation, I demonstrate that dynamic-threading platforms can provide composable abstractions for synchronization without sacrificing provable guarantees of performance and correctness. More specifically, my dissertation describes the following contributions:

- ***Task-graph synchronization in fork-join platforms.*** I present the first library in a fork-join dynamic-threading platform for provably efficient parallel execution of task graphs with arbitrary dependencies. Conventional wisdom suggests that executing an arbitrary task graph in a fork-join platform requires either special runtime support or conversion of the task graph into a series-parallel computation which has less parallelism. I demonstrate, however, that provably efficient task-graph execution using work-stealing is possible without introducing runtime modifications or additional constraints on parallelism. This work was done jointly with Kunal Agrawal and Charles E. Leiserson and appears in [9].
- ***Helper locks.*** I present helper locks, the first abstraction for synchronization in a dynamic-threading platform to effectively exploit asynchronous task parallelism inside locked critical sections. In particular, I describe a user-level runtime scheduler for supporting helper locks and prove theoretical bounds on completion time for platforms that use this scheduler. This work was done jointly with Kunal Agrawal and Charles E. Leiserson and appears in [10].
- ***Nested Parallelism in Transactional Memory.*** I describe the first design of transactional memory (TM) that supports transactions with nested parallelism and nested parallel transactions of unbounded nesting depth. In particular, I provide a theoretical performance bound on the overhead of conflict detection in TM which is independent of the maximum nesting depth of transactions. This work was done jointly with Kunal Agrawal and Jeremy T. Fineman and appears in [3].
- ***Ownership-aware Transactional Memory.*** I describe ownership-aware TM, the first design of TM that supports open-nested transactions with provable correctness guarantees. This design uses information about which memory locations a software module owns to constrain TM with open nesting, thereby making open-nested transactions safer and more intuitive to use. This work was done jointly with Kunal Agrawal and I-Ting Angelina Lee and appears in [5].

To demonstrate the utility of some of these designs, this dissertation also describes two prototype implementations.

First, I present ***Nabbit***, a Cilk++ library which demonstrates efficient execution of arbitrary task graphs using work-stealing. Nabbit supports parallel task-graph execution without introducing runtime modifications to Cilk++ or introducing any additional edges to the task graph. Nabbit provides composable parallelism, since it allows a Cilk function to efficiently run in parallel with task-graph execution. It also allows the computation of each task-graph node to be a parallel Cilk function.

I also present ***HELPER***, a prototype implementation of helper locks in MIT Cilk. To provide runtime support for helper locks, HELPER adds a new "parallel region" construct to Cilk. HELPER demonstrates that this construct can be implemented without introducing additional overheads to ordinary Cilk programs without helper locks. I also demonstrate that parallel regions can help improve the composability of Cilk with legacy code. More specifically, I show how parallel regions can enable legacy-C functions to perform callbacks to Cilk functions, a capability not permitted by MIT Cilk or Cilk++.

The remainder of this chapter briefly reviews some background material and provides a high-level overview of each of the topics covered in this dissertation. Section 1.1 briefly reviews the advantages of programming with dynamic threads as compared to static threads, the traditional approach for writing multithreaded code. Section 1.2 presents a high-level overview of task-graph execution in Cilk-like platforms. Section 1.3 discusses helper locks and their runtime support. Section 1.4 covers synchronization using transactional memory, namely CWSTM and ownership-aware TM. Finally, Section 1.5 outlines the remaining chapters in this dissertation.

1.1 Dynamic Threading and Composable Parallelism

This section briefly reviews the programming model of dynamic threading and compares it to static threading, the traditional approach to writing multithreaded programs. Programs written using dynamic threads exhibit composable parallelism, a property that programs written using static threads generally lack. As I discuss later in this dissertation, this lack of composable parallelism turns out to complicate the design of composable synchronization abstractions, since many synchronization primitives are designed for a static threading programming model.

Dynamic threading provides a threading model with several advantages over static threading — the traditional approach to writing multithreaded programs. Programming using static threading often requires developers to explicitly schedule and load-balance tasks on multiple processors. Static threading can also expose developers to unnecessary nondeterminism due to scheduling, thereby complicating the development of multithreaded programs. In contrast, dynamic threading offers a "processor-oblivious" programming model which frees developers from the burden of explicit task scheduling and enables libraries written using dynamic threading to exhibit composable parallelism.

Traditionally, developers have written multithreaded programs using ***static threads***, which couple the specification of parallelism in an application and its scheduling. With static threading, a programmer typically creates one static thread for each available (hardware) processor on the target system, and then explicitly assign tasks to each static thread. Static threads, which are also called ***persistent threads*** or ***pthreads***, are exemplified by POSIX threads [70], Windows API threads [60], and the threading model of the Java programming language [53].[1]

[1] No confusion should arise with the conventional use of the term "Pthread" as meaning a POSIX thread, since Pthreads are a type of pthread.

```
int fib(int n) {
  if (n < 2) {
    return n;
  }
  else {
    int x, y;
    x = spawn fib(n-1);
    y = fib(n-2);
    sync;
    return (x+y);
  }
}
```

Figure 1-2: The canonical Cilk example program of fib. This computation is tricky to program using static threads, because the recursive calls to fib are of differing sizes and it not obvious how to partition work equally across *P* processors. Although this recursive exponential-time method is a highly inefficient way to calculate the Fibonacci numbers, this code is useful for measuring runtime scheduling overheads in a dynamic-threading platform.

Since using static threads often requires programmers to explicitly handle task scheduling, it can be tricky in static-threaded programs to maintain adequate ***load balance***, an even distribution of work across processors. Achieving good load balance is especially difficult for programs whose parallelism is difficult to partition statically. For example, it is difficult to statically partition the computation of `fib(n)` shown in Figure 1-2 into *P* equal pieces, since the two recursive calls in `fib` are of different sizes. It is straightforward, however, to code and execute the `fib` computation efficiently using dynamic threading, because dynamic threading frees programmers from the burden of explicitly scheduling tasks on processors. A `spawn` does not create a new thread, but only indicates potential parallelism in a program that the runtime may or may not exploit.

Code using dynamic threads is often more composable than code using static threads because dynamic-threading languages offer a ***processor-oblivious*** model of computation, where linguistic extensions to the serial base language expose the logical parallelism within an application, without referring to the number of processors on which the application runs. Thus, in a ***dthreaded*** program — a program that uses dynamic threads— a developer coding a parallel function `F` can seamlessly call other dthreaded parallel functions `G1` and `G2` in parallel with each other without needing to worry about how many processors `G1` and `G2` each need to execute. In contrast, if `G1` and `G2` were from a library parallelized using static threads, the developer would need to control exactly how many threads `G1` and `G2` create to avoid oversubscribing a system with more threads than hardware processors. For example, the processor-oblivious model enables a simple but effective computation of `fib(n)` in Figure 1-2: the developer can recursively invoke `fib(n-1)` and `fib(n-2)` without needing to specify how many threads to use to execute each invocation.

As a more realistic example, consider a BLAS [89] library for linear algebra that provides a parallel matrix-multiplication method `G` coded with static threads. In libraries such as GotoBLAS [54, 116] or Intel MKL [74], often G creates one thread for each hardware processor, i.e., on a system with 16 processors, G creates 16 static threads. If a developer wants to write a parallel program F that creates two static threads, and each thread executes its own instance of parallel matrix multiplication G, then each instance of G creates 16 static threads, thereby creating a total of 32 static threads for a system with only 16 processors. This oversubscription of threads on the system can drastically hurt performance, since the operating system must context-switch between threads on each processor. Furthermore, if the developer writing F wants to perform multiple calls to G in parallel but on different size inputs, it is not obvious using static threads how the developer should allocate processors across the multiple instances. On a dynamic-threading platform, however, the developer leaves the scheduling to the runtime; the platform's runtime scheduler uses the same 16 (static) worker threads to efficiently execute all instances of G within F, even if the instances of G are different sizes. Thus, the platform provides composable parallelism.

As I discuss in this dissertation, however, synchronization primitives generally hurt the composability of dynamic-threading platforms such as Cilk [51, 73, 93]. Primitives such as locks or transactional memory are often designed for programs using static threads. Furthermore, Cilk's runtime scheduler is not designed to permit nested parallelism inside critical sections protected by locks or transactions. Thus, it can be difficult to effectively compose a parallel function G inside a function F if F is part of a critical section that requires some synchronization. This dissertation describes several approaches for providing composable abstractions for synchronization in Cilk-like platforms which require extending the programming model or runtime scheduler.

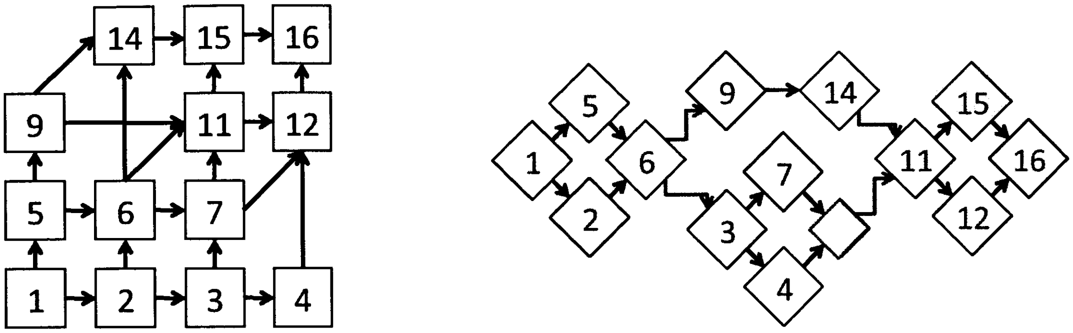
1.2 Task-Graph Synchronization

Chapter 2 demonstrates how to execute an arbitrary task graph in a fork-join platform in a provably efficient way using traditional work-stealing.[2] Task-graph synchronization is useful for applications whose parallel tasks have predictable but arbitrary dependencies, namely that the computation of one task depends on the computation of another task. Conventional wisdom suggests that task graphs with arbitrary dependency edges, such as the graph in Figure 1-3(a), cannot be executed on a fork-join dynamic-threading platform without executing a modified computation that has extra edges, e.g., the graph in Figure 1-3(b). In general, adding edges can significantly limit the parallelism of a computation, especially if tasks vary in size. As Chapter 2 shows, however, it is possible to execute arbitrary task graphs in a fork-join platform without introducing extra dependency edges and without requiring special runtime support. Furthermore, this approach to task-graph execution provides composable parallelism: one can easily and efficiently execute a fork-join computation and a task-graph computation in parallel, as well as exploit parallelism inside individual nodes in a task graph.

To demonstrate this approach in practice, I present ***Nabbit***, a Cilk++[3] library that enables programmers to specify and execute task graphs with arbitrary dependencies, such as

[2] These results on task-graph execution represent joint work [9] with Kunal Agrawal and Charles E. Leiserson.

[3] Cilk++ [93], which has since evolved into Cilk Plus [73], is a C++ version inspired by MIT Cilk [51].



(a) Arbitrary Task Graph.

(b) Series-Parallel Task Graph.

Figure 1-3: A task-graph computation with arbitrary dependency edges and a series-parallel conversion. Each numbered square represents a task. An edge from task i to j means j must execute after i. The graph in 1-3(a) shows a task-graph computation with arbitrary dependency edges, which is not easily expressed directly in Cilk. The graph in 1-3(b) is a conversion of the graph from 1-3(a) into a "series-parallel" graph. This conversion introduces additional dependency edges.

the graph in Figure 1-3(a). Nabbit requires minimal additional bookkeeping and requires no runtime-system modifications. In Chapter 2, I illustrate the efficiency of Nabbit theoretically by proving efficient completion-time bounds for Nabbit. In particular, I show that Nabbit achieves an asymptotically optimal completion-time bound for task graphs with constant degree. I also demonstrate the effectiveness of Nabbit empirically on a dynamic programming microbenchmark and on an asynchronous Cholesky factorization.[4]

Cilk++ with Nabbit exhibits composable parallelism. First, Nabbit allows fork-join Cilk++ code and task-graph execution to efficiently run together in parallel. Users of Nabbit can also write parallel Cilk++ code for the computation of each node in the task graph, and Cilk++'s work-stealing scheduler can efficiently exploit both parallelism between different task-graph nodes and parallelism within a node itself. Thus, Nabbit demonstrates composable task-graph synchronization with provably efficient runtime scheduling.

1.3 Helper Locks

Chapter 3 introduces helper locks, a new lock-based synchronization abstraction that enables dynamic-threading platforms to exploit asynchronous task parallelism inside critical sections.[5] A ***helper lock*** is similar to an ordinary lock, except that it may be connected to a large critical section containing nested parallelism. When a processor fails to acquire a helper lock `L` for a task `g1`, if `L` is connected to a large critical section `g2`, then the processor can help complete `g2` instead of simply waiting for `g2` to complete. For example, in a program that uses a concurrent resizable hash table, `g1` might represent an insert into the

[4] These results on asynchronous Cholesky factorization represent joint work [46] with David Ferry and Kunal Agrawal.

[5] The design of helper locks represents joint work [10] with Kunal Agrawal and Charles E. Leiserson.

table, and g2 might represent a resize operation that rebuilds the entire table. Typically, to achieve good performance, programmers need to use fine-grained locks to make critical sections as small as possible. Using helper locks, however, programmers can avoid some of the complexities of fine-grained locking without paying the performance penalty of a large critical section, provided that the critical section can be parallelized efficiently. Helper locks provide lock-based synchronization that exhibits composable parallelism, since they enable the composition of parallel functions inside critical sections.

I also demonstrate provably efficient runtime support for helper locks by presenting ***HELPER***, a prototype implementation of helper locks in MIT Cilk. HELPER relies on a new ***parallel region*** construct, which conceptually creates data structures for scheduling a nested code region R that are separate but connected to the data structures for R's parent region. The construct also allows the scheduler to adaptively move processors in and out of region R as R executes and the parallelism of R changes. One significant contribution presented in Chapter 3 is the analysis of randomized work-stealing when programs have parallel regions. For a computation where each parallel region contains sufficient parallelism, this analysis proves that HELPER is asymptotically efficient. More specifically, in a computation, let N be the number of parallel regions protected by helper locks, let T_1 be its work, and let $\widetilde{T}_\infty$ be its "aggregate span" — roughly, the sum of the spans (critical-path lengths) of all its parallel regions. Then, HELPER can complete the computation in expected time $O(T_1/P + \widetilde{T}_\infty + PN)$ on P processors. This bound indicates that programs with a small number of highly parallel critical regions can attain linear speedup.

Finally, HELPER's parallel region construct also turns out to be useful for improving the legacy compatibility of Cilk-like dynamic-threading platforms. Chapter 4 shows that parallel regions can be used to implement "subtree-restricted work-stealing," a type of restricted work-stealing. Dynamic-threading platforms such as Cilk and Intel TBB can use restricted work-stealing to support ***legacy callbacks***, that is, calls from a legacy-C function (a C function compiled using an ordinary serial compiler) back to a parallel Cilk function. Chapter 4 also provides a theoretical analysis of restricted work-stealing, demonstrating that a dynamic-threading platform can provide theoretical performance guarantees for legacy callbacks.[6]

1.4 Transactional Memory

Chapters 5 through 7 discuss composable synchronization in dynamic-threading platforms using nested transactions and transactional memory. This section briefly describes background material on transactional memory. It then gives an overview of the relevant topics covered in this dissertation: the transactional computation framework for modeling series-parallel computations with transactions, the CWSTM design for nested parallelism in transactions, and ownership-aware transactional memory for open-nested transactions. Nested transactions enhance the composability of transaction-based synchronization because they enable programmers to compose a function that uses transactions inside another transaction.

[6] This chapter covers joint work with Kunal Agrawal and I-Ting Angelina Lee [6], as well as work presented in [115].

Transactional memory, or TM for short, describes a collection of hardware and software mechanisms that provide a transactional interface for accessing memory. A TM system guarantees that any section of code that the programmer has specified as a ***transaction*** either appears to execute atomically or appears not to happen at all, even though other transactions may be running concurrently. Transactional memory was originally proposed by Herlihy and Moss in 1993 as a hardware mechanism for ensuring the atomic execution of critical sections [65]. More recently, however, there has been a renewed interest in TM, with many researchers proposing numerous designs for TM in hardware (e.g. [14, 59, 99]), in software (e.g., [2, 64, 97]), or in hybrids of hardware and software (e.g., [39, 84]). For a survey of these systems and other TM-related literature, see [88].

In the literature, researchers have argued that transactions and TM can provide the simplicity of use of coarse-grain locking while still providing an efficiency close to that of fine-grain locking. Normally, to guarantee that a code block executes atomically using a lock `L`, the user must ensure that all other conflicting code blocks also acquire the same lock `L`. The promise of TM is that a user can guarantee that a particular code block executes atomically by simply specifying that block as a ***transaction***. The TM system guarantees atomicity by tracking all the memory locations accessed by transactions, detecting conflicts with other parallel code, and then aborting and retrying transactions in case of a conflict. lock. Because a TM runtime executes transactions optimistically, in programs where conflicts are rare, TM can potentially provide performance comparable to programs that use hand-tuned fine-grain locking.

Researchers have also argued that using TM is more composable than using locks because TM systems can handle nested transactions. A transaction Y is said to be ***nested*** inside transaction X if transaction Y is called from within transaction X. For example, Figure 1-4 shows a simple transaction X with a nested transaction Y. In early work on TM, any nested transaction Y was assumed to be ***flat-nested*** inside the outer transaction X. A flat-nested transaction Y is subsumed into the parent transaction X. For the example in Figure 1-4, the code effectively behaves as though the `atomic` block enclosed between lines 4 and 6 has been elided. More generally, with flat-nesting semantics, a nested transaction Y that occurs within a function `f` called from within transaction X is also elided. This semantics for flat-nested transactions appears to be composable. One can call a function inside transaction X without needing to know whether the function will generate a nested transaction Y. In contrast, programming with locks is less composable, because programmers must carefully reason about the order of lock acquisitions and nesting of locks in order to avoid deadlock.

It turns out, however, that this argument for the composability of TM becomes more complicated when one considers transactions that can contain nested parallelism. Most proposed TM systems can detect conflicts between transactions correctly when a transaction `f` calls a ***nested transaction*** `g1`, but they do not allow `f` to contain nested parallelism or nested parallel transactions `g1` and `g2`. TM systems generally impose this restriction because they are designed for programs that use static threads, where generating nested parallelism inside a transaction is relatively expensive. Also, the semantics of transactions with nested parallelism is more complicated to reason about than the semantics of serial transactions.

This dissertation explores several of the issues of semantics and runtime support that

```
atomic {      // Transaction X
  x++;
  y++;
  atomic {    // Transaction Y
   i++;
  }
  z++;
}
```

Figure 1-4: A code example where transaction Y is nested inside X. The `atomic` block delimits the beginning and end of a transaction. If Y is flat-nested inside X, then the code behaves as though the `atomic` block for Y has been elided.

arise when transactions can contain nested parallelism and nested transactions.

First, Chapter 5 describes the ***transactional computation*** framework, a framework that is useful for precisely defining the semantics of TM with nested parallelism and nested transactions, and for proving the correctness of a TM runtime design.[7] This framework is used to describe nested transactions in CWSTM and ownership-aware TM.

Next, Chapter 6 presents CWSTM, the first design of a TM system that supports transactions with nested parallelism and nested parallel transactions of unbounded nesting depth in a dynamic-threading platform.[8] Nested parallelism turns out to complicate conflict detection between transactions considerably, since there is no longer a one-to-one mapping from a transaction to the processor executing the transaction. CWSTM shows, however, that one can provide a theoretical performance bound on the overhead of conflict detection in TM which is independent of the maximum nesting depth of transactions. CWSTM is a theoretical design of a software TM which integrates with a Cilk-style work-stealing scheduler. Thus, CWSTM demonstrates transaction-based synchronization with composable parallelism in a dynamic-threading platform.

Finally, Chapter 7 describes ownership-aware TM, a TM design that provides provable semantic guarantees for the optimization of "open-nested" transactions.[9] Because ordinary nested transactions can hurt performance by introducing unnecessary transaction conflicts, researchers proposed ***open-nested transactions*** as an optimization for eliminating such conflicts [98, 102, 103]. Unfortunately, by modeling open nesting more formally using the transactional computation framework, one can show that this optimization has semantic problems. In particular, its use can break the composability of transactions. This dissertation shows how one can resolve these problems using an ***ownership-aware TM*** — a TM that uses the information of which software module "owns" a particular memory location to constrain the behavior of nested transactions. Ownership-aware TM demonstrates that platforms can provide provable guarantees of correctness and safety for open-nested transactions.

[7] The transactional computation framework is derived from the formal model described in [8], which is joint work with Kunal Agrawal and Charles E. Leiserson.

[8] CWSTM represents joint work [3] with Kunal Agrawal and Jeremy T. Fineman.

[9] Ownership-aware TM represents joint work [5] with Kunal Agrawal and I-Ting Angelina Lee.

1.5 Outline

The remainder of this dissertation is organized into two major parts.

The first part — Chapters 2 through 4 — discusses extensions to the Cilk programming model for supporting more sophisticated forms of synchronization. Chapter 2 describes task-graph synchronization using work-stealing, and presents the Nabbit library for task-graph execution. Chapter 3 introduces helper locks and the HELPER runtime which efficiently supports helper locks. Chapter 4 discusses another use for HELPER's parallel region construct, namely for improving the compatibility of Cilk with legacy-C code.

The second part — Chapters 5 through 7 — discusses transaction-based approaches to synchronization in parallel programs. Chapter 5 presents the transactional computation framework, a formal model which is useful for understanding the behavior of TM systems with nested transactions. Chapter 6 presents CWSTM, a design of a software TM system that supports transactions with nested parallelism. Chapter 7 describes ownership-aware TM, a TM design which makes open-nested transactions safer and more intuitive to use.

Chapter 8 concludes with a summary of the topics covered in this dissertation.

Finally, Appendices A through C contain supplemental material which may be useful when reading this dissertation. For readers unfamiliar with Cilk, Appendix A reviews background material on Cilk's programming model, runtime scheduler, and theoretical time and space bounds. Appendix B summarizes the notation used in this dissertation. Appendix C includes the details of the correctness proof for operational models that are described in Chapters 5 and 7.

Chapter 2

Task-Graph Synchronization Using Work-Stealing

Many parallel programming problems can be expressed using a ***task graph***: a directed acyclic graph (DAG) $\mathcal{D} = (V_\mathcal{D}, E_\mathcal{D})$, where every node $A \in V_\mathcal{D}$ represents some task with computation $\text{COMPUTE}(A)$, and a directed edge $(A,B) \in E_\mathcal{D}$ represents the constraint that B's computation depends on results computed by A. ***Executing*** a task graph means assigning every node $A \in V_\mathcal{D}$ to a processor to execute at a given time and executing $\text{COMPUTE}(A)$ at that time such that every predecessor of A has finished its computation beforehand. In this chapter, I show that a dynamic-threading platform can provide arbitrary task-graph synchronization which composes with ordinary fork-join programs. I also prove efficient theoretical bounds on completion time for task-graph execution using an ordinary work-stealing scheduler.[1]

Task graphs come in two flavors. A ***static*** task graph $\mathcal{D}$ is one where the structure of $\mathcal{D}$ (the nodes $V_\mathcal{D}$ and edges $E_\mathcal{D}$) are known before execution of a task graph begins. In a ***dynamic*** task graph, the nodes and edges are created on the fly at runtime, with creation and execution of task-graph nodes potentially happening concurrently.

Executing a task graph $\mathcal{D}$ in parallel requires constructing a ***schedule*** for $\mathcal{D}$ — a mapping of nodes of $V_\mathcal{D}$ to processors and execution times. A scheduler for executing a task graph $\mathcal{D}$ can be classified as either clairvoyant or nonclairvoyant. A ***clairvoyant task-graph scheduler*** is a scheduler that is aware of the time required to execute $\text{COMPUTE}(A)$ for each $A \in V_\mathcal{D}$, and may construct a schedule offline.[2] In contrast, a ***nonclairvoyant task-graph scheduler*** does not know the compute times of nodes in advance, and must make scheduling decisions online at runtime to effectively load-balance execution across processors.

Many efficient approximation algorithms and heuristics exist for clairvoyant scheduling of task graphs. As summarized by Kwok and Ahmad in their survey [86], the problem of scheduling a task graph when the compute times of task nodes are known has been stud-

[1] These results on task-graph execution represent joint work [9] with Kunal Agrawal and Charles E. Leiserson.

[2] Existing literature on task-graph scheduling sometimes uses the term "static" to refer to a task graph where both the structure of nodes and edges *and* the compute times of task nodes are known a priori. This stronger definition corresponds to a task graph that is static as defined in this chapter and which can be executed using a clairvoyant scheduler.

ied extensively in a variety of computational models.[3] Unfortunately, as researchers have noted (e.g., [104]), clairvoyant scheduling can be difficult to apply in practice; the complexity of modern computer hardware and software systems greatly complicates the accurate estimation of the compute times of task-graph nodes, and for some scheduling algorithms or applications, the offline processing required to generate a clairvoyant schedule may be prohibitively expensive.

Thus, this chapter considers only nonclairvoyant task-graph schedulers. Most nonclairvoyant schedulers for generic task graphs rely on a ***task pool***, a data structure that dynamically maintains a collection of ***ready*** tasks whose predecessors have completed. Processors remove and work on ready tasks, posting new tasks to the task pool as dependencies are satisfied. Using task pools for scheduling avoids the need for accurate time estimates for the computation of each task, but maintaining a task pool may introduce runtime overheads.

One way to reduce the runtime overhead of task pools is to impose additional structure on the task graphs so that one can optimize the task-pool implementation. For example, Hoffman, Korch, and Rauber [68, 81] describe and empirically evaluate a variety of implementations of task pools in software and hardware. They focus on the case where tasks have hierarchical dependencies, i.e., a parent task depends only on child tasks that it creates. In their evaluation of software implementations of task pools, they observe that distributed task pools based on dynamic "task stealing" perform well and provide the best scalability.

Dynamic task-stealing is closely related to the ***work-stealing*** scheduling strategy used in dynamic-threading languages such as Cilk [28, 51], Cilk++ [72, 93], Fortress [13], X10 [37], and parallel runtime libraries such as Hood [31] and Intel Threading Building Blocks [110]. A work-stealing scheduler maintains a distributed collection of ready queues where processors can post work locally. Typically, a processor finds new work from its own work queue, but if its work queue is empty, it ***steals*** work from the work queue of another processor, typically chosen at random. Blumofe and Leiserson [30] provided the first work-stealing scheduling algorithm coupled with an asymptotic analysis showing that their algorithm performs near optimally.

Dynamic-threading languages support fork-join constructs for parallelism, which allow programmer to easily express ***series-parallel*** [120, 121] task graphs. For example, Figure 2-1 shows a Cilk program for the task graph shown in Figure 1-3(b). Unfortunately, these languages generally do not support task graphs with *arbitrary* dependencies however. One cannot express a task-graph computation with arbitrary dependency edges using only `spawn` and `sync` keywords, since one can show (e.g., as in [45]) that the execution of a Cilk program conceptually generates only computation DAGs which are series-parallel.[4]

Thus, conventional wisdom suggests that to execute a non-series-parallel task graph such as Figure 1-3(a) in a fork-join platform, one should convert it into a series-parallel DAG, such as the one in Figure 1-3(b). Converting an arbitrary task graph into a series-parallel DAG often introduces superfluous dependencies, however, which can potentially

[3] Hu in [69] presents one of the earliest algorithms for clairvoyant task-graph scheduling, in the context of parallel assembly-line scheduling. In general, the problem of finding a minimum-time clairvoyant schedule on P processors for a static task graph $\mathcal{D}$ is known to be NP-complete [119].

[4] Intuitively, a series-parallel DAG is one that can be recursively built using only series and parallel compositions. For more details, see Appendix A, which reviews how Cilk programs generate a series-parallel computation DAG (Section A.2), or equivalently, a series-parallel computation tree (Section A.4).

```
void TaskGraphExecute() {
  BlockCompute(1, 2, 5, 6);
  spawn BlockCompute(3, 4, 7, 0);
  BlockCompute(9, 0, 0, 14);
  sync;
  BlockCompute(11, 12, 15, 16);
}
void BlockCompute(int v1, int v2, int v3, int v4) {
  if (v1 > 0) Compute(v1);
  if (v2 > 0) spawn Compute(v2);
  if (v3 > 0) Compute(v3);
  sync;
  if (v4 > 0) Compute(v4);
}
```

Figure 2-1: A Cilk program for executing the task graph in Figure 1-3(b). The Cilk keywords of `spawn` and `sync`, which express potential parallelism, appear in bold. The `Compute` method performs a computation for a task based on the task-node number passed in as argument. The `BlockCompute` method calls `Compute` for a 2×2 grid subgraph of Figure 1-3(b).

limit parallelism in the task-graph execution. For example, in Figure 1-3(a), the node 11 can execute after the set of nodes $\{1,2,3,5,6,7,9\}$ completes, but after a series-parallel conversion to Figure 1-3(b), node 11 also must wait for nodes 4 and 14 to complete.

Alternatively, to support execution of arbitrary task graphs, one might try to add special runtime support to a fork-join platform. This approach can be tricky to design and implement, however, because one must carefully consider how new mechanisms interact with the existing scheduler and programming model. Also, runtime modifications can potentially invalidate the theorems that guarantee the theoretical efficiency of work-stealing.

Contributions

In this chapter, I demonstrate that efficient execution of arbitrary task graphs in a fork-join dynamic-threading platform is possible without modifying the underlying work-stealing scheduler or introducing extra dependency edges. Conventional wisdom suggests that in a fork-join platform, executing a task graph with arbitrary dependency edges requires either special runtime support or requires conversion of the task graph into a series-parallel task graph which has less parallelism. I show, however, that both of these "requirements" can be avoided. In particular, this chapter describes the following contributions:

- Nabbit, the first library in a fork-join platform for provably efficient parallel execution of task graphs with arbitrary dependencies.
- Efficient theoretical bounds on the time required to execute task graphs using a nonclairvoyant scheduler. More precisely, using a work-stealing scheduler, one can guarantee asymptotically optimal completion-time bounds for task graphs whose nodes have constant in-degree and out-degree.

- An extension of Nabbit and its theory to dynamic task graphs.

Nabbit demonstrates a composable programming interface for specifying task graphs in a fork-join platform: it allows the computation of individual task-graph nodes to be parallel functions, and it allows ordinary fork-join code to run efficiently in parallel with a task-graph execution.

Chapter Outline

The remainder of this chapter is organized as follows.

First, this chapter presents results for static task graphs. Section 2.1 describes the programming interface for static Nabbit, an interface which enables a fork-join dynamic-threading platform to execute static task graphs. Section 2.2 determines theoretical bounds on the time required for a Cilk-like work-stealing scheduler to execute static task graphs specified using static Nabbit, and shows that these bounds are asymptotically optimal for task graphs with constant degree. Section 2.3 evaluates static Nabbit empirically on a irregular dynamic-programming benchmark, which demonstrates that Nabbit is competitive with and in some cases can even outperform traditional series-parallel algorithms for the same benchmark.

Next, this chapter discusses alternative programming interfaces for specifying static task graphs in Nabbit. Section 2.4 describes the "data-block interface" for Nabbit, an interface which is designed to simplify the process of creating static task graphs for computations which fit into a "data-block access" programming pattern. Section 2.5 presents empirical results from an asynchronous Cholesky factorization benchmark, which demonstrates the effectiveness of Nabbit and the data-block interface.

Finally, this chapter extends the theory and implementation of Nabbit to dynamic task graphs. Section 2.6 presents the interface and theoretical analysis of dynamic Nabbit, which executes dynamic task graphs. Section 2.7 evaluates the performance of static and dynamic Nabbit on a random-graph microbenchmark. Section 2.8 discusses related work and potential future work on task-graph execution in fork-join platforms.

2.1 Nabbit for Static Task Graphs

This section introduces Nabbit by describing the interface and implementation of ***static Nabbit***, a version of Nabbit optimized to execute static task graphs. This interface and implementation demonstrate that arbitrary static task graphs can be easily specified and executed in a fork-join dynamic-threading platform.

Interface

In static Nabbit, programmers specify task graphs by creating nodes that extend from a base DAGNODE object, specifying the dependencies of each node, and providing a COMPUTE method for each node.

As a concrete example, consider a dynamic program on an $n \times n$ grid which takes an $n \times n$ input matrix s and computes the value $M(n,n)$ based on the following recurrence:

$$M(i,j) = \begin{cases} \max\begin{cases} M(i-1,j)+s(i-1,j) \\ M(i,j-1)+s(i,j-1) \end{cases} & \text{if } i \geq 1 \text{ and } j \geq 1 \,, \\ 0 & \text{if } i < 1 \text{ or } j < 1 \,. \end{cases} \tag{2.1}$$

One can formulate this problem as a task graph with a task node for every cell $M(i,j)$ and dependencies on the cells $M(i-1,j)$ and $M(i,j-1)$. This dynamic program generates a task graph which is a 2d $n \times n$ grid graph, a graph with structure, but nevertheless non-series-parallel.

Figure 2-2 formulates this problem as a task graph. The code constructs a node for every cell $M(i,j)$, with the node's class extending from a base DAGNODE class. The programmer uses two methods of this base class: ADDDEP specifies a predecessor node on which the current node depends, and EXECUTE tells Nabbit to execute a task graph using the current node (with no predecessors) as a source node.

In the example from Figure 2-2, the COMPUTE method for each task-graph node is a short, serial section of code. More generally, however, Nabbit allows programmers to use `spawn` and `cilk_for` to expose additional parallelism within a node's COMPUTE method. This capability arises naturally because Nabbit is implemented directly in Cilk++, without requiring any modifications to the Cilk++ runtime. Also, programmers can easily use Nabbit as part of a larger Cilk++ program, since the scheduler can efficiently run a task-graph execution using Nabbit in parallel with ordinary fork-join code written in Cilk++.

Implementation

To perform bookkeeping for a task-graph execution, static Nabbit maintains the following fields for each task-graph node A:

- **Successor array:** An array of pointers to A's immediate successors in the task graph.
- **Join counter:** A variable whose value tracks the number of A's immediate predecessors that have not completed their COMPUTE method.
- **Predecessor array:** An array of pointers to A's immediate predecessors in the task graph, i.e., the nodes on which A depends.[5]

The EXECUTE method for a node A is implemented as a call to the COMPUTEANDNOTIFY method for A. The code for this method is shown in Figure 2-3.

2.2 Analysis of Static Nabbit

This section provides a theoretical analysis of the performance of the Nabbit library when executing a static task graph on multiple processors, which shows that Nabbit executes

[5] Maintaining this array is not always necessary. Nabbit maintains this array so that A's COMPUTE method can conveniently access its immediate predecessors. In examples such as Figure 2-2, one can also find a node's predecessors through pointer and index calculations.

```
class DynProgDag {
  int n; int* s; MNode* g;
  DynProgDag(int n_, int* s_): n(n_), s(s_) {
    g = new MNode[(n+1)*(n+1)];
    for (int i = 0; i <= n; ++i) {
      for (int j = 0; j <= n; ++j) {
        int k = (n+1)*i+j;
        g[k].pos = k; g[k].dag = (void*) this;
        if (i > 0) {g[k].AddDep(&MNode[k-(n+1)])};
        if (j > 0) {g[k].AddDep(&MNode[k-1])};
      }
    }
  }
  int Execute() {
    g[0]->Execute();
  }
};
class MNode: public DAGNode {
  int res;
  void Compute() {
    this->res = 0;
    for (int i = 0; i < predecessors.size(); i++) {
      MNode* pred = predecessors.get(i);
      int pred_val = pred->res + s[pred->pos];
      this->res = MAX(pred_val, res);
    }
  }
};
```

Figure 2-2: Cilk++ code using Nabbit to solve the dynamic program in Equation (2.1). The identifiers in bold correspond to classes or methods specific to Nabbit. The code constructs a task-graph node for every cell $M(i,j)$. The array g of nodes is stored in row-major layout. This example code executes the task graph by calling EXECUTE on g[0], the source of the graph.

```
COMPUTEANDNOTIFY($A$)
1  COMPUTE($A$)
2  parallel for all $B \in A.successors$
3      $val$ = ATOMDECANDFETCH($B.join$)
4      if $val == 0$
5          COMPUTEANDNOTIFY($B$)
```

Figure 2-3: Static task-graph execution in Nabbit. COMPUTEANDNOTIFY computes a node A and then greedily spawns the computation for any immediate successors of A which are enabled by A's computation. The **parallel for** in line 2 indicates that the loop iterations are spawned in binary-tree fashion, and all can potentially run in parallel.

static task graphs with constant degree in time which is asymptotically optimal. To analyze the runtime of Nabbit, we employ a work/span analysis (e.g., as in [38, Chapter 27]) and calculate upper bounds on the work and span[6] of the executions of the code in Figure 2-3. Then, we translate these bounds into completion-time bounds for Nabbit using known theoretical bounds on the completion time of fork-join parallel programs scheduled with randomized work-stealing [16,30]. For a review of the computation DAG model used in this analysis, see Section A.2.

Definitions

To analyze the performance of Nabbit, we require some definitions. Consider a task graph $\mathcal{D} = (V_{\mathcal{D}}, E_{\mathcal{D}})$. For each node $A \in V_{\mathcal{D}}$, let $\texttt{ipred}(A)$ denote the set of immediate predecessors of A, and let $\texttt{isucc}(A)$ denote the immediate successors of A. Let $\texttt{outDeg}(A) = |\texttt{isucc}(A)|$ and $\texttt{inDeg}(A) = |\texttt{ipred}(A)|$ be the out-degree and in-degree of A, respectively. For simplicity in stating the results, we assume that for a task graph $\mathcal{D}$, every node is a successor of a unique ***source*** node $s_{\mathcal{D}}$ with no incoming edges, and a predecessor of a unique ***sink*** node $t_{\mathcal{D}}$ with no outgoing edges. Let $\texttt{paths}(A,B)$ be the set of all paths in $\mathcal{D}$ from node A to node B.

Every execution of a task graph invokes COMPUTEANDNOTIFY(A) exactly once for each $A \in V_{\mathcal{D}}$. For many task graphs, such as the one in Figure 2-4, the execution of COMPUTEANDNOTIFY can be nondeterministic, since COMPUTE(A) may be invoked by a different immediate predecessor depending on how the runtime schedules the computation on multiple processors. Each possible execution can be represented as a computation DAG $\mathcal{G}$, which should not be confused with the task graph $\mathcal{D}$ itself. The nodes of the computation DAG are serial chains of executed instructions, and the edges represent dependencies between them.

We shall define several notations for subgraphs of a computation DAG. For a particular computation DAG $\mathcal{G}$ and a task-graph node A, let $CN^{\mathcal{G}}(A)$ be the subgraph corresponding to the call COMPUTEANDNOTIFY(A), and let $com^{\mathcal{G}}(A)$ be the subgraph corresponding to

[6] "Span" is sometimes called "critical-path length" or "computation depth" in the literature.

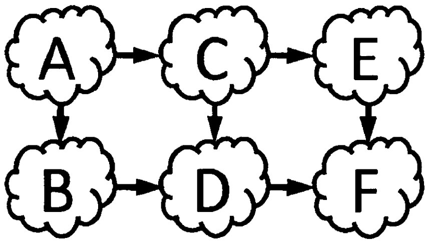


Figure 2-4: A task graph for computing $M(2,3)$ using Equation (2.1). The execution of ComputeAndNotify(A) is nondeterministic — it recursively calls both ComputeAndNotify(B) and ComputeAndNotify(C), but in a particular execution, only one, but not both of these methods recursively calls ComputeAndNotify(D).

Compute(A). For any subgraph $\mathcal{G}'$ of a computation DAG, define the ***work*** of $\mathcal{G}'$, denoted by $T_1(\mathcal{G}')$, to be the sum of the execution times of all the nodes in $\mathcal{G}'$. Similarly, define the ***span*** of $\mathcal{G}'$, denoted by $T_\infty(\mathcal{G}')$, to be the longest execution time along any path through $\mathcal{G}'$. We overload notation so that when the superscript $\mathcal{G}$ is omitted, we mean the maximum of the quantity over all possible computation DAGs generated by executions of the task graph $\mathcal{D}$. For example, $T_1(CN(A))$ denotes the maximum work for ComputeAndNotify(A) over all possible executions of $\mathcal{D}$.

To analyze Nabbit's running time on a task graph $\mathcal{D}$ with source $s_\mathcal{D}$, examine the execution of ComputeAndNotify$(s_\mathcal{D})$. The total work done by a computation $\mathcal{G}$ of $\mathcal{D}$ is $T_1(CN^\mathcal{G}(s_\mathcal{D}))$, and the span is $T_\infty(CN^\mathcal{G}(s_\mathcal{D}))$. Since the computation DAG is nondeterministic, we consider the maximum of these values — namely, $T_1(CN(s_\mathcal{D}))$ and $T_\infty(CN(s_\mathcal{D}))$ — and use these values as upper bounds in the analysis.

Work Analysis

The following lemma calculates the work required by Nabbit to execute a task graph $\mathcal{D}$.

Lemma 2.1. *Any execution of $\mathcal{D}$ using Nabbit has work*

$$T_1(CN(s_\mathcal{D})) \leq \sum_{A \in V_\mathcal{D}} T_1(com(A)) + O(|E_\mathcal{D}|) + \psi_W ,$$

where

$$\psi_W = O\left(\sum_{B \in V_\mathcal{D}} \texttt{inDeg}(B) \cdot \min\{\texttt{inDeg}(B), P\} \right) .$$

Proof. The first term arises from the work of the Compute functions. The second term $O(|E_\mathcal{D}|)$ bounds the work of traversing $\mathcal{D}$, assuming no contention.

The third term ψ_W covers the contention cost on the join counter. For each node $B \in V_\mathcal{D}$, we decrement the join counter of B $\texttt{inDeg}(B)$ times. Assuming that processors can queue when there is contention on the atomic decrement, in the worst case each decrement takes $O(\min\{\texttt{inDeg}(B), P\})$ time. □

Span Analysis

The nondeterministic nature of task-graph execution in Nabbit complicates the direct calculation of $T_\infty(CN(s_{\mathcal{D}}))$. Consequently, our strategy is to construct a new, deterministic computation DAG $\mathcal{G}^*$ whose span upper-bounds the span of COMPUTEANDNOTIFY$(s_{\mathcal{D}})$, and then analyze the span of $\mathcal{G}^*$. We define the method COMPUTEANDNOTIFY$^*(A)$ to be the same as the original method, except that line 4 in Figure 2-3 is omitted. In other words, the method COMPUTEANDNOTIFY$^*(A)$ always makes recursive calls on all of A's successors. Let $CN^*(A)$ be the computation DAG corresponding to this modified method, and let $\mathcal{G}^*$ be the computation DAG for COMPUTEANDNOTIFY$^*(s_{\mathcal{D}})$. Figure 2-5 shows $\mathcal{G}^*$ for the task graph shown in Figure 2-4. Since any computation $\mathcal{G}$ forms a subDAG of $\mathcal{G}^*$, we have $T_\infty(CN^*(A)) \geq T_\infty(CN^{\mathcal{G}}(A))$. Then, we bound $T_\infty(CN^*(s_{\mathcal{D}}))$ using Lemma 2.2.

Lemma 2.2. *Any execution of $\mathcal{D}$ using Nabbit has span*

$$T_\infty(CN^*(s_{\mathcal{D}})) \leq \max_{\mathsf{p} \in paths(s_{\mathcal{D}}, t_{\mathcal{D}})} \left\{ \sum_{X \in \mathsf{p}} n(X) + \sum_{(X,Y) \in \mathsf{p}} \psi_S(X,Y) \right\} ,$$

where

$$\begin{aligned} n(X) &= T_\infty(com(X)) + O\left(\lg(\mathit{outDeg}(X))\right) , \\ \psi_S(X,Y) &= O(\min\{\mathit{inDeg}(Y), P\}) . \end{aligned}$$

Proof. For each node X, the method COMPUTEANDNOTIFY$^*(X)$ enables all of X's immediate successors, with the recursive calls to COMPUTEANDNOTIFY* operating in parallel. As Figure 2-5 illustrates, any path through the computation DAG $CN^*(X)$ contains the COMPUTE of only those nodes along a corresponding path through $\mathcal{D}$.

Along any path p, the term $n(X)$ accounts for $T_\infty(com(X))$, the span of X itself, plus the additional span $O(\lg(\texttt{outDeg}(X)))$ required to spawn recursive calls along X's outgoing edges using a **parallel for** loop. In Cilk++, a **parallel for** loop[7] spawns iterations in the form of a balanced binary tree, and thus the depth of the tree is logarithmic.

The term $\psi_S(X,Y)$ accounts for the contention cost of decrementing the join counter for Y, where Y is a successor of X. In the worst case, this decrement might wait for $\min\{\texttt{inDeg}(Y), P\}$ other decrements. □

Completion-Time Bounds

Lemmas 2.1 and 2.2 bound the work and span of the computation DAG using characteristics of the task graph. Now, we relate these bounds back to the time it takes to execute a task

[7] The Cilk++ keyword for a **parallel for** loop is actually `cilk_for`.

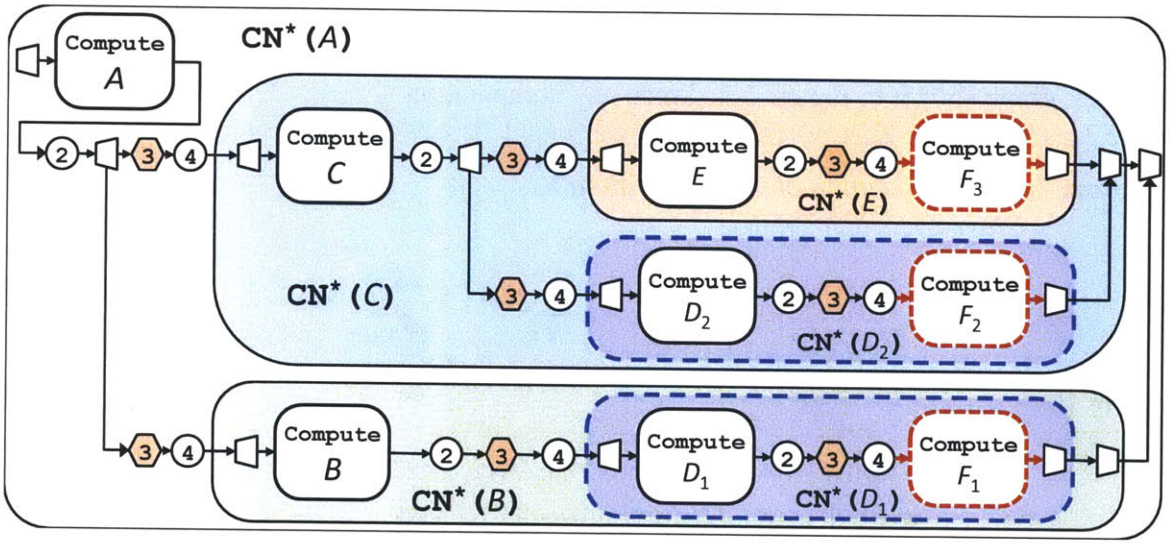


Figure 2-5: The computation DAG $CN^*(A)$ for the execution of the task graph from Figure 2-4. Numbers correspond to line numbers from Figure 2-3. Hexagons correspond to atomic decrements of join counters, which may require synchronization. An execution of COMPUTEANDNOTIFY(A) generates a DAG which is a subDAG of $CN^*(A)$. Each possible subDAG contains exactly one element drawn from the $\{D_1, D_2\}$, and one from $\{F_1, F_2, F_3\}$. Four subDAGs of $CN^*(A)$ are possible: (A,B,C,D_1,E,F_1), (A,B,C,D_1,E,F_3), (A,B,C,D_2,E,F_2), or (A,B,C,D_2,E,F_3).

graph $\mathcal{D} = (V_{\mathcal{D}}, E_{\mathcal{D}})$ on an ideal parallel computer. Let $T_1(\mathcal{D})$ be the work of $\mathcal{D}$ — the time it takes to execute $\mathcal{D}$ on a single processor. We have that

$$T_1(\mathcal{D}) = \sum_{A \in V_{\mathcal{D}}} T_1(com(A)) + O(|E_{\mathcal{D}}|) ,$$

since any execution of $\mathcal{D}$ executes the COMPUTE method of every node once and must traverse every edge. Similarly, let $T_\infty(\mathcal{D})$ be the span of $\mathcal{D}$ — the time it takes to execute $\mathcal{D}$ on an ideal parallel computer with an infinite number of processors. Define V_∞ as the number of nodes on the longest path in $\mathcal{D}$ from the source $s_{\mathcal{D}}$ to the sink $t_{\mathcal{D}}$. We have

$$T_\infty(\mathcal{D}) = \max_{\mathtt{p} \in \mathtt{paths}(s_{\mathcal{D}}, t_{\mathcal{D}})} \left\{ \sum_{X \in \mathtt{p}} T_\infty(com(X)) \right\} + O(V_\infty) ,$$

since nodes along any path through $\mathcal{D}$ cannot execute in parallel.

Let $T_P(\mathcal{D})$ denote the time Nabbit requires to execute a task graph $\mathcal{D}$ on P processors. By the work and span laws [38, p. 780], we have $T_P(\mathcal{D}) \geq \max\{T_1/P, T_\infty\}$.

Using Lemmas 2.1 and 2.2 and the analysis of a Cilk-like work-stealing scheduler, we obtain the following completion-time bound for Nabbit.

Theorem 2.3. *Let $\mathcal{D} = (V_{\mathcal{D}}, E_{\mathcal{D}})$ be a task graph with maximum in-degree Δ_i and maximum out-degree Δ_o. With probability at least $1 - \varepsilon$, Nabbit executes $\mathcal{D}$ on P processors in time*

$$T_P(\mathcal{D}) = O\left(\frac{T_1(\mathcal{D})}{P} + T_\infty(\mathcal{D}) + \lg\left(\frac{P}{\varepsilon}\right) + V_\infty \lg \Delta_o + \psi(\mathcal{D}) \right) ,$$

where

$$\psi(\mathcal{D}) = O\left(\left(\frac{|E_{\mathcal{D}}|}{P} + V_\infty \right) \min\{\Delta_i, P\} \right) .$$

Proof. Blumofe and Leiserson in [30] show that a Cilk-like work-stealing scheduler completes a computation with work T_1 and span T_∞ in time $O(T_1/P + T_\infty + \lg(P/\varepsilon))$ on P processors with probability at least $1 - \varepsilon$. (This result is reviewed in Theorem A.1 in Section A.3.) To bound the completion time, we relate the work and span of the computation DAG $CN(s_{\mathcal{D}})$ to T_1 and T_∞. Bounding the contention term in Lemma 2.1 using Δ_i, we have

$$T_1(CN(s_{\mathcal{D}})) = T_1(\mathcal{D}) + O(|E_{\mathcal{D}}| \cdot \min\{\Delta_i, P\}) ,$$

since the sum of the in-degrees of the nodes in a graph is the cardinality of the edge set. Similarly, one can use Δ_i and Δ_o in Lemma 2.2 to show that

$$T_\infty(CN(s_{\mathcal{D}})) = T_\infty(\mathcal{D}) + O\left(V_\infty \lg \Delta_o + V_\infty \cdot \min\{\Delta_i, P\}\right) .$$

The theorem follows directly from the result in [30]. □

The $V_\infty \lg \Delta_o$ term accounts for the additional span required to visit all the successors of a node in parallel. Whereas Nabbit allows a programmer to specify task graphs whose nodes have large degrees, fork-join languages such as Cilk++ produce computation DAGs where every node has constant out-degree, in which case this term is absorbed in the T_∞ term. Even when the out-degree is not constant, one would expect this term to be dominated by the T_∞ term if the task-graph nodes contain a reasonable amount of work.

The $\psi(\mathcal{D})$ term in Theorem 2.3 is an upper bound on the contention due to synchronization during the task-graph execution. The term $|E_{\mathcal{D}}|/P + V_\infty$ is a bound on the P-processor execution time needed for a parallel traversal of $\mathcal{D}$, including updating the join counters on every edge. The extra factor of $\min\{\Delta_i, P\}$ appears because we assume worst-case contention, i.e., that processors wait as long as possible on every decrement of a join counter. In the case where every node has constant degree, the term $\psi(\mathcal{D})$ is absorbed by $T_1/P + T_\infty$, and thus in this case the running time in Theorem 2.3 is asymptotically optimal. Even when the degree is more than constant, worst-case contention is unlikely to occur in practice for every decrement.

Although the contention term in Theorem 2.3 grows linearly with the maximum out-degree, in principle, one can modify the scheduler to asymptotically eliminate the contention term $\psi(\mathcal{D})$ from the completion-time bound.

Corollary 2.4. *There exists a work-stealing scheduler that can execute any static task graph $\mathcal{D}$ with maximum degree Δ on P processors in time*

$$T_P(\mathcal{D}) = O\left(\frac{T_1(\mathcal{D})}{P} + T_\infty(\mathcal{D}) + V_\infty \lg \Delta + \lg\left(\frac{P}{\varepsilon}\right)\right)$$

with probability at least $1 - \varepsilon$.

Proof. Given $\mathcal{D} = (V_{\mathcal{D}}, E_{\mathcal{D}})$, the scheduler creates an equivalent task graph $\mathcal{D}'$ in which every node has constant degree by adding dummy nodes to $\mathcal{D}$. This construction adds at most $O(|E_{\mathcal{D}}|)$ dummy nodes and extends the longest path by $O(V_\infty \lg \Delta)$ nodes. By Theorem 2.3, executing $\mathcal{D}'$ with Nabbit gives us the desired bound. □

This modification to Nabbit was not implemented, since in practice for relatively small values of P, the overheads of this modification are likely to be more expensive than simply suffering the contention.

2.3 Empirical Results for Static Nabbit

This section presents an empirical evaluation of the performance of static Nabbit on an irregular dynamic-programming benchmark based on the Smith-Waterman [112] dynamic-programming algorithm used in computational biology. This empirical study indicates that Nabbit is competitive with and can often outperform other series-parallel implementations of the same benchmark. These results demonstrate that the ability to execute a task graph with arbitrary dependencies can improve overall performance and scalability, despite the added overhead that Nabbit requires to track dependencies during work-stealing that are not series-parallel.

An Irregular Dynamic Program

One common application for task-graph execution is dynamic-programming computations with irregular structure. Consider an irregular dynamic program on a 2-dimensional grid that computes a value $M(i,j)$ based on the following set of recursive equations:

$$\begin{aligned}
E(i,j) &= \max_{k\in\{0,1,\ldots,i-1\}} M(k,j)+\gamma(i-k) \; ; \\
F(i,j) &= \max_{k\in\{0,1,\ldots,j-1\}} M(i,k)+\gamma(j-k) \; ; \\
M(i,j) &= \max\begin{cases} M(i-1,j-1)+s(i,j) \; , \\ E(i,j) \; , \\ F(i,j) \; . \end{cases}
\end{aligned} \tag{2.2}$$

In Equation (2.2), the functions $s(i,j)$ and $\gamma(z)$, and the base cases (not shown) can all be computed in constant time. As described in [95], this particular dynamic program models the computation used for the Smith-Waterman [112] algorithm with a general penalty gap function γ. This dynamic program is irregular because the work for computing the cells is not the same for each cell. Specifically, $\Theta(i+j)$ work must be done to compute cell $M(i,j)$. Therefore, in total, computing $M(m,n)$ using Equation (2.2) requires $\Theta(mn(m+n))$ work, or equivalently, $\Theta(n^3)$, when $m=n$.

I consider three types of parallel algorithms for solving this dynamic program. The first type of algorithm uses Nabbit to create and execute a task graph. The second type performs a wavefront computation, and the third type uses a divide-and-conquer approach. These latter two types are specific algorithms for solving this particular dynamic program, and both algorithm types generate series-parallel computation DAGs that can be coded directly in a fork-join language such as Cilk++. For each of the three algorithms, in order to improve cache locality and to amortize overheads, the cells are grouped into $B\times B$ blocks, where block (b_i,b_j) represents the block with upper-left corner at cell (b_iB,b_jB).

The first algorithm expresses the dynamic program in Equation (2.2) by creating a task graph $\mathcal{D}$ similar to the code in Figure 2-2, except that each node of the task graph computes a $B\times B$ block of cells. The COMPUTE method for each node computes the values of M for the entire block serially. Block (b_i,b_j) depends on (at most) two blocks (b_i-1,b_j) and (b_i,b_j-1). Although block (b_i,b_j) depends on the entire block row b_i to the left of (b_i,b_j) and the entire block column b_j above (b_i,b_j), it is sufficient to create a task graph with dependencies only from (b_i-1,b_j) and (b_i,b_j-1) because transitivity ensures that the other dependencies are satisfied.

The second algorithm performs a wavefront computation. The computation is divided into $\lceil n/B\rceil$ phases, with phase i handling the ith block antidiagonal of the grid. Since blocks within a single antidiagonal can be computed independently, in each phase the blocks along the antidiagonal are executed using a **parallel for** loop (with grainsize of 1).

The third algorithm is a divide-and-conquer algorithm for the dynamic program that divides the grid into 4 subgrids and then computes the cells in each subgrid recursively. This algorithm computes the upper-left subgrid first, then the lower-left and upper-right subgrids in parallel next, and then finally the lower-right subgrid. Figure 2-6 gives Cilk pseudocode for implementing this divide-and-conquer algorithm.

```
ComputeM(n) { ComputeMHelper(0, 0, n); }
ComputeMHelper(i, j, n) {
  if (n <=B) { ComputeMBase(i, j, n); }
  else {
    ComputeMHelper(i, j, n/2);
    spawn ComputeMHelper(i+n/2, j, n/2);
    ComputeMHelper(i, j+n/2, n/2);
    sync;
    ComputeMHelper(i+n/2 j+n/2, n/2);
  }
}
```

Figure 2-6: Cilk pseudocode for a parallel divide-and-conquer algorithm to compute $M(n,n)$ as defined by Equation (2.2). The `ComputeMHelper(i, j, n)` method computes M for an $n \times n$ grid with upper-left corner at cell (i,j). For simplicity, we only show code for the case when $m = n = 2^k \cdot B$ for some integer k.

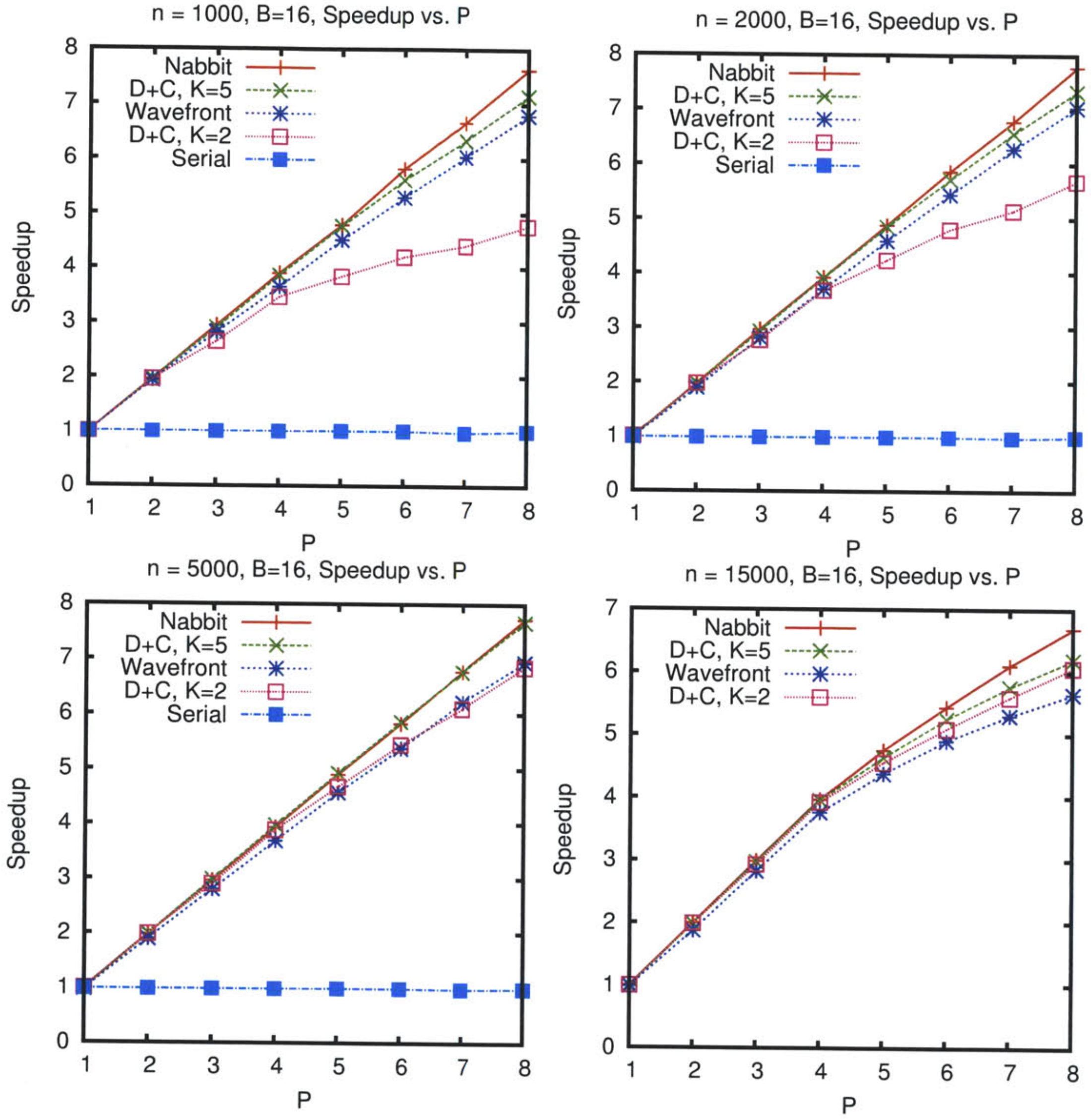
One can show that asymptotically, if $n > B$, the parallelism (work divided by span) of both the task-graph and the wavefront algorithms is $\Theta(n/B)$, since both algorithms have $O(n^3)$ work and $\Theta(n^2B)$ span. The span of $\mathcal{D}$ consists of $\Theta(n/B)$ blocks, with at least half the blocks requiring $\Theta(nB^2)$ work.

One can also show that the divide-and-conquer algorithm has a span of $\Theta(n^{\lg 6}B^{3-\lg 6}) \approx \Theta(n^{2.585}B^{0.415})$. Thus, its theoretical parallelism is $\Theta((n/B)^{\lg 6}) \approx \Theta((n/B)^{0.415})$, which is lower than the parallelism of the task-graph or wavefront algorithms. This algorithm incurs lower synchronization overhead than the other two, however. One can asymptotically increase the parallelism of a divide-and-conquer algorithm by dividing into more subproblems (e.g., divide one n-by-n grid into K^2 subgrids, each of size approximately $n/K \times n/K$), but the code becomes more complex. In the limit, the resulting algorithm is equivalent to the wavefront computation.

Dynamic Program Implementations

I compared four parallel implementations of Equation (2.2), based on (1) a task graph using Nabbit, (2) a wavefront algorithm, (3) a divide-and-conquer algorithm, dividing each dimension of the matrix by $K = 2$, and (4) a divide-and-conquer algorithm, dividing each dimension by $K = 5$. For a fair comparison, all implementations use the same memory layout and reuse the same code for core methods, e.g., computing a single $B \times B$ block. Each implementation looks up values for s and γ from arrays in memory.

Since memory layout impacts performance significantly for large problem sizes, both $M(i,j)$ and $s(i,j)$ are stored in a cache-oblivious [50] layout. Because the calculations of $E(i,j)$ and $F(i,j)$ require scanning along a column and row, respectively, simply storing M in a row-major or column-major layout would be suboptimal for one of these calculations. To support efficient iteration over rows and columns, the code used dilated integers [126] as indices into the grid and employed techniques for fast conversion between dilated and normal integers from [108].



n = 1000, B=16, Speedup vs. P

n = 2000, B=16, Speedup vs. P

n = 5000, B=16, Speedup vs. P

n = 15000, B=16, Speedup vs. P

Figure 2-7: The performance of the dynamic program on an $n \times n$ grid. Speedups are normalized against the fastest run with $P = 1$. For $n = 1000$, the baseline is 2.05 s for serial execution. For $n = 2000$, the baseline is 16.6 s using divide-and-conquer with $K = 2$. For $n = 5000$, the baseline is 263 s using divide-and-conquer with $K = 2$. For $n = 15000$, the baseline is 8279 s using Nabbit.

Two different types of experiments were run for the dynamic program. The first experiment measures the parallel speedups for the four different algorithms on various problem sizes. The second measures the sensitivity of the algorithms to different choices in block size. Both experiments were run on a multicore machine with 8 total cores.[8]

The first experiment compared the speedup provided by the four algorithms, with a fixed block size at $B = 16$. Each task-graph node was responsible for computing a 16×16 block of the original grid, and the wavefront and divide-and-conquer algorithms operated on blocks of size 16×16 in the base case.

Figure 2-7 shows the speedup on P processors for $n \in \{1000, 2000, 5000, 15000\}$. Nabbit outperforms the other implementations in all these experiments. For example, at $n = 1000$, the divide-and-conquer algorithm achieves speedup of 5 on 8 processors, while the Nabbit implementation exhibits a speedup of about 7. This result is not surprising, since the task-graph execution has a higher asymptotic parallelism than the divide-and-conquer algorithm. Even though the wavefront algorithm has the same asymptotic parallelism as the task-graph execution, however, it performs worse than the divide-and-conquer algorithm for $K = 5$, which is slightly worse than the Nabbit implementation. As n increases to 5000, all the algorithms improve in scalability, and the gap between Nabbit and the other algorithms narrows. As n increases even more to 15000, however, the speedup starts to level off, and eventually decrease. I conjecture that this slowdown is due to a lack of memory bandwidth and locality when computing the terms $E(i, j)$ and $F(i, j)$. In Equation (2.2), if the γ term is replaced with indices which are independent of k, then I observe a significant improvement in speedup on $n = 15000$.

The second experiment studied the sensitivity of the algorithms to block size by fixing n and varying B. Figure 2-8 shows the results for $n = 4000$. For small block sizes, the task-graph algorithm using Nabbit performs worse than the divide-and-conquer algorithm with $K = 5$. For example, for $P = 1$ and $B = 1$, both divide-and-conquer algorithms require about 156 seconds, as opposed to 196 seconds for the task-graph algorithm. This result is not surprising, since for small block sizes, each node does not do enough work to amortize Nabbit's overhead for each node. In addition, Nabbit also has significant space overhead for each node.

As B increases, however, at $P = 1$, the runtime using Nabbit approaches the runtime for divide-and-conquer with $K = 5$, and Nabbit begins to slightly outperform the other algorithms when $B \geq 16$. The wavefront algorithm at small B appears to have overhead which is even higher than the task-graph algorithm. In particular, for $P = 1$ and $B = 1$, the wavefront algorithm required about 241 seconds. Some of the wavefront algorithm's overhead is likely due to the cost of spawning computations on small blocks on each antidiagonal.

In summary, these experiments indicate that while Nabbit may suffer from high overheads when each node does little work, for this dynamic program, Nabbit generally exhibits relatively small overheads and is at least competitive with — and sometimes faster than — both divide-and-conquer and wavefront implementations for blocks of reasonable sizes.

[8] The machine contained two chip sockets with each socket containing a quad-core 3.16 GHz Intel Xeon X5460 processor. Each processor had 6 MB of cache shared among the four cores and a 1333 MHz FSB. The machine had a total of 8 GB RAM and ran a version of Debian 4.0, modified for MIT CSAIL, with Linux kernel version 2.6.18.8. All code was compiled using the Cilk++ compiler, which is based on GCC 4.2.4, with optimization flag -O2.

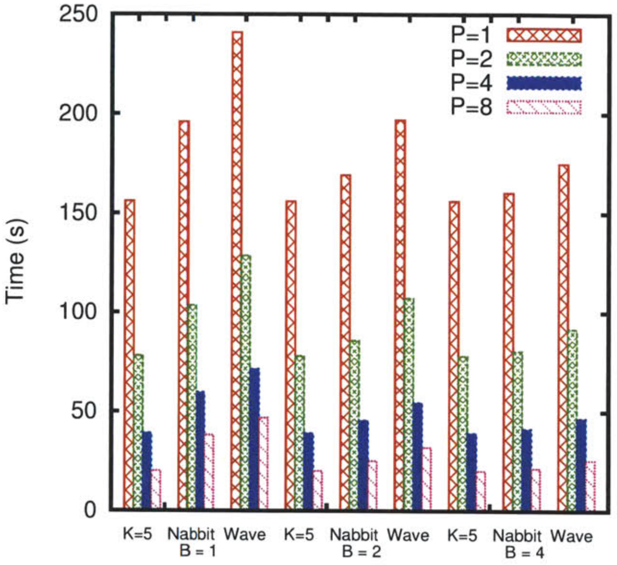


Figure 2-8: Running time for dynamic program benchmark with $n = 4000$, varying block size B for the base case.

2.4 Data-Block Interface for Static Nabbit

Static Nabbit can be used to support a convenient parallel-programming pattern — the data-block access pattern. In this pattern, a programmer decomposes a computation into tasks with "readsets" and "writesets" and then provides a sequential specification of tasks for the program. From this information, a platform can extract a task-graph computation and execute this task graph in parallel. For applications that can be decomposed into sufficiently coarse-grained tasks that operate on blocks of memory, the data-block access pattern can reduce the programming effort required to parallelize the application.[9]

In this section, we first explain how programmers can use the data-block access pattern to specify task-graph computations. We then describe the ***data-block interface***, an interface for Nabbit that supports this pattern. Finally, we compare Nabbit's interface with other platforms which can be used to implement the data-block access pattern — specifically SMPSs [18] and Intel Concurrent Collections [80].

The Data-Block Access Pattern

A programmer using the ***data-block access pattern*** codes a computation according to the following procedure.

1. Decompose a computation into a set of tasks $X_1, X_2, \ldots, X_{|V|}$, where each task operates on blocks of memory drawn from a universe $\mathcal{M}$ of all memory blocks.
2. Specify for each task X_i a ***readset*** $\mathtt{r}(X_i) \subseteq \mathcal{M}$, which is the set of blocks that X_i reads

[9] This section describes joint work with David Ferry and Kunal Agrawal [46].

from, and a ***writeset*** $\mathrm{w}(X_i) \subseteq \mathcal{M}$, which is the set of blocks that X_i writes to.

3. Provide any valid serial ordering for the execution of the tasks. Typically, this ordering comes from a sequential specification of the program.

A platform supporting the data-block access pattern then executes this computation in two stages:

1. Analyze the dependencies between tasks to construct a task graph.
2. Schedule and execute this task graph in parallel.

As a concrete example, consider the dynamic program from Equation (2.1), which operates on an $n \times n$ grid. For this computation, a programmer using the data-block access pattern might write the code in Figure 2-9. Using this pattern, instead of computing the values of $M(i, j)$ directly, the loops in lines 5–8 construct tasks to compute the values using the APPENDTASK method. A programmer uses the APPENDTASK method by specifying a function instance to execute as a task (e.g., MTASK(i, j)) as well as the readset and writeset of the task, as shown in line 8. Then, after specifying all tasks, the programmer can execute the task graph in parallel, as in line 9. The core of the data-block access pattern lies in a platform's implementation of the APPENDTASK method. When the programmer calls this method to add a new task node, the underlying platform (e.g., Nabbit) analyzes the dependencies in the computation and adds the appropriate edges into the task graph.

Figure 2-9 gives one sequential ordering of tasks that correctly computes the dynamic program, i.e., for any task (i, j), all the tasks that (i, j) depends on are generated before (i, j) itself is generated. In general, there may be many correct orderings of tasks that lead to a correct execution of the program. For example, the order of the loop nests in Figure 2-9 could be exchanged without changing the program's output. The programmer might choose to specify any one of these legal total orders.[10] The platform's runtime scheduler is free to reorder tasks and/or execute tasks in parallel as long as the dependencies implied by the readsets and writesets are satisfied. Thus, the platform can extract a parallel specification and execute it on multiple processors.

Of course, to achieve good performance using the data-block access pattern in practice, one needs to have coarse-grained tasks. In particular, tasks need to be sufficiently coarse-grained that two conditions are satisfied. First, the overhead of creating tasks should be small compared to the time to compute tasks. Second, the time to construct the task graph serially should be much smaller than the time to execute tasks in parallel.

Also, the data-block access pattern as described in this section does not allow the readset and writeset for each task X_i to depend on results computed from other tasks X_j. This restriction translates to the requirement that the entire task graph be constructed before any tasks execute. As discussed later in Section 2.6, it is possible to loosen this restriction in Nabbit, at the cost of potentially complicating the programming model.

The Data-Block Interface

For programs written using the data-block access pattern, one can construct a task graph automatically from the specification of tasks and their readsets and writesets. This section

[10] In principle, the programmer might even specify a partial order, but for simplicity, I do not consider the possibility of parallel graph construction.

```
// Initialize base cases.
$M(1,1) \leftarrow 0$
parallel for $k = 2$ to $n$
    $M(k,1) \leftarrow 0$
    $M(1,k) \leftarrow 0$

// Build task graph $\mathcal{D}$.
$\mathcal{D} \leftarrow \emptyset$
for $i = 2$ to $n$
    for $j = 2$ to $n$
        $\mathcal{D}$.APPENDTASK[Task = MTASK$(i,j)$,
                                  ReadSet = $\{M(i-1,j)\}$, WriteSet = $\{M(i,j)\}$]
$\mathcal{D}$.EXECUTE()    // Execute task graph.
```

Figure 2-9: Pseudocode that uses the data-block access pattern. In this code, MTASK is a task which computes $M(i,j)$ using the recurrence in Equation (2.1). To achieve good performance for this example in practice, one would also need to block the computation.

explains the theory behind this construction and then describes the ***data-block interface*** for Nabbit, an interface for supporting the data-block access pattern.

To describe the algorithm for task-graph creation, we require some definitions. For any task X_i and memory block $\ell \in M$, define the ***last-writer function*** as $\phi(X_i,\ell) = X_j$, where j is the largest index $j < i$ for which $\ell \in \mathtt{w}(X_j)$. Intuitively, $\phi(X_i,\ell)$ is the last task before X_i which writes to the specified block ℓ. For convenience, we assume there exists a source task X_0 with $\mathtt{w}(X_0) = \mathcal{M}$, i.e., the source task writes to all of memory.

The last-writer function allows us to define a data-dependence graph for a given sequence of tasks.

Definition 2.1. *Consider a sequence of tasks $X_0, X_1, \ldots X_{|V|}$. The* ***data-dependence task graph*** *is a graph where each task X_i has the following types of incoming edges:*

1. ***Dependency Edge:*** *For each memory block $\ell \in \mathtt{r}(X_i)$, add an edge from task $\phi(X_i,\ell)$ to task X_i.*

2. ***Antidependency Edge:*** *For each memory block $\ell \in \mathtt{w}(X_i)$, let $X_k = \phi(X_i,\ell)$. Then, for any task X_j which satisfies $\ell \in \mathtt{r}(X_j)$ and $k < j < i$, add an edge from X_j to X_i.*

We assume that the writeset is a subset of the readset ($\mathtt{w}(X_i) \subseteq \mathtt{r}(X_i)$). This assumption is not strictly necessary to produce a valid program, but it simplifies the formal statement of Definition 2.1 without significantly affecting the expressiveness of the programming model.

Intuitively, the data-dependence graph captures the data dependencies imposed by creating the tasks $X_1, X_2, \ldots, X_{|V|}$ in sequential order. Dependency edges arise when a new task X_i wants to access a memory block ℓ. The new task X_i must wait for the last task X_j

```
task* appendTask(void (*taskFunc)(void*),
                  void* params,
                  unsigned int paramSize,
                  void* writeAddresses,
                  void* readAddresses,
                  task* arbitraryDependency)
```

Figure 2-10: Signature for AppendTask method for Nabbit's data-block interface.

which previously wrote to ℓ. Similarly, an antidependency edge captures the constraint that a task X_i should not overwrite the value of ℓ until all the tasks X_j wishing to read from X_k (the last previous writer to ℓ) have done so. One can show (e.g., as in [77], Chapter 2) that any reordering of the sequence of tasks which does not contradict the edges in the data-dependence graph (i.e., any topological sort of the data-dependence graph) does not change the subsequence of tasks which read from or write to any memory block ℓ. Thus, any execution of tasks which obeys the edges of the data-dependence graph produces the same final state of memory.[11]

We support the data-block access pattern in Nabbit by creating a new ***data-block interface***. Using this interface, a programmer first creates an empty task graph by constructing a `taskGraph` object using a single constructor that accepts no arguments. This constructor creates a single task node which serves as the source node for the computation. All other task nodes are then added through the `taskGraph` member function AppendTask. Figure 2-10 shows the signature of the AppendTask method.

Programmers use the AppendTask method to create tasks X_i by specifying a function which represents the computational work for the task and providing a list of individual memory addresses[12] which represent the readset and writeset for each task. Parameters are passed to `taskFunc` through a pthreads-style `void*` struct pointer. The AppendTask method also returns a reference to the task node being created. Programmers can use this reference and Nabbit's existing AddDep method to insert additional arbitrary dependencies between task-graph nodes. The data-block interface for Nabbit assumes that each memory block ℓ must remain at a fixed memory address (because tasks X_i can access and modify a block ℓ through pointers), and assumes that data blocks cannot partially overlap.

Other Platforms for Task-Graph Execution

This section discusses two other platforms for parallel task-graph execution that can support the data-block access pattern, namely SMP superscalar and Intel Concurrent Collections. Static Nabbit provides a programming interface which is more constrained than these two

[11] The analysis described in Definition 2.1 can be considered as a simplified version of the dependence analysis used in modern optimizing compilers [77]. Unlike traditional compiler analysis, however, our analysis happens at runtime and aims to extract parallelism dynamically. Much in the same way that modern processors can extract instruction-level parallelism from an instruction stream, a platform supporting the data-block access pattern aims to extract parallelism from a sequential specification of tasks.

[12] This interface assumes that data blocks are disjoint, so each block may be uniquely identified by its starting address.

platforms, but which is arguably simpler to understand and schedule efficiently.

SMP superscalar (SMPSs) [18] is a platform that enables programmers to specify and execute task graphs using source-code annotations and compiler support. To create a task in SMPSs, programmers label a function with input and output annotations, specifying which data blocks the function reads from and writes to, respectively. Then, the SMPSs compiler translates any call of the function into a call into the SMPSs runtime, which adds a task for that function into a task graph. Thus, the user's original program provides the sequential specification for creating a task graph. As in the data-block interface for Nabbit, SMPSs infers the dependence edges in the task graph automatically, from an analysis of the data blocks accessed by each task.

SMPSs provides a more complex programming model than static Nabbit because it conceptually overlaps the creation of the task graph with execution of tasks. For example, consider a function `F` which normally calls two functions serially — `G1` and then `G2`. If an SMPSs programmer annotates `G1` and converts it into a task, but does not annotate `G2`, then in `F`, to ensure that the task `G1` completes before `G2` begins executing, the programmer needs to insert additional synchronization (using methods provided by SMPSs) between `G1` and `G2`. This mixture of task-graph creation and execution due to implicit task-graph creation can potentially improve performance if creating the graph is expensive, but it can also be more complicated for programmers to reason about.

In contrast, using static Nabbit with the data-block interface, the programmer must finish creating all task-graph nodes before beginning execution of the task graph. This explicit separation between phases has simpler semantics for the programmer to reason about. For computations where the time to construct a task graph is small compared to the time to execute the graph, having separate phases can also potentially be more efficient than overlapping task-graph creation and execution. When the creation and execution phases overlap, the runtime may need to incur additional synchronization overhead to add a task to the graph, e.g., to add a task Y to the graph which depends on task X while X is being executed. Section 2.6 describes a dynamic task-graph interface which extends Nabbit to support this kind of overlap, but at the cost of requiring additional runtime data structures.

Finally, SMPSs supports one additional extension to the data-block access pattern that Nabbit does not currently provide, which the authors of SMPSs refer to as ***parameter renaming***. Parameter renaming, which is analogous to register renaming in modern computer processors, is a technique for eliminating write-after-read and write-after-write dependencies on data blocks by allocating temporary storage for these data blocks. Parameter renaming can expose additional parallelism, but at the cost of using additional memory.

Similarly, Nabbit could also support ***block renaming***, i.e., copying blocks to eliminate antidependency edges, but only if the COMPUTE of every task only accesses data blocks as values which can be copied freely. Nabbit's interface assumes that each memory block ℓ must remain at a fixed memory address, which allows code to store pointers to data in blocks. If programmers do not store pointers, however, then Nabbit could potentially expose more parallelism through renaming. Supporting renaming also requires careful runtime scheduling to avoid excessive space usage.

Intel's Concurrent Collections language (CnC) [80] also enables programmers to create and execute dynamic task graphs. CnC programmers conceptually specify computations by creating graphs with three different kinds of nodes: computational steps, data items, and control tags. In terms of a task-graph execution, one can think of computational steps as task nodes that can potentially execute, and control tags as keys which the programmer uses to identify the tasks that a program should actually execute at runtime. Each computational step can both consume (i.e., depend on) a set of input data items and/or control tags, and produce a set of output data items and/or control tags.

CnC provides a more general programming model with richer semantics than Nabbit. For example, using static Nabbit, one cannot easily specify a conditional branch at the level of tasks, e.g., a programmer cannot easily specify a task graph where task X executes, and then either task Y or Z runs depending on the result of X. To execute this kind of computation using Nabbit, the programmer must create a single task node which represents the combined task of either Y or Z. In contrast, this kind of control flow for tasks is easier to specify using CnC, because CnC makes an explicit distinction between a step (a task that may or may not get executed in a program), and its control tag, which governs if/when steps can get executed.

Also, unlike Nabbit, the CnC programming model makes a semantic distinction between control tags (i.e., tasks) and data items that are consumed or produced by tasks. Although theoretically one can simulate a data item in Nabbit as a task node with an empty COMPUTE method, having a semantic distinction between a control dependency and a data dependency can potentially enable additional runtime optimizations. On the other hand, having two kinds of dependencies also adds complexity to both the programming model and the runtime scheduler.

2.5 Cholesky Factorization in Nabbit

This section discusses how one can use task graphs to implement a parallel asynchronous algorithm for Cholesky factorization. It also presents some empirical results for an implementation of this algorithm using Nabbit. The results demonstrate that task-graph execution using Nabbit can achieve performance and scalability for Cholesky factorization which is competitive with specialized systems designed for linear-algebra computations.[13]

Dense Cholesky Factorization

First, we review the problem of dense Cholesky factorization and describe parallel but synchronous algorithms for this problem.

For a symmetric positive-definite matrix M, the Cholesky factorization is defined to be a lower-triangular matrix A such that $M = AA^T$. LAPACK [15], the standard library interface for linear-algebra routines, computes the Cholesky factorization using *DPOTF2*. This routine accepts an input matrix M and computes the factorization in-place.

To achieve good performance, linear-algebra algorithms operate on blocked matrices. A blocked $n \times n$ matrix M is divided into blocks of size $B \times B$ with each block appearing contiguously in memory. Figure 2-11 shows a blocked Cholesky factorization algorithm, as described by Buttari et al. [34]. More specifically, Figure 2-11 describes a ***right-looking***

[13] This section describes joint work with David Ferry and Kunal Agrawal [46].

```
CHOLESKY-SEQUENTIAL
for $i = 1$ to $\lceil n/B \rceil$
    $DPOTF2[i]$                          // factor $M(i,i)$ in place
    for $j = i+1$ to $\lceil n/B \rceil$
        $DTRSM[j,i]$                     // $M(j,i) \leftarrow M(i,i)^{-T} \cdot M(j,i)$
    for $k = i+1$ to $\lceil n/B \rceil$
        for $j = k$ to $\lceil n/B \rceil$
            $DGEMM[i,j,k]$               // $M(j,k) \leftarrow M(j,k) - M(j,i) \cdot M(k,i)^T$
```

Figure 2-11: Sequential code for a blocked Cholesky factorization [34]. For clarity in description, we have parametrized the *DPOTF2*, *DTRSM*, and *DGEMM* calls in terms of i, j, and k. The original code from [34] does not actually use this parameterization.

```
CHOLESKY-SYNCHRONOUS
for $i = 1$ to $\lceil n/B \rceil$
    $DPOTF2[i]$                          // factor $M(i,i)$ in place
    parallel for $j = i+1$ to $\lceil n/B \rceil$
        $DTRSM[j,i]$                     // $M(j,i) \leftarrow M(i,i)^{-T} \cdot M(j,i)$
    parallel for $k = i+1$ to $\lceil n/B \rceil$
        parallel for $j = k$ to $\lceil n/B \rceil$
            $DGEMM[i,j,k]$               // $M(j,k) \leftarrow M(j,k) - M(j,i) \cdot M(k,i)^T$
```

Figure 2-12: A synchronous blocked Cholesky factorization using fork-join parallelism. This code is parallelizes the factorization in Figure 2-11 by executing the *DTRSM* and *DGEMM* tasks in parallel.

algorithm which operates on a blocked matrix with blocks of size $B \times B$. On each iteration, the algorithm first factors a main-diagonal block with *DPOTRF*, the LAPACK [15] routine for serial Cholesky factorization. It then updates the blocks below the diagonal block with *DTRSM*, the routine for triangular system solve. Finally, it completes the iteration by updating blocks below and to the right of the diagonal using *DGEMM*, the routine for matrix multiplication.

The naive approach to parallelizing Figure 2-11 uses fork-join parallelism to convert each inner **for** loop a **parallel for** loop, as shown in Figure 2-12. This algorithm computes a Cholesky factor (i,i) by itself, then computes each block (j,i) in the column underneath (i,i) in parallel, and then finally computes the trailing submatrix blocks below the diagonal and to the right of (i,i) in parallel. This algorithm is ***synchronous***, in the sense that execution of the iterations of the outermost loop cannot overlap. An implementation of this synchronous algorithm in Cilk was described in [85].

Although the algorithm in Figure 2-12 exposes some parallelism, it overspecifies dependencies in Cholesky factorization. For example, consider the case of Cholesky when

$\lceil n/B \rceil = 4$. In this case, Figure 2-12 generates tasks as shown in Figure 2-13(a). This algorithm enforces some extra dependencies, however, which are not required for correctness. For example, *DGEMM*[2,2] only needs to wait for *DTRSM*[2,1] to finish, but not for *DTRSM*[3,1]; in Figure 2-12 it must wait for both.

Alternatively, one can also implement a recursive divide-and-conquer algorithm for Cholesky factorization in a fork-join language such as Cilk. This algorithm divides an $n \times n$ matrix lower-triangular matrix into 3 submatrices of size $n/2 \times n/2$: an upper left triangular matrix L_{11}, a lower left square matrix L_{21}, and then the lower right triangular matrix L_{22}. Conceptually, this algorithm can be thought of as an execution of Figure 2-12 on a block matrix with $\lceil n/B \rceil = 2$, except the algorithm operates recursively. First, the algorithm recursively computes a Cholesky factorization on L_{11}, then a triangular solve on L_{21}, and finally an update on L_{22}. Although these three steps must execute sequentially, the algorithm is parallel because it utilizes parallel subroutines (e.g., recursive matrix multiplication). All recursive subroutines use standard BLAS routines in the base case — individual tiles of size $B \times B$. Like Figure 2-12, however, this approach still enforces additional dependencies that are unnecessary for correctness.

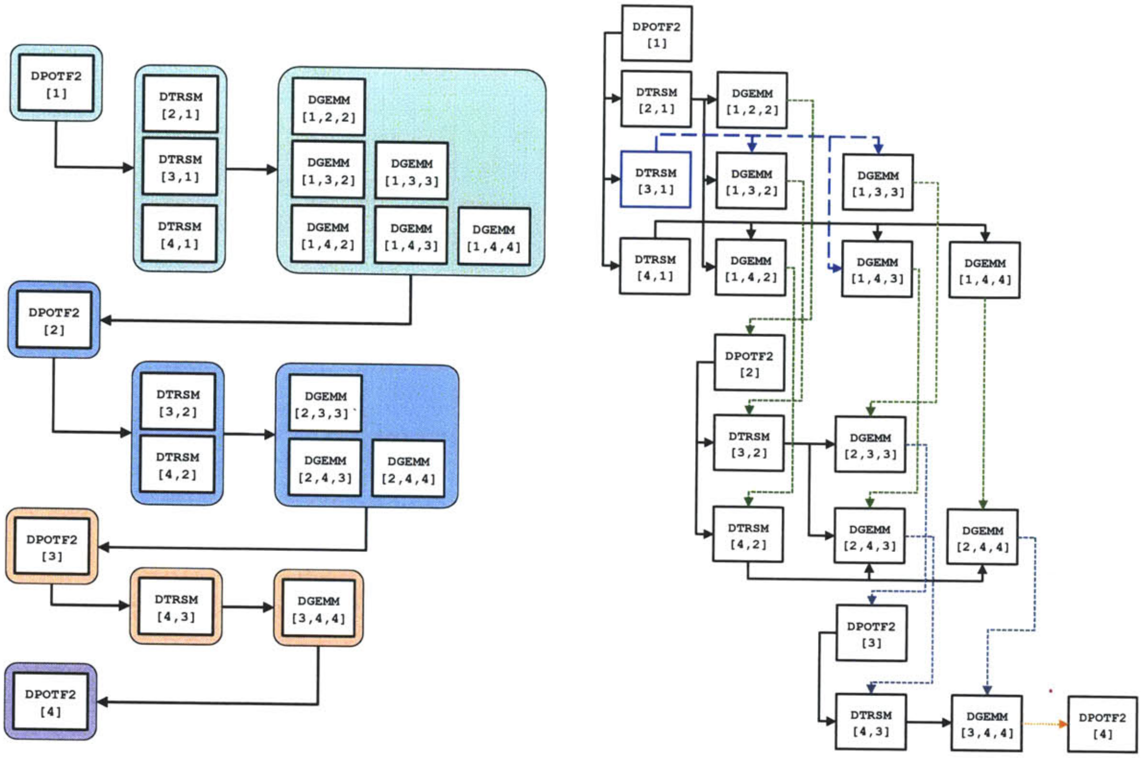
Asynchronous Algorithms for Cholesky

To exploit the additional parallelism in Cholesky factorization, researchers have proposed asynchronous algorithms, which construct a task graph with only the necessary dependencies (as shown in Figure 2-13(b)), and execute this task graph in parallel. Buttari et al. [34] implemented such an asynchronous algorithm for Cholesky using an unnamed DAG scheduler and achieved parallel speedup of roughly 6 on an 8-core machine. More recently, researchers in [36] implemented the same asynchronous Cholesky using Intel's Concurrent Collections (CnC) and achieved a speedup of 11 on a 16-core machine.

Unfortunately, implementing an asynchronous algorithm can generally be tricky because the resulting task graph may have a complex set of dependences. For example, if one begins with a serial algorithm such as Figure 2-11, it is not immediately obvious what the dependencies between tasks should be in the parallel algorithm. Some dependencies are clear from the original algorithm, while others are not. Within each iteration of the first inner loop of Figure 2-12, it is clear that each *DTRSM*$[j,i]$ task in iteration i is dependent upon the preceding *DPOTF2*$[i]$ task. It is less obvious from Figure 2-12, however, that each *DTRSM*$[j,i]$ task also depends upon the *DGEMM*$[i-1,j,i]$ task which previously wrote to the same block. Without the second dependency, it would be possible for the *DTRSM*$[k,i]$ task to race ahead of the preceding *DGEMM*$[i-1,j,i]$ task and execute out of order.

It turns out that the rules shown in Figure 2-14 describe a correct implementation of Cholesky factorization. Given these rules, it is relatively straightforward to translate these rules and implement an asynchronous Cholesky factorization using Intel Concurrent Collections (CnC). Unfortunately, if a programmer starts with the sequential code in Figure 2-11, careful reasoning is required to derive a correct set of rules for constructing the task graph.

Using Nabbit's data-block interface, however, one can more easily convert Figure 2-11 into a parallel task graph. The resulting pseudocode for this implementation is shown in Figure 2-15. The structure of this code is identical to Figure 2-11, except that calls to the



(a) Synchronous Cholesky.

(b) Asynchronous Cholesky.

Figure 2-13: Tasks generated for Cholesky factorization when $\lceil n/B \rceil = 4$. The graph in 2-13(a) shows a synchronous algorithm (Figure 2-12), while the graph in 2-13(b) shows the dependencies that are required for the correctness of Cholesky factorization.

Task	Input Data	Output Data	Tasks Produced
$DPOTF2[i]$	$M[i-1,i,i]$	$M[i,i,i]$	$DTRSM[j,i]$ for all $i < j \leq \lceil n/B \rceil$
$DTRSM[j,i]$	$M[i,i,i]$ and $M[i-1,j,i]$	$M[i,j,i]$	$DGEMM[i,j,k]$ for all $i < k \leq j$
$DGEMM[i,j,k]$	$M[i,j,i]$, $M[i,k,i]$, and $M[i-1,j,k]$	$M[i,j,k]$	$DPOTF2[i+1]$ **if** $j = k = (i+1)$

Figure 2-14: Asynchronous Cholesky factorization algorithm using CnC. $M[i,j,k]$ represents block (j,k) in the matrix after iteration i. In this CnC implementation, each task has its own control tag. Each task performs a `get` for each input data item, and a `put` to generate output data items and control tags. This implementation constructs the task graph dynamically, i.e., it tries to avoid executing a `put` for a task X until at least one of the dependences of X has been satisfied.

```
CHOLESKY-DATABLOCK
$\mathcal{D} \leftarrow \emptyset$
for $i = 1$ to $\lceil n/B \rceil$
    $\mathcal{D}$.APPENDTASK[Task $= DPOTF2[i]$,
                        ReadSet $= \{M(j,j)\}$, WriteSet $= \{M(j,j)\}$]
    for $j = i+1$ to $\lceil n/B \rceil$
        $\mathcal{D}$.APPENDTASK[Task $= DTRSM[j,i]$,
                            ReadSet $= \{M(i,i), M(j,i)\}$, WriteSet $= \{M(j,i)\}$]
    for $k = i+1$ to $\lceil n/B \rceil$
        for $j = k$ to $\lceil n/B \rceil$
            $\mathcal{D}$.APPENDTASK[Task $= DGEMM[i,j,k]$,
                                ReadSet $= \{M(j,i), M(k,i)\}$,
                                WriteSet $= \{M(j,k)\}$]
$\mathcal{D}$.EXECUTE()
```

Figure 2-15: Asynchronous Cholesky factorization using Nabbit's data-block interface.

LAPACK routines are replaced with methods to create a task-graph node with a specified readset and writeset.

Empirical Results

This section presents empirical results for the Nabbit implementation of dense Cholesky factorization. The Nabbit implementation achieved a speedup of about 30 on 48 cores. With a large number of cores, the application appears to be constrained by memory bandwidth. These results show that the Nabbit implementation of Cholesky factorization is competitive with other implementations, including PLASMA [11], a library specifically designed for asynchronous linear-algebra computations.

All experiments were conducted on a four-socket Opteron 6100 SMP server. Each socket contained an AMD Opteron 6168 processor, which had 12 cores clocked at 1.9 GHz. The server contained 128 GB of DDR3 RAM and ran a CentOS distribution 5.5 with kernel version 2.6.18-194.8.1.el5. In our benchmarks, Nabbit and Cilk++ timings were collected using the Cilk++ Cilkview tool, while SMPSs, PLASMA, and CnC timings were collected using GOMP and `OMP_GET_WTIME`.

The following implementations of a Cholesky factorization (*DPOTRF*) were evaluated:

1. **Nabbit**: an implementation of Cholesky factorization using the data-block access pattern, as described in Figure 2-15. This factorization operates on matrices stored in a tiled layout. Each tile was a block matrix of size $B \times B$ stored in column-major layout, with the tiles themselves also arranged in a column-major layout in the overall matrix.

2. **PLASMA**: a library designed specifically for asynchronous linear-algebra computations which implements its own optimized runtime scheduler. In PLASMA, the

method that accepts tiled matrices as input was used.

3. **CnC**: a factorization for tiled matrices, implemented using the algorithm described in Figure 2-14.

4. **SMPSs**: a factorization for tiled matrices. This default factorization from the SMPSs distribution was used, except modified to operate on double-precision instead of single-precision floating-point values. This code was compiled and run using the default SMPSs configuration, except for increasing the soft limit on tasks in the task graph to 1500 and the hard limit to 2000.

5. **Cilk++**: a recursive divide-and-conquer implementation on a tiled matrix, with an $n \times n$ matrix being divided into 4 submatrices, each of size $n/2 \times n/2$.

6. **ACML**: the *DPOTRF* method from ACML, AMD's math library. This version is parallelized using OpenMP and operates on matrices stored in a column-major layout.

All implementations except the last one used the sequential BLAS implementation provided by ACML for the base cases. The ACML version utilizes its own parallel BLAS implementation.

Figure 2-16 suggests an approximate ranking of performance of the various versions: the PLASMA implementation generally achieves the best performance, followed by CnC and Nabbit, then SMPSs, then the recursive Cilk++ implementation, and finally the ACML implementation.

Good performance from PLASMA is expected, since PLASMA is a library with a scheduler designed and optimized for asynchronous linear algebra computations. The CnC, Nabbit, and SMPSs versions are all asynchronous algorithms based on task-graph execution, and thus achieve similar performance.

The performance of SMPSs appears to degrade when too many processors are used. For example, for the smallest matrix size ($n = 10000$), the performance of SMPSs begins to degrade after about $P = 30$ processors. By turning on SMPSs's tracing functionality and profiling the Cholesky factorization, it was observed that many processors were idle, waiting to find work to do. This observation suggests that the scheduler for the Cholesky factorization, SMPSs may not be as efficient as Nabbit or CnC at scheduling tasks on the critical path of the computation, particularly when the matrix size is smaller and parallelism is more limited.

The Cilk++ divide-and-conquer implementation for the smaller matrix sizes does not scale as well. For $n = 20000$, the Cilk++ implementation achieves a speedup of at most 10. This result is not surprising, since the divide-and-conquer algorithm corresponds to a series-parallel task-graph execution, which enforces some unnecessary dependency edges and limits the parallelism of the overall algorithm. Cilkview reports a parallelism of slightly less than 42 for $n = 20000$, which highlights this limitation more concretely.

Finally, the ACML code has the worst performance, achieving almost no speedups or even slowdown after about $P = 12$ processors (which requires using more than one socket). This performance difference likely arises because all the other implementations operate on

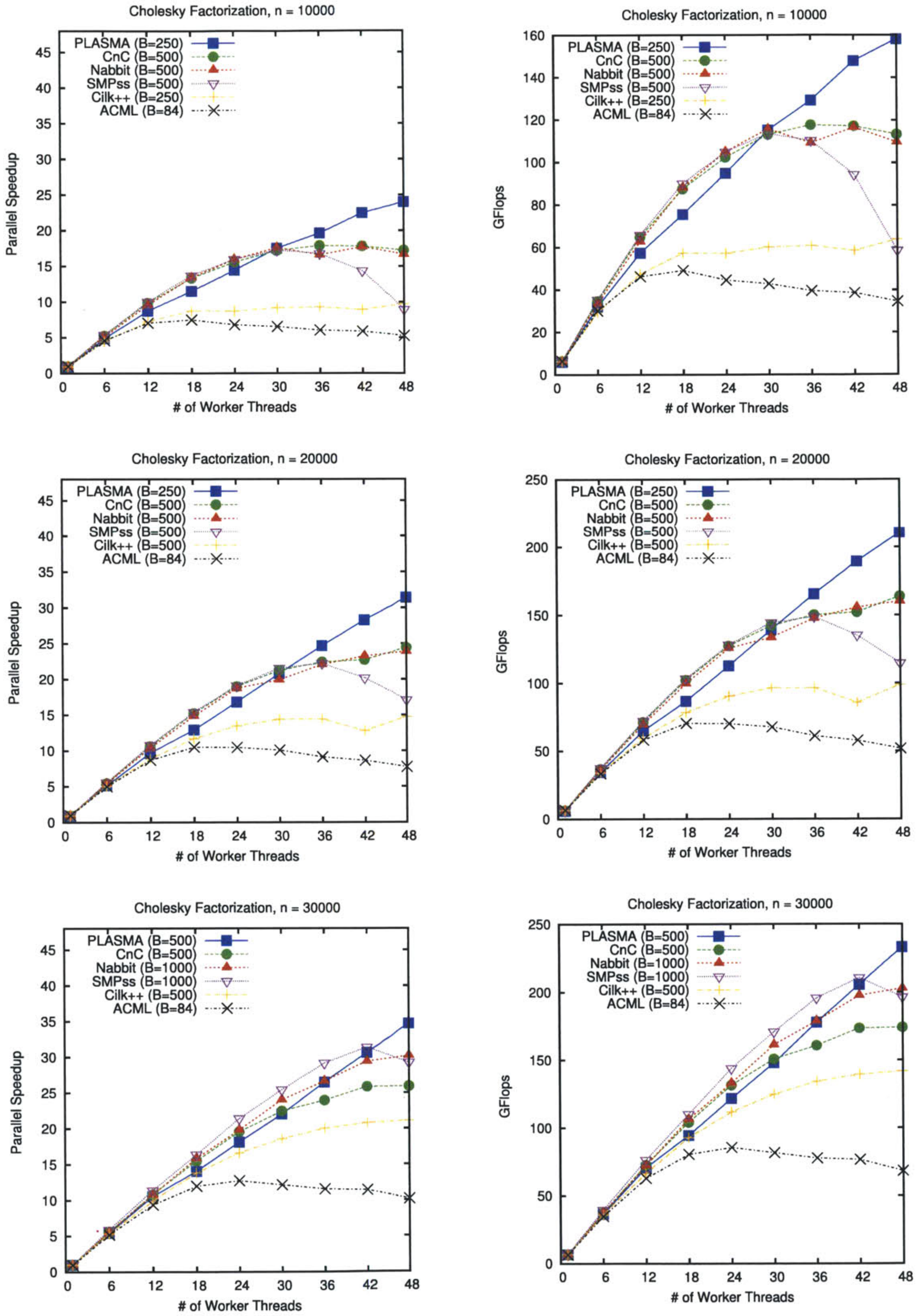


Figure 2-16: Cholesky factorization of $n \times n$ matrices. For each implementation, we chose the block size B that achieved the highest absolute performance. All times are normalized against the time for the fastest serial implementation.

matrices stored in tiled layouts, whereas the ACML code accepts an input matrix stored in a column-major layout.

Ideally, one would like to have perfect linear speedup out to all 48 cores. The lack of perfect linear speedup is likely due to the machine's limited memory-bandwidth however, rather than a lack of parallelism in the algorithm. To assess memory-bandwidth in a heuristic way, multiple small independent Cholesky computations were run in parallel (such as four computations using twelve cores each, or eight computations using six cores each). Individually these small computations exhibited near-linear speedup, but when run together, their performance suffered. For example, when running 4 instances of Cholesky factorization on 12 cores each, each instance had an average speedup of about 7.

In summary, these empirical results demonstrate that task-graph execution using Nabbit's data-block interface is effective on a practical benchmark, namely Cholesky factorization. The Nabbit implementation achieved performance which was close to that of PLASMA, a library specifically optimized to execute linear-algebra computations. In fact, Nabbit often outperformed PLASMA when using about 30 or fewer cores. Nabbit is also competitive with SMPSs and CnC, other platforms that provide more complex programming interfaces for task-graph execution.

2.6 Nabbit for Dynamic Task Graphs

This section presents dynamic Nabbit, an extension of static Nabbit that supports the execution of ***dynamic task graphs*** — task graphs whose nodes and edges are created on the fly at runtime. In particular, this section describes the interface and theoretical guarantees provided by dynamic Nabbit. Compared to static Nabbit, dynamic Nabbit provides users a more complex but flexible programming interface for task-graph execution.

Interface

Dynamic Nabbit provides an interface to programmers for executing a task graph $\mathcal{D}$ whose nodes and edges are created on the fly at runtime. As in static Nabbit, for each node A in $\mathcal{D}$, programmers must specify a COMPUTE method, which performs A's computation only after all of A's predecessors in $\mathcal{D}$ have been computed. Unlike static Nabbit, however, dynamic Nabbit first "discovers" the nodes on which a node A depends before executing the computation of A. This interface reflects the notion that in some task-graph applications, a node A knows which nodes it requires values from, but A may not be aware of all nodes that may use its value.

In dynamic Nabbit, the programmer refers to nodes using keys that can be hashed, rather than to nodes directly, which allows the space of possible nodes to be much larger than those that are actually created. A node with key k discovers its immediate predecessors by executing a programmer-specified INIT method. In the INIT method, the programmer specifies the other keys k' on which k depends using the (library-provided) function ADDDEP(k'). Although multiple keys may depend on the same key k, Nabbit creates only one node object for each key, thus guaranteeing that INIT and COMPUTE execute exactly once per key.

Nabbit implementation

Dynamic task graphs are more complicated to support than static task graphs because a new node B that is a successor of A can be created at any time with respect to A's creation. Specifically, B can be created (1) before A has been created, (2) after A has been created but before A has completed its computation and notified its successors, or (3) after A has completed its notification. Thus, dynamic Nabbit requires additional bookkeeping.

Dynamic Nabbit maintains the following fields for each task-graph node A:

- **Key**: A unique 64-bit integer identifier for A.
- **Predecessor-key array**: The keys of A's immediate predecessors.
- **Status**: A field which changes monotonically from UNVISITED to VISITED, then to COMPUTED, and finally to COMPLETED.
- **Notification array**: An array of A's successor nodes that need to be notified when A completes.
- **Join counter**: A counter for A that reaches 0 when the COMPUTE for A is ready to be executed.

To compare dynamic Nabbit with the implementation of static Nabbit described in Section 2.1, the predecessor-key array replaces the predecessor array in static Nabbit, and the notification array replaces the successor array. In dynamic Nabbit, the notification array for A need not contain all of A's successors, since some successors may be created after A finishes its notifications.

Since dynamic Nabbit works with keys instead of pointers to node objects, Nabbit maintains a hash table for task-graph nodes and guarantees that a node with a particular key is never created more than once. Nabbit uses a hash-table implementation that supports two functions: INSERTTASKIFABSENT(k) and GETTASK(k). The first atomically adds a new node object for a specified key k to the hash table if none exists, and the second looks up a node for a key k.[14] The atomic insertion of a node into the hash table also changes the node's status from UNVISITED to VISITED.

Dynamic Nabbit generally tries to execute a task graph in a depth-first fashion. Execution begins with a call to STARTEXECUTION(f) (shown in Figure 2-17), where f is the key of $\mathcal{D}$'s sink node $t_{\mathcal{D}}$. Nabbit assumes that only a single call to STARTEXECUTION is active at any time. As its first step, Nabbit attempts to create a new node A for key f and atomically insert A into its hash table. Then, if this insertion is successful, Nabbit calls INITANDCOMPUTE(A).[15] The INITANDCOMPUTE method (shown in Figure 2-18) first creates A by calling INIT(A) and then recursively creates any dependencies (i.e., predecessors) of A. When this recursion reaches any node B with no dependencies, Nabbit calls COMPUTEANDNOTIFY(B) to compute B and any successors of B that are subsequently enabled. When Nabbit uses multiple processors, these methods still attempt to execute in a depth-first fashion if possible, but the execution is not strictly depth-first because of Cilk++'s work-stealing strategy.

[14] Nabbit could be easily modified to use any user-provided hash table that supports these two functions. This functionality would allow programmers to optimize by using an application-specific hash table.

[15] If this insertion fails, then a task with key f has already been created and/or computed, and thus the method does nothing.

```
StartExecution(f)
inserted = InsertTaskIfAbsent(f)
A = GetTask(f)
if inserted
    InitAndCompute(A)
```

Figure 2-17: Subroutine for dynamic Nabbit for starting execution of a task graph at a node with key f.

```
TryInitCompute(A,pkey)
inserted = InsertTaskIfAbsent(pkey)
B = GetTask(pkey)
if inserted
    spawn InitAndCompute(B)
finished = TRUE
lock(B)
if B.status < COMPUTED
    add A to B.notifyArray
    finished = FALSE
unlock(B)
if finished
    val = AtomDecAndFetch(A.join)
    if val = 0
        ComputeAndNotify(A)
sync
```

```
InitAndCompute(A)
assert(A.status == VISITED)
assert(A.join ≥ 1)
Init(A)
for pkey ∈ A.predecessors
    spawn TryInitCompute(A,pkey)
val = AtomDecAndFetch(A.join)
if val == 0
    ComputeAndNotify(A)
sync
```

```
DecComputeNotify(X)
val = AtomDecAndFetch(X.join)
if val == 0
    ComputeAndNotify(X)
```

```
ComputeAndNotify(A)
Compute(A)
A.status = COMPUTED
n = SizeOf(A.notifyArray)
A.notified = 0
while A.notified < n
    for i ∈ [A.notified,n)
        X = A.notifyArray[i]
        spawn DecComputeNotify(X)
    A.notified = n
    lock(A)
    n = SizeOf(A.notifyArray)
    if A.notified == n
        A.status = COMPLETED
    unlock(A)
sync
```

Figure 2-18: Pseudocode for executing dynamic task graphs using Nabbit. For a node A, the TryInitCompute(A) method attempts to create a predecessor (i.e., dependency) of A with the key *pkey*. InitAndCompute(A) spawns calls to try to create all of A's predecessors. Eventually, this method or one its spawned calls triggers ComputeAndNotify(A), which executes A and all successors of A enabled by the completion of A.

Synchronization in Nabbit

For static task graphs, synchronization occurs primarily through changes to join counters. The dynamic protocol is slightly more complicated, however, because the number of other nodes on which A depends is unknown before INIT is executed. Instead, A's join counter is atomically incremented when the user calls ADDDEP(k) inside INIT(A). For every node A, the join counter for A is initialized to 1 in order to prevent the join counter from reaching 0 before all the dependencies of node A have been created. Finally, the join counter for A is atomically decremented after INIT(A) has completed.

During a task-graph execution, dynamic Nabbit decrements the join counter for a node A once for every edge from some node Y to node A. If Nabbit tries to traverse an edge (Y,A) after Y has been COMPUTED, then A's join counter is decremented in line 12 of TRYINITCOMPUTE. If Nabbit tries to traverse an edge (Y,A) before Y has been COMPUTED, then A is added to $Y.\mathit{notifyArray}$, the list of nodes that Y notifies upon its completion. Eventually, A's join counter is decremented in line 1 of COMPUTEANDNOTIFY(Y). To avoid race conditions, dynamic Nabbit performs the additions of nodes to the notification array of a node A while holding A's lock. Similarly, it also changes the status of a node A to COMPLETED (as shown in line 13 of the COMPUTEANDNOTIFY method) while holding A's lock.

Discussion of Theory

One can prove a completion-time bound analogous to Theorem 2.3 for dynamic task graphs executed using Nabbit. The proof requires some additional definitions. For a task graph $\mathcal{D} = (V_{\mathcal{D}}, E_{\mathcal{D}})$ and any node $A \in V_{\mathcal{D}}$, consider the set

$$\texttt{loops}(A) = \bigcup_{X \in V_{\mathcal{D}}} \texttt{paths}(X,A) \times \texttt{paths}(X,A) .$$

Intuitively, $\texttt{loops}(A)$ is the set of all pairs of paths $(\mathrm{p}_1, \mathrm{p}_2)$ in $\mathcal{D}$ from any node X to A. Conceptually, $(\mathrm{p}_1, \mathrm{p}_2)$ represents a loop that walks from A to X along edges in p_1 backwards, and then walks back from X to A along forward edges in p_2. For a given computation DAG $\mathcal{G}$ and a node A, let $\mathit{init}^{\mathcal{G}}(A)$ be the subgraph corresponding to INIT(A), and let $\mathit{IC}^{\mathcal{G}}(A)$ be the subgraph corresponding to INITANDCOMPUTE(A). As before, let $\mathit{IC}^*(A)$ be the computation DAG representing an execution where all potential recursive calls occur, i.e., where line 7 of INITANDCOMPUTE is omitted.

For a dynamic task graph $\mathcal{D}$, the values of T_1 and T_∞ are greater than those for a static version of $\mathcal{D}$, since any execution must traverse $\mathcal{D}$ backwards from $t_{\mathcal{D}}$ to discover the dependencies of each node. More precisely, we have

$$T_1 = \sum_{A \in V_{\mathcal{D}}} \left(T_1(\mathit{init}(A)) + T_1(\mathit{com}(A))\right) + O(|E_{\mathcal{D}}|) ,$$

$$T_\infty = \max_{(\mathrm{p}_1,\mathrm{p}_2) \in \texttt{loops}(t_{\mathcal{D}})} \left\{ \sum_{X_1 \in \mathrm{p}_1} T_\infty(\mathit{init}(X_1)) + \sum_{X_2 \in \mathrm{p}_2} T_\infty(\mathit{com}(X_2)) \right\} + O(V_\infty) .$$

Theorem 2.5 states the completion-time bound for dynamic task graphs, which matches the bound in Theorem 2.3 except for an $O(V_\infty\Delta)$ term instead of $O(V_\infty \lg\Delta)$. This difference arises since the successors of a node A might be created and added sequentially to $A.notifyArray$, and thus may be notified one by one, instead of in parallel as in static Nabbit.

Theorem 2.5. *Let $\mathcal{D} = (V_{\mathcal{D}}, E_{\mathcal{D}})$ be a dynamic task graph with maximum degree Δ. With probability at least $1-\varepsilon$, Nabbit executes $\mathcal{D}$ in time*

$$O\left(\frac{T_1}{P} + T_\infty + \lg\left(\frac{P}{\varepsilon}\right) + V_\infty\Delta + C(\mathcal{D})\right)$$

where $C(\mathcal{D}) = O\left(\left(|E_{\mathcal{D}}|/P + V_\infty\right)\min\{\Delta, P\}\right)$.

Proof sketch. Nabbit's execution of $\mathcal{D}$ is modeled by the computation DAG $IC(t_{\mathcal{D}})$. As in Theorem 2.3, we bound the completion time by calculating $T_1(IC(t_{\mathcal{D}}))$ and $T_\infty(IC(t_{\mathcal{D}}))$ and then applying the analysis for Cilk.

We have $T_1(IC(t_{\mathcal{D}})) = T_1 + O(|E_{\mathcal{D}}| \cdot \min\{\Delta, P\})$, since INIT and COMPUTE for each node A happens exactly once, and $O(1)$ synchronization operations happen for every edge $(A,B) \in E_{\mathcal{D}}$, with each operation waiting at most $O(\min\{\Delta, P\})$ time due to contention.

We now argue the span $T_\infty(IC(t_{\mathcal{D}}))$ is bounded by $T_\infty(IC^*(t_{\mathcal{D}}))$, and then we show that $T_\infty(IC^*(t_{\mathcal{D}})) = T_\infty + O(V_\infty\Delta)$. From Figures 2-19 and 2-20, we can see that any path through $IC^*(t_{\mathcal{D}})$ travels along a single loop $(\mathsf{p}_1, \mathsf{p}_2) \in \mathtt{loops}(t_{\mathcal{D}})$, which is to say that it walks backward along p_1 calling INIT and then forward along p_2 calling COMPUTE. Thus, INIT and COMPUTE contribute at most T_∞ to the span. For every edge (A,B) along this loop, the added overhead due to bookkeeping and contention on synchronization is $O(\Delta)$: in the worst case, Δ iterations of the loop in line 5 of COMPUTEANDNOTIFY occur, each notifying one successor of A. Since each loop contains $O(V_\infty)$ edges, the total overhead along the span is at most $O(V_\infty\Delta)$. □

Strongly Dynamic Task Graphs

Although dynamic Nabbit discovers the nodes and edges of a task graph at runtime, the interface described thus far does not allow for the creation of a new task node based on the result of the COMPUTE of an existing task node. One can extend Nabbit to handle this more general class of ***strongly dynamic*** task graphs — task graphs for which the COMPUTE for a node A can trigger the creation of new task nodes.

To support strongly dynamic task graphs, the Nabbit interface allows users to make multiple calls to a CREATEKEY method inside the COMPUTE of any task node B. Each call to CREATEKEY takes as in input some key f_i as input. After Nabbit executes COMPUTE(B), but before computing any successors of B, it creates new task nodes for each key f_i, and then starts execution of a new task graph $\mathcal{D}_i$ with sink node having key f_i in parallel with the original task graph execution. Said differently, Nabbit behaves as though a new call to STARTEXECUTION for key f_i was spawned.[16] The Nabbit implementation correctly sup-

[16] Intuitively, in dynamic Nabbit, a key f is analogous to a tag in CnC, and a call to STARTEXECUTION for a key f is roughly equivalent to a `put` operation.

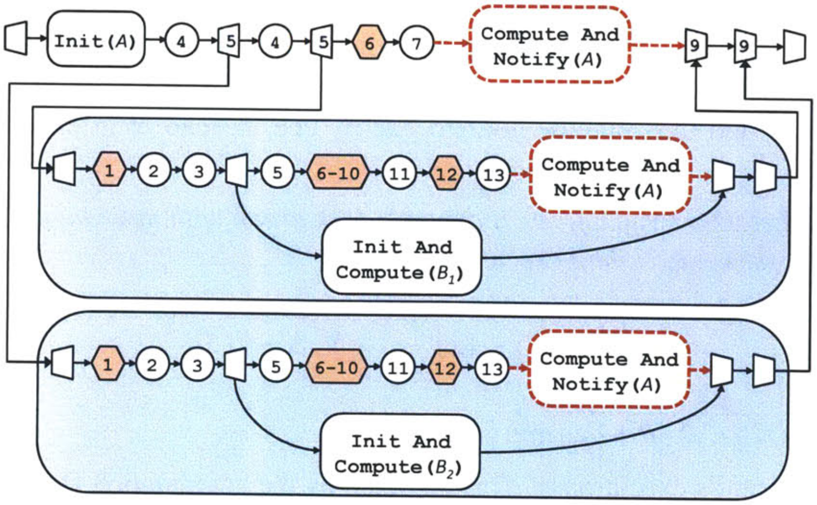


Figure 2-19: Example computation DAG generated by InitAndCompute(A). Nodes are labeled with line numbers from Figure 2-18. Hexagons correspond to synchronization operations that may experience contention. In InitAndCompute(A), two calls to TryInitCompute for A are shown, for two predecessors B_1 and B_2. Potential calls to ComputeAndNotify have a dashed border. In any execution of InitAndCompute(A), exactly one call to ComputeAndNotify(A) occurs.

ports concurrent calls to StartExecution, even when the task graphs $\mathcal{D}_i$ overlap (i.e., share nodes). In this case, Nabbit guarantees only that all $\mathcal{D}_i$ are computed after the last call to StartExecution finishes and the system reaches a quiescent state.

The creation of new task nodes complicates the theoretical analysis of runtime, however, because different task graphs may begin executing at different times. Theorem 2.5 does not hold for strongly dynamic task graphs in part because it does not handle the dependencies and interactions between task graphs $\mathcal{D}_i$ that overlap. An interesting direction for future work is to extend the theory to handle strongly dynamic task graphs.

2.7 Random Task-Graph Benchmark

This section investigates the overheads associated with the static and dynamic versions of Nabbit using a random task-graph microbenchmark. The empirical results presented in this section indicate that although Nabbit exhibits significant overhead on dynamic task graphs, this overhead can be amortized when each node does enough work. The results also demonstrate that Nabbit can successfully exploit both the DAG-level parallelism and the parallelism within each task.

To measure overhead in Nabbit, I constructed a microbenchmark that executes randomly constructed task graphs. This benchmark generates a random task graph $\mathcal{D}$ based on three parameters: Δ_i, the maximum in-degree of any node; U, the size of the universe from which keys are chosen; and W, the work in the Compute of each node. The task

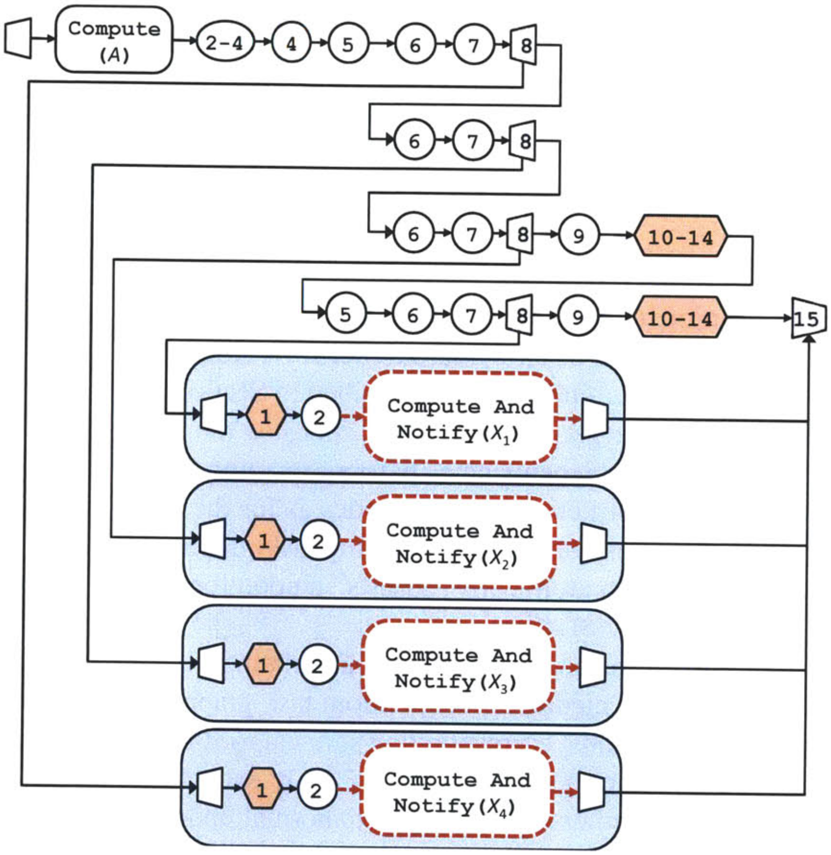

Compute (A)
2-4
4
5
6
7
8
6
7
8
6
7
8
9
10-14
5
6
7
8
9
10-14
15
1
2
Compute And Notify(X_1)
1
2
Compute And Notify(X_2)
1
2
Compute And Notify(X_3)
1
2
Compute And Notify(X_4)

Figure 2-20: An example computation DAG generated by ComputeAndNotify(A). Nodes are labeled with line numbers from Figure 2-18. Shaded hexagons correspond to synchronization operations that may experience contention. Potential calls to ComputeAndNotify are represented by dashed hexagons.

graph $\mathcal{D}$ has a single sink node A_0 with key 0. The graph is constructed by iterating over k from 0 to $U-1$ and repeating the following process:

- If $\mathcal{D}$ has a node A_k, choose an integer d_k uniformly at random from the closed interval $[1,\Delta_i]$.
- Create a multiset S_k of d_k integers, with each element chosen uniformly at random from $[k+1,U]$.
- Remove any duplicates from S_k, and for all $k' \in S_k$, add an edge $(A_{k'},A_k)$ to the task graph (creating $A_{k'}$ if it does not already exist).

In $\mathcal{D}$, each task-graph node A_k performs W work to compute $k^W \bmod p$ using repeated multiplication, where p is a fixed 32-bit prime number. The benchmark provides the option of either performing this work serially, or in parallel (dividing the work in half, spawning each half, and recursing down to a base case of $W = 25$).

Experiments

The random task-graph benchmark was used in three experiments: (1) to measure the overhead of parallel execution, (2) to compare the overheads of the static and dynamic Nabbit, and (3) to evaluate the benefits of allowing parallelism inside the computes of nodes. All experiments use the same machine setup as described in Section 2.3.

For static Nabbit, the task-graph benchmarks allocated the memory for nodes and created nodes with pointers to its dependencies before executing the task graph. For dynamic Nabbit, the benchmarks constructed the same nodes as for the static benchmark, and then inserted these nodes into a hash table before starting task-graph execution. The implementation of dynamic Nabbit then atomically "inserts" a node for a key by looking it up in a hash table and marking it as VISITED.

To measure the approximate overhead for manipulating node objects and for parallel bookkeeping, I constructed a medium-sized random task graph and varied W. I compared static and dynamic Nabbit against corresponding serial algorithms. These serial algorithms perform the same computation as Nabbit with $P = 1$, except that all lock acquires are removed and all atomic decrements are changed to normal updates.

Figure 2-21 shows that when $W = 1$ (each node does small work), the overhead of bookkeeping for static Nabbit is about 20% more than the serial version of the same algorithm. For dynamic Nabbit, the slowdown is about 16% over the serial algorithm. This baseline overhead indicates that one would not want to use Nabbit for task graphs where each node does little work, since the overheads of bookkeeping dominate. As each node does more and more work and W increases to 1000, however, the difference becomes less than 5%.

This data also suggests that the implementation of dynamic Nabbit exhibits a factor of 5 overhead over static Nabbit when $W = 1$. This difference is not surprising, since dynamic Nabbit ends up traversing a DAG twice — from the final node to the root and then back — while static Nabbit only traverses the DAG from root to final node. Also, in our benchmark, dynamic Nabbit performs additional look-ups in a hash table that the static version avoids. It was observed that each node generally requires W to be on the order of at least $1,000$ to $10,000$ before the dynamic Nabbit attains performance comparable to the static version.

	Static		*Dynamic*	
W	*Nabbit*	*Serial*	*Nabbit*	*Serial*
1	0.010	0.008	0.051	0.044
10	0.010	0.009	0.052	0.046
100	0.022	0.022	0.064	0.057
1000	0.137	0.134	0.178	0.177
10,000	1.267	1.265	1.306	1.301

Figure 2-21: Serial overhead of Nabbit on a random task graph. Time in seconds for serial execution of $\mathcal{D}$ with $|V_{\mathcal{D}}| = 14259$, $|E_{\mathcal{D}}| = 78434$, and $V_\infty = 99$ nodes. The task graph $\mathcal{D}$ was randomly generated with $\Delta_i = 10$, $U = 100{,}000$, and $W = 1$.

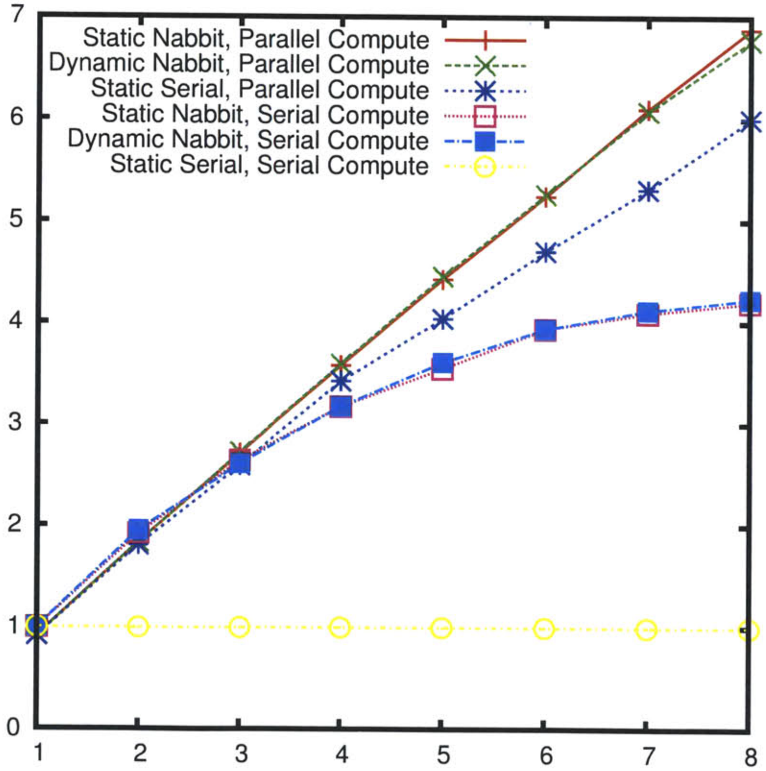


Figure 2-22: Comparison of static and dynamic Nabbit with and without parallelism in the COMPUTE function. For the random DAG, $|V_{\mathcal{D}}| = 127$, $|E_{\mathcal{D}}| = 614$, $V_\infty = 29$, and $W = 10^6$. Speedup is normalized over the time (1.12 s) for the static serial execution.

The next set of experiments compares the speedups of static and dynamic Nabbit. The first experiment created a large random task graph with small work ($W = 1$) per node. Even in the case when each node has small work and Nabbit has large overheads, Nabbit provided a speedup of up to 4.5 on 8 processors (graph not shown). Dynamic Nabbit scaled and achieved a speedup of about 3.7 on 6 processors over the serial dynamic Nabbit execution. Compared to the static versions, however, dynamic Nabbit was overwhelmed by overheads.

On the other hand, the second experiment demonstrated that when each node has a large amount of work to do, the performances of static and dynamic Nabbit are nearly identical, as shown in Figure 2-22. In this case, the task graph contains relatively few nodes (only 127). Examining the version where each node is computed serially, the theoretical parallelism was only about $127/29 = 4.4$. The static and dynamic versions of Nabbit both exploited most of this parallelism, providing a speedup of up to 4.2.

More importantly, however, Figure 2-22 demonstrates that to attain the best performance on this graph, one needs to exploit parallelism both in the task graph and within the COMPUTE functions. When only the DAG-level parallelism is exploited, Nabbit achieves a speedup of 4.2. On the other hand, when Nabbit is not used and nodes are visited serially — only exploiting parallelism within the compute function — the speedup is about 6. The best case occurs by exploiting the parallelism both between nodes and within nodes, in which case both static and dynamic versions of Nabbit provide a speedup of 7.

In summary, these experiments on random DAGs indicate that although Nabbit exhibits significant overhead on dynamic task graphs, this overhead can be amortized when each node does enough work. We also see that to get the best speedup, it pays to exploit both the DAG-level parallelism and the parallelism within each task. Nabbit allows a programmer to exploit both seamlessly.

2.8 Conclusions

This chapter has explored the problem of task-graph execution in dynamic-threading platforms. I presented Nabbit, the first library in a fork-join platform for provably efficient parallel execution of task graphs with arbitrary dependencies. Nabbit demonstrates that a dynamic-threading platform can provide arbitrary task-graph synchronization which composes with ordinary fork-join programs and provides efficient theoretical bounds on completion time. I conclude this chapter with a discussion of related work on task-graph synchronization and potential directions for future research.

Other Approaches for Task-Graph Execution

Intel TBB [110] provides a low-level library interface for parallel tasks which can support the execution of arbitrary task graphs. Using TBB's interface, a programmer can simulate the execution of a fork-join platform, provided that the programmer codes the appropriate continuations for functions. Nabbit provides a simpler high-level interface for task graphs than TBB. For example, TBB requires programmers to explicitly code reference-counting for tasks to determine when they are ready to execute.

Dynamic task graphs have often been studied in the context of linear algebra algorithms. For example, Johnson et al. [76] describe an interface for a dynamic task graph which was motivated by parallel algorithms for sparse LU factorization. SMPSs [18] was motivated by the problem of parallelizing dense linear-algebra libraries. As discussed in Section 2.5, both SMPSs and Concurrent Collections use asynchronous Cholesky factorization as a benchmark for performance evaluation.

Parallel programming with arbitrary task-graph synchronization is closely related to the model of parallel futures [57,58]. In this model, a programmer can declare an expression as ***future*** to spawn the computation of the expression. A runtime may need to suspend the current strand of execution when it encounters a ***touch*** of the future — a request for the value of a future — if that future that has not been computed yet. A platform which supports futures can be used to construct and execute arbitrary task graphs: each node in the task graph can be declared as a future which touches all its predecessor futures that it depends on. Similarly, one can also use I-structures [17] to specify and execute task graphs.

In principle, one can always simulate a computation with futures using an equivalent strongly dynamic task graph in Nabbit. This transformation requires two types of task-graph nodes in Nabbit: data nodes and control nodes. Data nodes correspond to futures — the key for a data node is a pointer to a future,and COMPUTE for the node calculates the value of the future. Control nodes correspond to the computation occurs between points in the program where futures are created or touched. More precisely, consider a control node A that represents the computation before the touch of a future. Then, the computation of A generates two nodes, a node X for the touched future, and a node B for the continuation of execution after the touch of X, with B also depending on X.

This simulation technique is of limited utility in practice, however, because it requires programmers to manage function continuations explicitly. In a parallel language with futures, a function `f` usually stores local variables in frames on a stack. Also, parallel languages may implement a cactus-stack abstraction [61], which allows `f` to access stack frames of its ancestor functions. To simulate such a program execution using Nabbit, one must ensure that the COMPUTE of each control node has access to the same stack frames that it would in a normal program execution. Also, programmers also need to inject the appropriate cleanup code into the COMPUTE of control tasks, that is, code for reclaiming memory for stack frames and task nodes which are no longer needed by the computation.

Theoretical Analysis of Work-Stealing

Arora et al. [16] explored the behavior of work-stealing schedulers for computation DAGs with arbitrary dependencies. More specifically, they describe and analyze a work-stealing scheduler for computation DAGs whose nodes have in-degree and out-degree of at most 2. This analysis can be applied to task graphs with arbitrary dependencies, assuming that one inserts dummy task nodes (e.g., as in the proof of Corollary 2.4). This work-stealing scheduler uses essentially the same algorithm as Cilk, which was analyzed by Blumofe and Leiserson [30]. Since Nabbit is coded directly in Cilk++, it is able to provide guarantee the same provably efficient bounds as in [16,30]. The analysis for Nabbit extends these results to more precisely account for contention in task graphs where nodes have arbitrary degree. Although an arbitrary graph is theoretically equivalent to a graph with degree at most 2, arbitrary graphs may occur more frequently in practice because of the overheads of creating and managing dummy nodes.

Spoonhower et al. [113] analyze the behavior of the work-stealing scheduler of [16] for computations that use parallel futures. Their analysis counts the number of "deviations" in a parallel execution — roughly, the number of times a parallel execution of the computation DAG differs from a serial execution order. These bounds on time and space can be applied to Nabbit as well, since it uses the same scheduler.

Directions for Future Research

The exploration of task-graph execution using Nabbit suggests several opportunities for future work.

One question to investigate is whether one can extend Nabbit to support pipeline computations. As discussed in [26, 113], one use for parallel futures is to implement programs with parallel pipelines. Intel TBB [110] supports pipeline computations whose stages form a linear chain, but conceptually, one might like to execute a pipeline with stages arranged as an arbitrary DAG. In principle, one could replicate stages of a pipeline computation in Nabbit and unroll it into a large task graph, similar to how compilers unroll multiple iterations of a loop. This approach may waste space, however, as it does not efficiently reuse task-graph nodes for different stages as items pass through the pipeline. One may be able to extend Nabbit to overcome these difficulties.

It would also be interesting to explore how one might provide integrated runtime or compiler support for task-graph execution in a dynamic-threading platform. One of the benefits of Nabbit's design is that it requires no modifications to the Cilk runtime. Nevertheless, having integrated runtime support would likely improve the performance and robustness of Nabbit. For example, in the Cilk++ and Intel Cilk Plus platforms, the runtime appears to impose a limit on the maximum spawn-depth of functions which constrains the kind of task graphs that Nabbit can execute. To execute a static task graph with V_∞ nodes on its longest path, Nabbit uses at most $O(V_\infty)$ stack frames for each worker. The default spawn-depth limit in Cilk++ and Intel Cilk Plus is relatively small however. More precisely, these platforms appear to limit V_∞ to be less than some value on the order of 1000. In fact, if one considers the programming interface for Nabbit, one does not need to have a stack at all, since the COMPUTE for each task node in Nabbit should only read values from the heap anyway! Thus, with integrated runtime support for Nabbit in Cilk, one could eliminate this stack issue altogether. Integrated runtime support would also likely decrease overheads and make fine-grained task-graph synchronization more practical. Compiler support might also enable platforms to provide a more convenient programming interface for specifying task-graph synchronization.

Chapter 3

Helper Locks

Locks are a commonly used synchronization primitive in multithreaded programs. The runtime scheduler for dynamic-threading platforms such as Cilk are generally not designed with locks in mind, however. Thus, although programmers frequently write Cilk code that utilizes nested parallelism, Cilk does not effectively support nested parallelism inside critical sections that are protected by locks.

This chapter introduces the idea of a ***helper lock***, a new kind of lock that can protect a critical section with nested parallelism.[1] When a processor fails to acquire a helper lock `L`, the processor can help complete the parallel critical section A which may be holding lock `L` instead of simply waiting for the lock `L` to be released. Using helper locks, dynamic-threading platforms can support the composition of parallel functions with locked-based synchronization while still providing provable guarantees on performance.

I discuss three main contributions in this chapter:

1. Helper locks, a new synchronization abstraction that enables Cilk programmers to exploit nested parallelism inside critical sections.
2. The design of ***HELPER***, runtime support which enables a dynamic-threading platform to execute computations with helper locks. To support helper locks, HELPER introduces a new "parallel region" construct into Cilk.
3. Theoretical bounds on the completion time and stack-space usage for computations executed using HELPER. These bounds apply even for computations with parallel regions nested to an arbitrary depth.

This chapter also describes a prototype implementation of HELPER in MIT Cilk, which demonstrates that platforms can provide this runtime support efficiently without hurting the performance of existing programs that do not require helper locks.

Chapter Outline

The remainder of this chapter is organized as follows. Section 3.1 motivates the use of helper locks through the example of a resizable hash table, and it explains why modifications to the existing Cilk runtime are needed to support helper locks. Section 3.2 describes the "parallel region" construct, the language construct that HELPER adds to MIT Cilk to

[1] The design of helper locks represents joint work [10] with Kunal Agrawal and Charles E. Leiserson.

implement helper locks. Section 3.3 describes the runtime support for parallel regions. Section 3.4 states the time bound for HELPER. Section 3.5 describes a formal execution model, which is used in Section 3.6 to prove the time bound. Section 3.7 presents the space bound for HELPER. Section 3.8 describes a prototype implementation of helper locks, and Section 3.9 gives some empirical results on microbenchmarks and on a resizable hash table benchmark. Finally, Section 3.10 concludes the chapter with a discussion of related work.

3.1 Motivating Example

This section motivates the utility of helper locks through the example of a resizable hash table and discusses some of the challenges in supporting helper locks in Cilk. A simple implementation of a resizable hash table using a reader-writer lock can exhibit poor performance because the resize operation can generate a large critical section. Helper locks can improve performance by allowing a programmer to exploit parallelism inside large critical sections.

Cilk programmers can use ordinary locks to protect critical sections. For example, the code in Figure 3-1 uses a reader-writer lock to implement a resizable concurrent hash table. Every insert operation acquires the table's reader lock, and a resize operation acquires the table's writer lock. Thus, insert operations (on different buckets) may run in parallel, but a table resize cannot execute in parallel with any insertion.

Unfortunately, the resize operation in Figure 3-1 can be a performance bottleneck because it entails a large critical section. While a worker is executing a resize, it prevents any other worker which might be trying to insert into the table from making progress. For a function which can have many inserts into the hash table happening in parallel, e.g., the function shown in Figure 3-2, this hash-table implementation exhibits poor performance.

When a large critical section introduces such a bottleneck, programmers typically try to improve performance by reducing the size of the critical section. For example, since a resizable hash table is an amortized data structure, one might try to deamortize the data structure by performing the resize incrementally, as inserts occur. This approach often requires programmers to use fine-grained locking however. Using fine-grained locks generally makes programs more difficult to implement and debug, since it increases the number of interleavings of parallel strands of execution that the programmer must reason about.

Helper locks provide an alternative approach, improving program performance while still retaining the simplicity of coding large critical sections. Instead of using fine-grained locks to try to split a large critical section into many smaller critical sections, programmers can use a helper lock to mitigate the effect of the large critical section by parallelizing the section.

In many parallel programming environments, including Cilk, when a worker fails to acquire a lock, it typically spins (i.e., waits) until the lock is released. Workers that spin on a lock protecting a large critical section may waste a substantial number of processor cycles waiting for the lock to be released however. Consequently, for large critical sections, many lock implementations avoid tying up the blocked worker thread by yielding the scheduling quantum after spinning for a short length of time. This altruistic strategy works well if there are other jobs that can use the cycles in a multiprogrammed environment, but if the

```
 1 void insert(HashTable* H, Key k, void* val) {
 2   int idx = hashcode(H, k);
 3   List* L = H->buckets[idx];
 4   list_lock(L);
 5   list_insert(L, k, val);
 6   list_unlock(L);
 7 }
 8 int try_insert(HashTable* H, Key k, void* val) {
 9   int success = try_read_acquire(H->resize_lock);
10   if (!success)
11     return FAILED;
12   insert(H, k, val);
13   release(H->resize_lock);
14   return SUCCESS;
15 }
16 void resize_table(HashTable* H) {
17   List** new_buckets;
18   int new_n = H->num_buckets*2;
19   new_buckets = create_buckets(new_n);
20   for (int i = 0; i < H->num_buckets; i++) {
21     rehash_list(H->buckets[i], new_buckets, new_n);
22   }
23   free_buckets(H->buckets);
24   H->buckets = new_buckets;
25   H->num_buckets = new_n;
26 }
27 void resize_table_if_overflow(HashTable* H) {
28   if (is_table_overflow(H)) {
29     write_acquire(H->resize_lock);
30     resize_table(H);
31     release(H->resize_lock);
32   }
33 }
```

Figure 3-1: A resizable concurrent hash table implemented using a reader-writer lock. Each bucket in the hash table is protected by its own separate lock (lines 4–6). Since each insert (line 8) acquires the lock `H->resize_lock` as a reader, inserts can occur in parallel. The resize operation (line 27) acquires `H->resize_lock` in writer mode, however, and so a resize cannot run concurrently with inserts or another resize.

```
void rand_inserts(HashTable* H, int n) {
  cilk_for(int i = 0; i < n; i++) {
    int res;
    Key k = rand();
    do {
      res = try_insert(H, k, k);
    } while (res == FAILED);
    resize_table_if_overflow(H);
  }
}
```

Figure 3-2: An example Cilk function which performs n hash-table insertions, potentially in parallel. The `cilk_for` keyword (in bold) indicates that the iterations of the loop are allowed to execute in parallel. After every insertion, the loop checks whether the insertion triggered an overflow and resizes the table if necessary.

focus is on completing the current computation, one would like to be able to put blocked workers to work on the computation itself, i.e., help to complete the critical section holding the lock.

A naive strategy for putting a blocked worker to work on the computation itself is for the worker to suspend the function that failed to acquire a lock and engage in work-stealing. Unfortunately, this strategy can waste resources in dramatic fashion in a dynamic threading platform such as Cilk. Turning back to the example of the hash table, imagine what would happen if one worker p write-acquires the resize lock while another worker p' attempts an insertion. Unable to read-acquire the resize lock, p' would suspend the insertion attempt and steal work. What work is lying around and available to steal? Why another insertion, of course! Indeed, while p is tooling away trying to resize, p' might attempt (and fail) to insert most of the items in the hash table, one at a time. More precisely, p' might systematically attempt an insertion, fail to read-acquire the resize lock, suspend the insertion, and then proceed to the next insertion.

This strategy is not only wasteful of p''s efforts, it results in profligate space usage. Each time a continuation is stolen, the runtime system requires an activation frame to store local variables. Thus, for the hash table example, the `rand_inserts` function could generate as much as $\Theta(n)$ space in suspended insertions. In general, the space requirement could grow as large as the total work in the computation. In contrast, Cilk's strategy of simply spinning and yielding, though potentially wasteful of processor cycles, uses at most $O(P \lg n)$ space on P processors in this example. This result follows from Cilk's provable bounds on space usage [27] (also reviewed in Theorem A.4 of Appendix A), since each worker requires only $O(\lg n)$ space to spawn n inserts in a divide-and-conquer fashion.

The idea of helper locks is to employ the blocked workers in productive work while controlling space usage by enlisting them to help complete a large parallel critical section, or ***critical region***.[2] For helper locks to be useful, however, the critical region must be parallelized and the runtime system must be able to migrate blocked workers to work on the

[2] In this chapter, I use the term "critical region" to refer to a critical section that may be parallel.

critical region. Ordinary Cilk-style work-stealing, as reviewed in Appendix A, is inadequate to this task, because stealing occurs at the top of the deque, and the critical region may be deeply nested where workers cannot find it. For example, in Figure 3-1, when an insertion triggers a resize, the parallel work of the resize (i.e., `rehash_list` in line 21) is generated at the bottom of a deque, with work for `rand_inserts` above it. In fact, in this example, by employing Cilk-style work-stealing, workers would more likely block on another insertion than help complete the resize. Thus, to enable workers to efficiently help within a critical region in Cilk, additional runtime support is necessary.

3.2 Parallel Regions and Helper Locks

This section introduces the "parallel region" construct, the additional language construct that HELPER adds to Cilk to support helper locks. This section first presents the parallel region construct provided by HELPER. Then, it shows how to use parallel regions to implement helper locks and the resizable table example from Section 3.1.

Parallel Region Construct

To support helper locks in Cilk, HELPER adds two new constructs, `start_region` and `help_region`, for supporting "parallel regions" and adds some runtime functions for managing helper locks. Figure 3-3 gives a code example that illustrates these language and runtime extensions.

The function `f` in Figure 3-3 illustrates the use of HELPER's `start_region` construct. The function `f` starts 10 instances of a parallel function `foo`, each as a ***parallel region*** that is protected by a randomly chosen lock from an array of helper locks.

Helper locks have the semantics of a mutex but with the additional behavior of a worker helping in a parallel region when the mutex cannot be acquired. For example, if two regions `foo(0)` and `foo(1)` are protected by the same lock `L`, then only one instance can execute at a time. Because `L` is a helper lock, however, if worker p_1 tries to start `foo(1)` but discovers that `foo(0)` already holds the helper lock `L`, then p_1 will try to help complete `foo(0)` instead of blocking and waiting on the lock `L`. For p_1 to help in `foo(0)` effectively, `foo(0)` must be a parallel-function instance, i.e., it must expose potential parallelism using the `spawn` or `parallel_for` keywords. Otherwise, if `foo(0)` is a serial function, there is no benefit to having more than one worker help to complete the region.

In general, while `start_region` for a parallel region A protected by a lock `L` is executing, we say that the parallel region A ***holds*** the helper lock `L`. Also, we say that `L` is held because of a ***region acquire*** by the worker p that started the region. Unlike a normal lock that protects a serial critical section, it is less useful to say that the worker p that started a region "holds" the helper lock, because multiple workers can be concurrently executing work from the same critical region.

Although helper locks are most useful for protecting parallel critical regions, it is still sometimes useful to allow a worker p to hold a helper lock for a serial critical section. Thus, HELPER provides calls for a ***serial acquire*** of a helper lock, i.e., a lock acquire that protects a serial critical section. In Figure 3-3, lines 17–21 show the serial acquire

```
 1 int f(HelperLock** L_array, int R) {
 2   int ans[10];
 3   cilk_for(int i = 0; i < 10; ++i) {
 4     HelperLock* L = L_array[rand() % R];
 5     ans[i] = start_region[L] foo(i);
 6   }
 7   int sum = 0;
 8   for (int i = 0; i < 10; ++i) {
 9     sum += ans[i];
10   }
11   helper_lock_array_destroy(L_array);
12   return sum;
13 }

14 int g(HelperLock* L) {
15   int ans = 0;
16   help_region[L];
17   while (!helper_serial_acquire(L)) {
18     help_region[L];
19   }
20   ans = bar();
21   helper_serial_release(L);
22   return ans;
23 }

24 int main(void) {
25   int x, y, z;
26   HelperLock** L_array;
27   L_array = helper_lock_array_create(100);
28   x = spawn f(L_array, 100);
29   y = spawn g(L_array[0]);
30   z = g(L_array[1]);
31   sync;
32   printf(``Final answer: %d\n'', x+y+z);
33   helper_lock_array_destroy(L_array);
34   return 0;
35 }
```

Figure 3-3: Code demonstrating the use of HELPER's parallel region construct and other runtime support for helper locks. The start_region and help_region keywords in the code appear in bold. Line 5 shows the start_region construct, which makes a call to the (potentially parallel) function foo(i), protected by a helper lock L. Line 16 shows the help_region construct, which indicates that the current worker should try to help complete any region that is currently holding the helper lock L. Lines 17–21 show the serial acquire and release of a helper lock. The function bar in line 20 *cannot* be a parallel function because HELPER requires that all parallel critical sections use the start_region construct.

and release of a helper lock L, which protects the call to a serial function bar. While an instance of g is holding lock L because of a serial acquire, an instance of f cannot start a region protected by L, i.e., no other worker can perform a region acquire of L. A helper lock that is held because of serial acquire behaves as normal mutex: when a worker fails to acquire the lock, it blocks and waits as with an ordinary lock.[3]

HELPER also provides a help_region construct, which enables workers that might be waiting for a serial acquire on a lock L to help complete any parallel region that might be holding L because of a region acquire. For example, in lines 16 and 18 of Figure 3-3, a worker *p* executing an instance of g tries to help in the parallel region holding the lock L before it tries to acquire lock L serially. If an instance of foo is currently holding L due to a region acquire, then *p* tries to complete the region. If, however, another instance of g is holding L, or L is not held, then the help_region call returns immediately without any effect.

Finally, HELPER provides library functions for creating and destroying helper locks, as shown in lines 27 and 33. Helper locks, like ordinary locks, are created and destroyed as ordinary objects in memory.

Resizable Table Example

By using the parallel region construct, and by extending helper locks to support reader-writer lock semantics on a serial acquire, one can improve the performance of the resizable table example from Figure 3-1. Figure 3-4 gives an implementation of a resizable table using helper locks.

In Figure 3-4, line 16, the resize_table method from Figure 3-1 is started as a parallel region, protected by the helper lock H->resize_lock. The start_region construct in line 30 enables parallelism inside the resize operation, i.e., the resize_table method can now have a parallel loop in lines 20 through 22 instead of being required to execute serially.

Figure 3-4 also uses the help_region construct to provide the desired helping behavior when a worker fails an insert because of an ongoing resize operation. In line 11 of Figure 3-4, the try_insert method uses the help_region construct to direct a worker to help in the region holding H->resize_lock when it fails to acquire this lock.

Finally, for this example, instead of using an ordinary serial acquire of the helper lock, we use a ***serial read-acquire*** of H->resize_lock, which has reader-writer semantics. With a serial read-acquire, the critical regions for inserts in lines 9–13 can run in parallel with each other. Inserts cannot, however, run concurrently with the resize operation, since this is protected by a region acquire of H->resize_lock. Said differently, as with a normal reader-writer lock, a successful acquire in line 9 prevents any worker from starting a new region for a resize in line 30.

More generally, one can imagine designing more complicated kinds of helper locks, which have different semantics from mutexes or reader-writer locks. In programs that use

[3] Conceptually any serial acquire could be replaced with a region acquire without changing the meaning of a program, since workers attempting to "help" in a serial region would end up waiting for the region to complete. In practice, however, a region acquire is more restrictive linguistically, since the critical region must correspond to a function call, whereas the critical section for a serial acquire need not be lexically scoped.

```
 8 int try_insert(HashTable* H, Key k, void* val) {
 9   int success =
        helper_serial_Racquire(H->resize_lock);
10   if (!success)
11     { help_region[H->resize_lock]; return FAILED; }
12   insert(H, k, val);
13   helper_serial_Rrelease(H->resize_lock);
14   return SUCCESS;
15 }

16 void resize_table(HashTable* H) {
17   List** new_buckets;
18   int new_n = H->num_buckets*2;
19   new_buckets = create_buckets(new_n);
20   cilk_for (int i=0; i<H->num_buckets; i++) {
21     rehash_list(H->buckets[i], new_buckets, new_n);
22   }
23   free_buckets(H->buckets);
24   H->buckets = new_buckets;
25   H->num_buckets = new_n;
26 }

27 void resize_table_if_overflow(HashTable* H) {
28   if (is_table_overflow(H)) {
29
30     start_region[H->resize_lock] resize_table(H);
31
32   }
33 }
```

Figure 3-4: Resizable hash table example using helper locks. This code example modifies lines 16 through 33 from Figure 3-1 to parallelize the resize operation for the table, and modifies lines 8 through 15 to enable inserts that block on the a resize to help complete the resize operation.

multiple helper locks, one must also consider the issue of potential deadlocks. Section 3.3 discusses some of these issues after explaining the runtime support and execution model for HELPER.

3.3 HELPER Runtime

This section describes the execution model for HELPER and the runtime data structures HELPER utilizes for scheduling parallel regions. First, the computation DAG model for Cilk (reviewed in Appendix A) is extended with two new constructs, `start_region` and `help_region`. Then, this section describes "deque pools" and "deque chains," the state that HELPER maintains to support parallel regions.

Computation DAGs

The execution of a Cilk computation $\mathcal{C}$ can be thought of as generating a ***computation DAG*** $\mathcal{G}(\mathcal{C}) = (V(\mathcal{C}), E(\mathcal{C}))$.[4] In this DAG, a function invocation F can be represented as a subDAG of $\mathcal{G}(\mathcal{C})$ enclosed between a ***source node*** $\texttt{source}(F)$ and a ***sink node*** $\texttt{sink}(F)$.

It is useful to define some terminology concerning the computation DAG. For any node $u \in V(\mathcal{C})$, let $\texttt{ipred}(v)$ be the set of ***immediate predecessors*** of v in $\mathcal{G}$, i.e., $u \in \texttt{ipred}(v)$ if and only if there exists an edge $(u,v) \in E(\mathcal{C})$. When it is clear that $\texttt{ipred}(v)$ has exactly one element (e.g., $\texttt{ipred}(v) = \{u\}$), we abuse notation and say that $\texttt{ipred}(v) = u$. Similarly, let $\texttt{isucc}(u)$ be the set of ***immediate successors*** of u, i.e., $v \in \texttt{isucc}(u)$ if and only if there exists an edge $(u,v) \in E(\mathcal{C})$. These definitions of $\texttt{ipred}(u)$ and $\texttt{isucc}(u)$ are valid only because $\mathcal{G}$ has the canonical form described in Section A.2, i.e., $\mathcal{G}$ is equal to its transitive reduction [12].

For HELPER, `start_region` represents a special function call. Thus, in a computation DAG $\mathcal{G}(\mathcal{C})$, a parallel region B corresponds to a subDAG of $\mathcal{G}(\mathcal{C}) = (V(\mathcal{C}), E(\mathcal{C}))$, enclosed between a source node $\texttt{source}(B)$ and a sink node $\texttt{sink}(B)$. Let $\texttt{regions}(\mathcal{C})$ denote the set of all regions in a computation $\mathcal{C}$.

A `help_region` call h is also a special function call with two nodes, $\texttt{source}(h)$ and $\texttt{sink}(h)$, except with two extra conditions. First, $\texttt{sink}(h)$ is always the immediate predecessor of $\texttt{source}(h)$ in the DAG. Second, when a worker executing a computation $\mathcal{G}(\mathcal{C})$ encounters a node $\texttt{source}(h)$, before it returns to execute $\texttt{sink}(h)$, it may jump to work on a different parallel region because of a helper lock. In an execution, we assume that a `start_region` call that fails (because the corresponding helper lock is already held by a different region) generates `help_region` calls in the computation DAG until it manages to acquire the lock.

We can also define some terminology for parallel regions. We say that a region B ***contains*** a node u in the computation *DAG* if u is along some path from $\texttt{source}(B)$ to $\texttt{sink}(B)$. We say that region B ***owns*** u, denoted by $\texttt{rg\_owner}(u) = B$, if B is the most deeply nested parallel region containing u. This ownership is uniquely defined because parallel regions are function calls, and function calls are always properly nested. Conversely, we say that u ***belongs*** to B if $\texttt{rg\_owner}(u) = B$.

[4] For details, see Section A.2 of Appendix A.

Finally, we assume that computation DAGs exhibit a certain canonical structure when starting regions, which allows us to define the "region predecessor" and "region successor" of certain nodes in the DAG. For a region A which starts a nested region B due to a `start_region` call, let node $u = \texttt{ipred}(\texttt{source}(B))$ and let $v = \texttt{isucc}(\texttt{sink}(B))$. We assume that u and v are uniquely defined, that both belong to A, and both have in-degree and out-degree of 1. We say that v is the ***region successor*** of u, denoted by $v = \texttt{rg\_succ}(u)$. Similarly, we say that $u = \texttt{rg\_pred}(v)$ is the ***region predecessor*** of v. For the case of a `help_region` call $h \in \texttt{helpCalls}(\mathcal{C})$, we say that $\texttt{source}(h) = \texttt{rg\_pred}(\texttt{sink}(h))$ and $\texttt{sink}(h) = \texttt{rg\_succ}(\texttt{source}(h))$, i.e., the source and sink of h form a pair of matching region predecessor and successor nodes.

Figure 3-5 shows an example program with parallel regions, and Figure 3-6 shows the computation DAG that might be generated by an execution of the program. The `help_region` call h_1 in region C (line 7 in Figure 3-5) generates the two nodes $x_1 = \texttt{source}(h_1)$ and $y_1 = \texttt{sink}(h_1)$ in the computation DAG (Figure 3-6). The region predecessors and successors for each region have also been highlighted.

Deque Pools and Deque Chains

The HELPER runtime maintains a "deque pool" data structure for every parallel region A to schedule the work of A and allow multiple workers to help complete A. Conceptually, one can think of the ordinary Cilk runtime (as reviewed in Section A.2) as maintaining a single deque pool of P deques for a computation $\mathcal{C}$, with one deque for each worker p. HELPER extends Cilk by maintaining a deque pool for every region A to handle the scheduling of region A. At any point in time, each worker p has an "active deque" q which it is operating on. When p runs out of work on q, it tries to steal work from within the deque pool containing q. To allow for nesting of parallel regions, every worker organizes its deques from different regions into a "deque chain."

To describe HELPER's runtime data structures more precisely, HELPER maintains a ***deque pool*** for every parallel region A, which we denote by $A.dqpool$. Conceptually, a deque pool is an object which stores P deques, a dedicated deque for every worker. A deque pool contains the following fields:

- $A.dqpool[p]$: the memory allocated for the deque for p in the pool for A.
- $A.valid[p]$: flag that is TRUE when p has been ***assigned to*** the pool.
- $A.psize$: a ***size*** field tracking the number of workers that have been assigned to the pool.
- $A.done$: a ***done flag*** for signaling when region A has completed.

Although $A.dqpool$ has space allocated for P deques, at any given time, not all workers may be assigned to A. Let $dq(p,A)$ be the pointer to $A.dqpool[p]$ if $A.valid[p]$ is TRUE (i.e., p has been assigned to A) and NULL otherwise. For a dthreaded computation $\mathcal{C}$, we let $\mathcal{C}.dqpool$ denote the root deque pool, i.e., the top-level pool of P deques used by an ordinary Cilk computation without parallel regions. Intuitively, HELPER uses the deque pool $A.dqpool$ for self-contained scheduling of region A on A's assigned workers. While p is assigned to A, when p tries to steal work, it randomly steals only from deques $q \in A.dqpool$.

```
void A()  { spawn a1(); spawn a2(); a3(); sync; }
void a1() { start_region[BLock] B(); }
void a2() { start_region[DLock] D(); }
void a3() { start_region[FLock] F(); }

void B()  { start_region[CLock] C(); }

void C()  { spawn c1(); c2(); sync; }
void c1() { help_region[DLock]; }
void c2() { s1; s2; }

void D()  { spawn d1(); d2(); sync; }
void d1() { s3; s4; s5; }
void d2() { start_region[ELock] E(); }

void E()  { spawn e1(); e2(); sync; }
void e1() { help_region[FLock]; s6; }
void e2() { s7; s8; }

void F()  { spawn f1(); f2(); sync; }
void f1() { s9; s10; }
void f2() { s11; s12; s13; }
```

Figure 3-5: A code example in HELPER with parallel regions *A* through *F*. This code assumes that each region is started using a distinct helper lock (`ALock` through `FLock`).

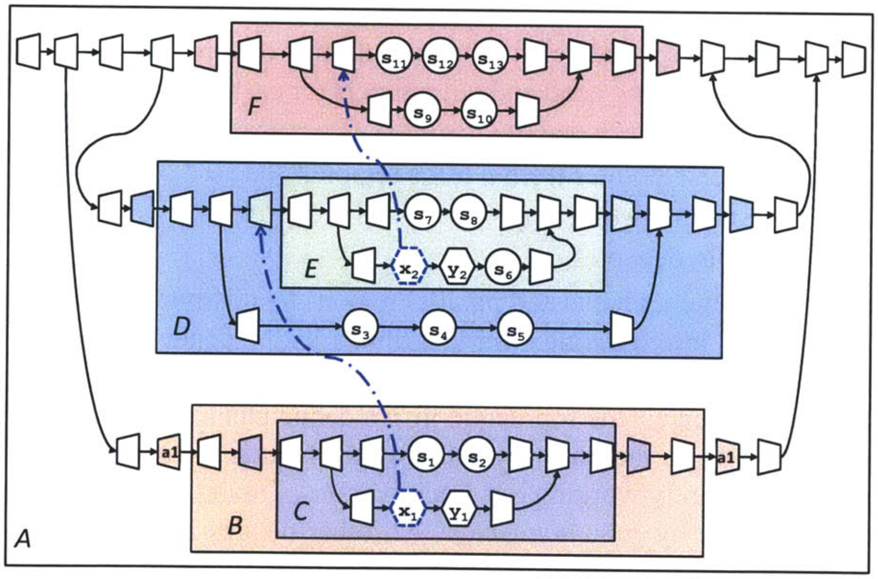


Figure 3-6: A computation DAG with parallel regions for Figure 3-5. The dashed arrows represent `help_region` calls. Source and sink nodes corresponding to region predecessors and successors have been shaded (e.g., `source(a1)` and `sink(a1)`).

Operation	**Updates**

InitPool(A):
Initialize deque pool
for region A.
	$A.psize = 0$
	$A.done =$ FALSE
	for all workers p:
	$\quad A.valid[p] =$ FALSE
	$\quad A.dqpool[p] \rightarrow region = A$

EnterPool(p,B): Worker p enters region B with new deque q.	$B.valid[p] =$ TRUE $B.psize = B.psize + 1$ $q_A = p \rightarrow activeDQ$ $dq(p,B) \rightarrow parent = q_A$ $dq(p,B) \rightarrow child =$ NULL $p \rightarrow activeDQ = dq(p,B)$ $q_A \rightarrow child = dq(p,B)$
LeavePool(p,B): Worker p leaves region B	assert ($B.done ==$ TRUE) assert ($dq(p,B) == p \rightarrow activeDQ$) $B.valid[p] =$ FALSE $B.psize = B.psize - 1$ $q_A = dq(p,B) \rightarrow parent$ $q_A \rightarrow child =$ NULL $p \rightarrow activeDQ = q_A$

Figure 3-7: Pseudocode for maintaining deque pools and deque chains in HELPER.

Every deque maintains several fields to support nesting of parallel regions. More specifically, in addition to maintaining head and tail pointers for each deque, HELPER also maintains the following fields for each deque $q = dq(p,A)$:

- $q \rightarrow region$: a reverse pointer from q to its region A, i.e., $q \in A.dqpool$.
- $q \rightarrow parent$: the parent deque of q.
- $q \rightarrow child$: the child deque of q.

To allow nesting of helper locks and parallel regions to an arbitrary depth, each worker p_i organizes its deques into a ***deque chain***, a linked list of deques using the parent and child pointers. Each deque in the chain for a worker belongs to the deque pool of a distinct region. For a computation C, the top deque in p's deque chain belongs to $C.dqpool$, the top-level deque pool, while the bottom deque in the chain is the ***active deque*** of p_i, i.e., the deque that p_i is currently working on. Each worker p_i maintains a pointer to its active deque, which we denote by $p_i \rightarrow activeDQ$. All other deques in the deque chain p_i are ***inactive***. When a worker p enters or leaves a region in HELPER, p changes deque pools and deque chains in a straightforward fashion, preserving the invariant that p is always currently working in the region at the bottom of its deque chain. Figure 3-7 summarizes the actions taken by HELPER to maintain deque pools.

Normally, when there is work on $p_i \rightarrow activeDQ$ (p_i's active deque), p_i only changes the tail pointer of this deque. When p_i runs out of work, however, it work-steals from deques within the deque pool of $p_i \to activeDQ \to region$, that is, the region for p_i's active deque. While p_i is executing normally, other workers p_j may concurrently work-steal from any deque q in p_i's chain, changing the head of these deques.

HELPER allows a worker p to ***enter*** a region B (i.e., be assigned $B.dqpool$) in three ways. First, p can enter B when p successfully starts region B by executing `start_region`. Second, p can enter a region B because of a `help_region` call because of a helper lock `L` while `L` is held by region B. This second case can occur because of an explicit `help_region` call for lock `L` or because of an implicit call generated by a failed `start_region` call using lock `L`. Finally, p can enter a region B due to random work-stealing.

This third case captures the key difference between work-stealing in HELPER as compared to ordinary Cilk. Suppose p with $p \to activeDQ \to region = A$ tries to steal another deque q in $A.dqpool$. If q is empty, but $q \to child$ exists and belongs to $B.dqpool$ for some other region B, then instead of simply failing to steal, then p enters region B. We call this case an ***entering steal*** (from A into B) for p.

In HELPER, the case of workers leaving regions is simpler. Once HELPER assigns a worker p to a region A, p does not ***leave*** region A until A completes, i.e., until some worker has set $A.done$ as `true`. We adopt this policy for simplicity of implementation and to guarantee good theoretical bounds. If a worker can leave a region before it completes, then in theory, a worker might incur significant synchronization overhead by repeatedly reentering and leaving the same region. Later in Section 3.4, we discuss some of the theoretical implications and possible extensions to HELPER for allowing workers to leave regions early when the region has no work to steal.

Example Execution

Figure 3-8 illustrates deque pools and deque chains for a simple computation DAG (Figure 3-6) executed using 4 workers. In this example, worker p_1 enters regions A through F. All regions are assumed to acquire different helper locks. Initially, p_1 starts a region B, p_4 randomly steals from p_1 in $A.dqpool$ and starts region D, and p_2 steals from p_4 in A and starts region F. Next, p_1 starts a region C nested inside B. Then, p_3 randomly work-steals from p_1 in A and enters B and C. Afterward, p_1 inside C makes a `help_region` call on the lock for D, enters D, and steals from p_4 in D. Finally, p_1 makes a `help_region` call on the lock for F and enters F.

Deadlock Freedom with Helper Locks

As with ordinary (nonhelper) locks, an arbitrary nesting of helper locks risks deadlock. From the HELPER execution model, however, we can state restrictions on helper locks which guarantee that helper locks do not introduce a deadlock.

For a computation C with parallel regions protected by helper locks, construct a ***region graph*** whose nodes are regions from `regions(`C`)` and which contains an edge (A,B) if there is a `start_region` call from region A which creates a nested region B, or a `help_region` call from a region A into region B. We say B is a ***child region*** of A if (A,B) is an edge in the region graph. If a program has serial acquires of helper locks, one should

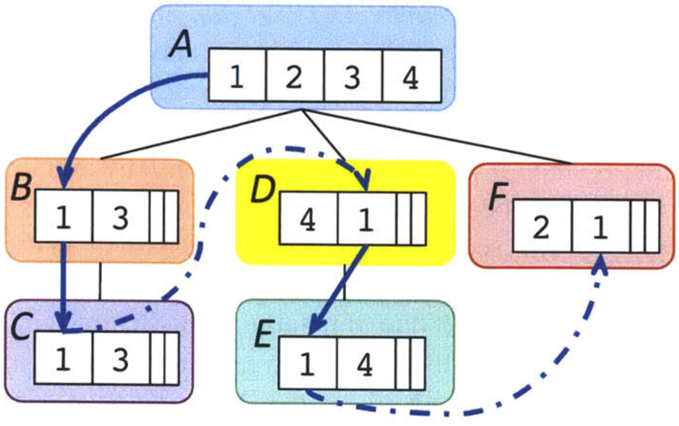


Figure 3-8: A snapshot of deque pools during execution of the DAG from Figure 3-6. Numbers correspond to workers in the pool.

consider each serial acquire as a potential help_region call when constructing the region graph.

If this region graph is acyclic, then HELPER guarantees that helper locks do not introduce deadlock. Entering steals in HELPER do not introduce any additional deadlocks because a worker can only enter a region B from a region A if there was already an edge (A,B) in the region graph from a previous start_region or help_region call.

One way to guarantee the region graph is acyclic is to maintain a strict partial order for acquiring helper locks — the same discipline which is often used to ensure deadlock freedom for ordinary locks. To avoid deadlock, the programmer must also ensure that any nested parallelism within critical sections is properly encapsulated using parallel regions. Specifically, the keywords spawn and sync should not appear inside a critical section unless both are enclosed within a parallel region initiated using start_region. This condition essentially requires that programmers use only the mechanisms provided by HELPER to generate nested parallelism in critical sections.

Implementing Deque Pools

The abstract description of deque pools described in this section can be implemented in a variety of ways in practice. One simple implementation of deque pools is to maintain an array of P deques with a dedicated slot for every worker. Each worker can add or remove itself from the array without waiting on other workers. Synchronization is required, however, to maintain the size field for a deque pool.

In fact, HELPER only needs to know whether a deque pool is empty or nonempty. Thus, for the purposes of analyzing execution time, we assume that synchronization for entering and leaving a region only requires $O(\lg P)$ time for a deque pool with P processors. One way to satisfy this theoretical assumption is to use a protocol based on a binary-tree network of size $O(P)$ and depth $O(\lg P)$ to track whether the pool is empty. With this scheme, each worker waits at most $O(\lg P)$ time to enter the region, and it takes at most $O(\lg P)$ time for all workers to leave the deque pool once the done flag of the region is set. In practice, updating an atomic counter is likely to be more efficient for modest values of P.

3.4 Completion-Time Bound

This section states the completion-time bound provided by HELPER. Stating this bound requires us to first generalize the definitions of work and span for computation DAGs to account for parallel regions. Then, we consider HELPER's time bound and study it in the context of the original bound for Cilk. Finally, we compare these theoretical bounds to the bounds for other potential implementations of helper locks. The proof of the completion-time bound is deferred until Section 3.6.

Definitions

To state the time bound, we use the computation DAG model described in Section 3.3 with some additional definitions.[5] We represent the execution of a program as a computation $\mathcal{C}$ with execution DAG $\mathcal{G}(\mathcal{C})$, where each node in $\mathcal{G}(\mathcal{C})$ represents a unit-time task, and each edge represents a dependence between tasks. We assume the computation executes on hardware with P processors with one worker assigned to each processor. The remainder of this chapter considers only programs that generate acyclic region graphs. Thus, any execution is guaranteed to be deadlock-free.

For any region A, define the ***work*** of A, denoted $T_1(A)$, as the number of nodes in the subDAG enclosed between $\texttt{source}(A)$ and $\texttt{sink}(A)$. Define the ***span*** of A, denoted $T_\infty(A)$, as the number of nodes along a longest path from $\texttt{source}(A)$ to $\texttt{sink}(A)$. We also define a "region" work and span for each region A, which considers only nodes belonging to A. Define the ***region work*** of A as

$$\tau_1(A) = |\{u \in V(A) \ : \ \texttt{rg\_owner}(u) = A\}| \ ,$$

i.e., the number of nodes in the DAG belonging to A. For any path z through the graph $\mathcal{G}$, define the ***path length*** of z for region A, denoted by $\texttt{rg\_path\_length}(z,A)$, as the number of nodes u along path z with $\texttt{rg\_owner}(u) = A$. Let $\texttt{paths}(u,v)$ denote the set of all paths from u to v in $\mathcal{G}(\mathcal{C})$. Define the ***region span*** of A as

$$\tau_\infty(A) = \max_{z \in \texttt{paths}(\texttt{source}(A),\texttt{sink}(A))} \texttt{rg\_path\_length}(z,A) \ ,$$

that is, the maximum path length for A over all paths z from $\texttt{source}(A)$ to $\texttt{sink}(A)$. Intuitively, the region span of A is the time to execute A on an infinite number of processors, assuming that A's nested regions complete instantaneously. For example, in the computation DAG in Figure 3-6, region D has $T_1(D) = 24$, $\tau_1(D) = 11$, $T_\infty(D) = 15$, and $\tau_\infty(D) = 9$.

The completion-time bound for HELPER depends on the structure of the region graph, i.e., the number of regions and how regions are nested and connected to each other. Let

[5] See Section A.2 for a complete description of this model.

$N = |\mathtt{regions}(\mathcal{C})|$ denote the number of regions in the region graph, and let M denote the number of edges in the region graph.

To model the contention due to serial acquires of helper locks, we define the ***bondage*** of a computation $\mathcal{C}$, denoted by $b(\mathcal{C})$, as the total number of nodes contained within critical regions for serial acquires of all helper locks.

Finally, we also define some quantities on the entire computation DAG $\mathcal{G}(\mathcal{C})$. For a computation $\mathcal{G}(\mathcal{C})$ with parallel regions, we define the ***aggregate region span*** as

$$\widetilde{T}_\infty = \sum_{A \in \mathtt{regions}(\mathcal{C})} \tau_\infty(A) \,.$$

As an abuse of notation, we let T_1 (without any arguments) denote the work of the entire computation, i.e., $T_1(\mathcal{C})$.

Statement of Completion-Time Bound

Let $T_P(\mathcal{C})$ be the running time of a computation $\mathcal{C}$ running on P processors using HELPER. We prove that a computation $\mathcal{C}$ with N regions, M edges between regions, work T_1, aggregate span $\widetilde{T}_\infty$, and bondage b runs in expected time

$$\mathrm{E}\left[T_P(\mathcal{C})\right] = O\left(\frac{T_1}{P} + \widetilde{T}_\infty + M\ln\left(1+\frac{PN}{M}\right) + b(\mathcal{C})\right). \qquad (3.1)$$

Moreover, for any $\varepsilon > 0$, we prove that with probability at least $1-\varepsilon$, the execution time is

$$T_P(\mathcal{C}) = O\left(\frac{T_1}{P} + \widetilde{T}_\infty + M\ln\left(1+\frac{PN}{M}\right) + b(\mathcal{C}) + \lg\left(\frac{1}{\varepsilon}\right)\right). \qquad (3.2)$$

Interpretation and Discussion of Bounds

To understand what the completion-time bound in Equation (3.1) means, we can compare it to the completion-time bound for a computation without regions. The ordinary bound for randomized work stealing [30] says that the expected completion time is $O(T_1/P + T_\infty)$. The bound in Equation (3.1) exhibits three differences.

First, there is an additive term of $M\ln(1+PN/M)$. We shall show that $N-1 \le M \le PN$. Therefore, in the best case, when $M = N-1$, the term reduces to $N\ln P$. This case occurs when a computation has no contention on helper locks, i.e., when no `help_region` calls are made. In the worst case, when $M = PN$, this term is equal to PN. This worst case assumes that each worker assigned to a region enters from a different region, whereas we expect in practice that most regions would have a limited number of entry points.

Even in the worst case, when the additive term is PN, if the number of parallel regions is small, then this term is insignificant compared to the other terms in the bound. Since parallel regions are meant to represent large critical sections, we expect N to be small in most programs. For example, in the hash-table example from Section 3.1, if we perform n insertions, only $O(\lg n)$ resizes occur during the execution. Furthermore, even if there are a large number of parallel regions, if each parallel region A is sufficiently large ($\tau_1(A) =$

$\Omega(P^2)$) or long ($\tau_\infty(A) \geq \Omega(P)$), then we have $PN = O(T_1/P + \widetilde{T}_\infty)$, and the PN term is asymptotically absorbed by the other terms. These conditions seem reasonable, since we expect programmers to use parallel regions only for large critical sections. Programmers should generally use serial acquires to protect small critical regions.

Second, the bound on HELPER's completion time involves the term $\widetilde{T}_\infty$ instead of T_∞. If the number of parallel regions is small, then the term $\widetilde{T}_\infty$ is generally close to T_∞. Even for programs with a large number of parallel regions, the $\widetilde{T}_\infty$ term does not slow down the execution if regions are sufficiently parallel. To understand why, we can rewrite the worst-case time bound from Equation (3.1) as follows:

$$\begin{aligned} \mathrm{E}[T] &= O\left(\frac{T_1}{P} + \widetilde{T}_\infty + PN + b(C)\right) \\ &= O\left(\sum_{A \in \texttt{regions}(C)} \left(\frac{\tau_1(A)}{P} + \tau_\infty(A) + P\right) + b(C)\right) . \end{aligned}$$

If the $b(C)$ term does not asymptotically dominate the parallel-work term, we can see that HELPER provides linear speedup if for each region A, the ***region parallelism*** $\tau_1(A)/\tau_\infty(A)$ is sufficiently large. More precisely, assuming $b = O(T_1/P)$, the program achieves linear speedup if the region parallelism of each region is at least $\Omega(P)$. In the hash-table example, a region A that resizes a table of size k completely in parallel has span $\tau_\infty(A) = O(\lg k)$, and thus on most machines, the parallelism should greatly exceed the number of processors.

Third, we have the additive term $b(C)$, term which accounts for contention on ordinary (serial) lock acquires. In most programs, it is unlikely that the real completion time with HELPER will include all of the bondage. It is difficult to prove a tighter bound, however, since theoretically, there exist computations for which no runtime system could do any better. For example, if all lock acquires (serial and region acquires) are for the same lock L, then, ignoring the $M \ln(1 + PN/M)$ term, HELPER's bound is asymptotically optimal, i.e., no runtime system can execute the computation asymptotically faster.

Comparison with Alternative Implementations

We now compare the bounds for parallel regions in HELPER with two other alternatives. The first option does not allow helping. When a worker p blocks on a lock L, it just waits until L to become available. Using this traditional implementation for locks, the completion time of a program with critical sections (either expressed as parallel regions or just expressed sequentially) can be $\Omega(T_1/P + \sum_{A \in \texttt{regions}(C)} \tau_1(A) + b(C))$. Notice that the second term is the sum of region work over all regions, as compared to Equation (3.1), which has the sum of region spans. Therefore, if the program has large (and highly parallel) critical sections (as in the hash-table example), then this implementation may run significantly slower than with helper locks.

Second, we can compare against an implementation where a worker that blocks on a lock suspends its current work and randomly work-steals. As the hash-table example from Section 3.1 illustrates, this implementation may use $\Omega(n)$ space for n inserts. In contrast, as we argue later in Section 3.7, HELPER uses $O(P \lg n)$ space for this example, which is typically much smaller than n in practice.

3.5 HELPER Execution Model

This section presents a formal execution model for HELPER based on the computation DAG model for Cilk. First, this section incorporates parallel regions, deque pools, and deque chains into the DAG model. Then, it describes the invariants on the computation DAG satisfied by the model. Section 3.6, uses this execution model to prove completion-time bounds for HELPER.

We extend the computation DAG model described in Section 3.3 (and Section A.2 of Appendix A) to capture the dynamic execution of a computation $\mathcal{C}$ using HELPER. We consider each deque q in HELPER as conceptually maintaining an ***assigned*** node, denoted as $q \rightarrow assigned$.[6] Normally, the field $p_i \rightarrow activeDQ \rightarrow assigned$ stores the node in the computation DAG that the worker p_i is currently executing.

For inactive deques q, HELPER maintains the invariant that $q \rightarrow assigned$ represents the node where p will resume execution once q becomes active again. More precisely, when a worker p with active deque q_A in a region A starts a nested region B with deque $q_B = q_A \rightarrow child$, the node $q_A \rightarrow assigned$ is set to `isucc(sink(`B`))` to preserve this invariant. Similarly, when p enters B from A because of a `help_region` call h, $q_A \rightarrow assigned$ is set to `sink(`h`)`. We refer to any such assigned node on an inactive deque as a ***blocked node***, since it cannot be executed until region below in the deque chain completes. If a worker creates the child deque $q \rightarrow child$ because of work-stealing, an inactive deque q may have $q \rightarrow assigned = \text{NULL}$.

In the computation DAG model, the assigned node of deques change on each ***step*** as workers execute ***instructions***. In a normal Cilk computation, a worker can execute one of the following instructions: a primitive operation (`primOp`), a call of a function (`call`) or its return (`cReturn`), a spawn of a function (`spawn`) or its return (`sReturn`), a sync instruction (`sync`), or a steal (`steal`). See Figure A-3 for a more detailed description of how these instructions change assigned nodes.

The behavior of most instructions remains the same after the addition of parallel regions for HELPER. More precisely, for a worker p, the `primOp`, `call`, `cReturn`, `spawn`, `sync`, and `sReturn` instructions operate on the active deque $p \rightarrow activeDQ$ in HELPER in the same way that these instructions operate on the (only) deque of p in the original DAG model. HELPER only modifies the behavior of the `steal` instruction, and adds new instructions for parallel regions, namely the `startRegion`, `helpRegion`, and `finishRegion` instructions.

Consider a worker p with $q = p \rightarrow activeDQ$, and $q \rightarrow assigned = u \neq \text{NULL}$. In other words, p has active deque q with an assigned node u. Then, HELPER takes the following actions depending on the instruction it issues.

1. **`startRegion`**: Suppose that `isucc(`u`)` = `source(`B`)`, i.e., p is starting a nested region B. First, HELPER updates deque pools by calling INITPOOL(B) and then

[6] This definition differs slightly from the model in Section A.2, in which the assigned node is associated with a worker p instead of a deque.

ENTERPOOL(p,B). Let $q_B = p \rightarrow activeDQ$ after these calls. Then, this instruction changes $q \rightarrow assigned$ from u to $\texttt{isucc}(\texttt{sink}(B)) = \texttt{rg\_succ}(u)$, and then sets $q_B \rightarrow assigned$ to $\texttt{source}(B)$.

2. **helpRegion**: Suppose that u is a help_region call for a region B. Then, to update deque pools, HELPER first calls ENTERPOOL(p,B). Let $q_B = p \rightarrow activeDQ$ after entering the pool. Then, this instruction changes $q \rightarrow assigned$ to $\texttt{rg\_succ}(u) = \texttt{isucc}(u)$, and sets $q_B \rightarrow assigned$ to NULL.

3. **finishRegion**: Suppose that $u = \texttt{sink}(B)$, i.e., p is finishing region B. To update deque pools, HELPER first sets $B.done$ to TRUE, waits for every other worker p' assigned to B to call LEAVEPOOL(p,B), and then calls LEAVEPOOL(p,B) itself. Let $q_A = p \rightarrow activeDQ$ after this process. Because $q_A \rightarrow assigned$ was set correctly when p entered region B, p resumes execution with the assigned node $q_A \rightarrow assigned$.[7]

Alternatively, suppose that p has $q = p \rightarrow activeDQ$ and $q \rightarrow assigned =$ NULL. Then p tries to randomly work-steal within its current deque pool via a steal instruction.

4. **steal**: Let $A = q \rightarrow region$ be the region for p's active deque. Then p chooses a ***victim*** deque $q_v \neq q$ uniformly at random from $A.dqpool$. The action of p depends on the state of q_v:

   (a) **Successful steal**. If q_v is not empty, then p takes the top node w from q_v and sets $q \rightarrow assigned$ to w.

   (b) **Unsuccessful steal**. If q_v is empty *and* $q_v \rightarrow child =$ NULL, then the steal attempt fails.

   (c) **Entering steal**. If q_v is empty but $q_v \rightarrow child$ exists, then p enters the region $B = q_v \rightarrow child \rightarrow region$, assuming that $B.done$ has not yet been set to TRUE. For an entering steal, worker p executes ENTERPOOL(p,B) and then sets $p \rightarrow activeDQ \rightarrow assigned$ to NULL.

   (d) **Leaving steal**. If $B = q_v \rightarrow child \rightarrow region$ has $B.done =$ TRUE, i.e., B is already finished, then the steal attempt fails.

For the last two cases of a steal instruction, we refer to any deque q which is empty but has a child $q \rightarrow child \neq$ NULL as an ***exposed deque***. Ordinary work-stealing in Cilk requires only the first two cases, but HELPER adds the latter cases to handle parallel regions.

Finally, we can show that HELPER preserves the following invariants on computation DAGs and deques.

Lemma 3.1. *For a computation C with DAG $\mathcal{G}(C)$, consider a deque $q = dq(p,A)$. Let $v_0 = q \rightarrow assigned$, and let $v_1, v_2, \ldots, v_k$ be the nodes on q arranged from bottom to top. Let u_i be any node $u_i \in \texttt{ipred}(v_i)$ if v_i is an unblocked node, or $\texttt{rg\_pred}(v_i)$ if v_i is a blocked node. The execution model maintains the following invariants:*

1. *For all i, we have $\texttt{rg\_owner}(v_i) = A$, i.e., node v_i belongs to region A.*

[7] Depending on the specific implementation, the last worker to leave B (i.e., the worker which sets $B.psize$ to 0) might also need to clean up B's deque pool.

2. *For all $i \geq 1$, node u_i is unique.*
3. *For all $i \geq 1$, v_i is an unblocked node and u_i is a spawn node.*
4. *The assigned node v_0 is blocked if and only if $q \neq p \rightarrow activeDQ$, i.e., q is inactive.*
5. *For $i = 2, 3, \ldots, k$, node u_i is a predecessor of node u_{i-1}.*
6. *If q is active, then v_0 is a successor of u_1.*
7. *If q is inactive, then u_0 is a successor of u_1.*

Proof. The invariants can be proved by induction on the actions of the execution model. Invariant 1 holds because HELPER only assigns a node u to q if $q = p \rightarrow activeDQ$ and $\texttt{rg\_owner}(u) = q \rightarrow region$, and because a `spawn` node u can only push nodes v with $\texttt{rg\_owner}(u) = \texttt{rg\_owner}(v)$ (i.e., from the same region) onto q. Invariants 2 and 3 hold because nodes v are added to a deque only when a worker p works on an assigned node u which is a spawn node (i.e., a `spawn` instruction from Figure A-3). Invariant 4 holds because only the `startRegion` and `helpRegion` instructions can create a blocked node v and only on a deque q that has a child deque.

We can show Invariants 5 and 6 hold by checking the various instructions for HELPER. For each instruction, consider a worker p with active deque $q = p \rightarrow activeDQ$ and assigned node $u = q \rightarrow assigned$. If u is a spawn node, then $q \rightarrow assigned$ is changed to a node w and its sibling v is pushed onto q. In this case, HELPER preserves Invariant 6 since w is a successor of u. When p executes a `steal` instruction, p resets the assigned node $q \rightarrow assigned$, but the deque q is guaranteed to be empty, and thus the invariants are automatically preserved on a `steal`. For all other instructions, p always replaces its assigned node u with a node v which is a successor of u.

Finally, for Invariant 7, we know that right before an active deque q becomes inactive, $v_0 = q \rightarrow assigned$ is a successor of u_1. When q becomes inactive, we know that $q \rightarrow assigned$ is changed from v_0 to $\texttt{rg\_succ}(v_0)$, which is a successor of v_0. Therefore, q satisfies Invariant 7 after it becomes inactive. □

3.6 Proof of Completion-Time Bounds

This section proves the bounds on completion time explained in Section 3.4. First, this section gives an overview of the proof, classifying the types of steps each processor can take on each time step. The main challenge of the proof is to account for the steps that processors spend stealing. Next, it bounds the number of steal attempts that HELPER performs in an execution by bounding the number of "contributing" steal attempts and then the number of "entering" steal attempts. Finally, it combines these pieces together to prove the completion-time bound.

Completion-Time Bound for HELPER

To prove the bound the completion time from Equation (3.1), we account for each possible action that each processor (worker) p can take on each time step. Every processor step falls into one of the following categories:

- **Working**: p executes its assigned node.

- **Entering**: p waits as it tries enter a region A. This category counts the time that p spends executing $\text{ENTERPOOL}(p,B)$ for some region B — in the `startRegion` (Case 1), the `helpRegion` instruction (Case 2), or in an entering steal (Case 4c).
- **Leaving**: p waits as it tries to leave a region A. This category counts the time workers p spend in $\text{LEAVEPOOL}(p,B)$ (Case 3).
- **Waiting**: p spin-waits on a helper lock `L` which is currently held by a serial lock acquire.
- **Stealing**: p attempts to steal work randomly in a region A.

For a computation $\mathcal{C}$ executed on P workers, let N be the number of regions, and let T_1 and $\widetilde{T}_\infty$ be the work and aggregate span of $\mathcal{G}(\mathcal{C})$, respectively. Then, $\mathcal{G}(\mathcal{C})$ has exactly T_1 working steps. As described in Section 3.3, the entering cost is $O(\lg P)$ per worker, and once a region A completes, each worker in A's deque pool leaves within $O(\lg P)$ time. Therefore, the total number of entering and leaving steps is $O(PN \lg P)$. The number of steps that workers spend waiting (on serial acquires of helper locks) is bounded by $Pb(\mathcal{C})$: in the worst case, $P-1$ processors can be waiting on a particular lock `L` while one processor executes a node in the critical region of `L`.

We bound the number of steal attempts by partitioning them into three types which we bound individually.

- A ***contributing*** steal for region A is any steal attempt in A that occurs on a step when A has no exposed deques.
- A ***leaving*** steal for region A is any steal attempt that occurs while A has an exposed deque and while there exists a region B and a worker p satisfying three conditions: B is a child of A, $p \rightarrow activeDQ \rightarrow region = B$, and $B.done = \text{TRUE}$.
- A steal attempt in a region A which is neither a leaving nor contributing steal is considered an ***entering*** steal.

We can extend the potential function argument of [16] (reviewed in Section A.3) to show that every region A has $O(P\tau_\infty(A))$ contributing steals in expectation. This result implies that the expected total number of contributing steals is bounded by $O(P\widetilde{T}_\infty)$. We can bound the number of leaving steals by P times the number of time steps when any leaving step occurs. Since any worker p in A's deque pool leaves within $O(\lg P)$ time of A's completion, and no worker enters the same region twice, the total number of time steps when any worker can be leaving a region is at most $O(N \lg P)$. Therefore, the total number of leaving steals is $O(PN \lg P)$. Finally, we can show that the total number of entering steals is at most $O(PM \ln(1+PN/M))$.

Contributing Steals

To analyze the number of contributing steals, we first extend the potential functions from Definition A.1, Section A.3 to handle parallel regions. Then, we divide steal attempts for a region A into "rounds," arguing that after a certain number of rounds, the potential of a region A is likely to reduce to 0 with high probability.

Our definition of potential for nodes, deques, and regions requires some terminology. For a node $u \in A$, its ***depth*** $d(u)$ is the maximum path length for region A over all paths from `source`(A) to u. The ***weight*** of a region A is $w(A) = \tau_\infty(A) - d(A)$.

Definition 3.1. *Consider a node $u \in A$. The **potential of node** u is*

$$\Phi(u) = \begin{cases} 3^{2w(u)-1} & \textit{if } u = q \rightarrow \textit{assigned for an active deque } q, \\ 3^{2w(u)} & \textit{if } u \textit{ is ready or blocked,} \\ 0 & \textit{if } u = \text{NULL}. \end{cases}$$

Similarly, let q be the deque belonging to a worker p, and let u be p's assigned node or NULL *if p has no assigned node. The **potential of deque** q is*

$$\Phi(q) = \Phi(u) + \sum_{v \in q} \Phi(v).$$

*Finally, the **potential of region** A is*

$$\Phi(A) = \sum_{q \in A.dqpool} \Phi(q).$$

In Definition 3.1, any inactive deque q has $u = \text{NULL}$, that is, q has no assigned node that contributes potential.

As in [16], we can show that weights of nodes along any deque strictly decrease from top to bottom (Lemma 3.2) and that the potential of a region never increases over time (Lemma 3.3).

Lemma 3.2. *Consider the execution of a computation C that generates an execution DAG $\mathcal{G}(C)$. For any deque q owned by a worker p, let $v_1, v_2, \ldots, v_k$ be the nodes (from $\mathcal{G}(C)$) in q ordered from the bottom of the deque to the top. Then, we have*

$$w(v_1) \leq w(v_2) < \cdots < w(v_k).$$

Also, let $v_0 = q \rightarrow assigned$. If $v_0 \neq \text{NULL}$, then either

1. *q is active and $w(v_0) \leq w(v_1)$, or*
2. *q is inactive and $w(v_0) < w(v_1)$.*

Proof. Define the u_i's as in Lemma 3.1. First, we prove the lemma for nodes v_i for $i \geq 1$. Since $\mathcal{G}(C)$ is a series-parallel DAG, the depth of any `spawn` node u is always 1 less than the depth of its two children. By Invariant 3, for all the unblocked nodes $v_1, v_2, \ldots, v_k$ in the deque, we have $d(u_i) = d(v_i) - 1$. Invariant 5 implies that u_i is a predecessor of u_{i-1} in $\mathcal{G}(C)$ for all $i \geq 2$. Thus, we have $d(v_{i-1}) > d(v_i)$ for all $i = 2, 3, \ldots, k$. Converting from depth to weight yields $w(v_1) < w(v_2) < \cdots < w(v_k)$.

Next, we prove the lemma for the assigned node v_0 in the case where $v_0 \neq \text{NULL}$. In this case, by Invariant 4, we know that either q is active or q is inactive and v_0 is blocked.

- If q is active, then by Invariant 6, we know v_0 is a successor of u_1. Thus, $d(v_0) > d(u_1)$. We know $d(u_1) = d(v_1) - 1$ because u_1 is a spawn node. Thus, we know $d(v_0) \geq d(v_1)$, since all depths are integers. Therefore, we have $w(v_0) \leq w(v_1)$.

- If q is inactive, then by Invariant 6, we know u_0 is a successor of u_1, or equivalently $d(u_0) > d(u_1)$. We know $d(u_1) = d(v_1) - 1$ because u_1 is a spawn node. We also have $d(u_0) = d(v_0) - 1$, because any inactive deque q must have an assigned node v_0 which is blocked, and then we must have $u_0 = \texttt{rg\_pred}(v_0)$. Thus, we get $d(v_0) > d(v_1)$, or equivalently, $w(v_0) < d(v_1)$.

□

Lemma 3.3. *During the execution of a computation C, the potential $\Phi(A)$ of any region $A \in \texttt{regions}(C)$ increases from 0 to $3^{2\tau_\infty(A)-1}$ when $\texttt{source}(A)$ becomes ready. At no other time does $\Phi(A)$ increase.*

Proof. To prove the lemma, one can check all the cases of the execution model. In general, actions that execute or assigned nodes within a region A affect only $\Phi(A)$, and by the proof given in [16], these actions only decrease the potential.

The new cases that HELPER introduces are the `start_region` and `help_region` instructions. Consider a worker p with active deque $q = p \rightarrow \mathit{activeDQ}$ and $A = q \rightarrow \mathit{region}$. Also, let $u = q \rightarrow \mathit{assigned}$.

- **`start_region`**: If u starts a nested region B, then $q \rightarrow \mathit{assigned}$ changes to $v = \texttt{rg\_succ}(u)$. By definition of the region successor, we have $d(v) = d(u) + 1$, and hence, the potential decrease in region A is $3^{2w(u)-1} - 3^{2w(v)} = 2 \cdot 3^{2w(v)}$. Also, since p creates a new deque q_B for the new region B, and sets $q_B \rightarrow \mathit{assigned}$ to $\texttt{source}(B)$, $\Phi(B)$ increases to $3^{2\tau_\infty(B)-1}$.

- **`help_region`**: If u corresponds to a `help_region` call, then as in the previous case, $q \rightarrow \mathit{assigned}$ is set to $\texttt{rg\_succ}(u)$, and the potential of A decreases as in the case of the `start_region` call. For the region B that p enters however, $\Phi(B)$ is unchanged because p creates an empty deque for B.

□

To bound the number of contributing steals in a region A, we divide steal attempts in A into rounds. The first round $R_1(A)$ for region A begins when $\texttt{source}(A)$ is set as the assigned node of some worker. A ***round*** $R_k(A)$ ends after at least P contributing steals in A have occurred in the round, or when A ends. Any round can have at most $2P - 1$ contributing steals (if P steals occur in the last time step of the round). Therefore, a round has $O(P)$ contributing steal attempts in region A. We say $R_{k+1}(A)$ begins on the same time step of the next contributing steal in A after $R_k(A)$ has ended. A round for A may have many entering steals and there may be gaps of time between rounds.

First, we bound the number of rounds for a region A. At the beginning of $R_k(A)$, let $D_k(A)$ be the sum of potentials of all nonempty deques that are not exposed in A's deque pool, and let $E_k(A)$ be the sum of potentials due to empty or exposed deques. The potential of A at the beginning of round k can then be expressed as $\Phi_k(A) = D_k(A) + E_k(A)$.

Lemma 3.4. *In a computation C, for any round $R_k(A)$ of a region $A \in \texttt{regions}(C)$, we have*

$$\Pr\{\Phi_k(A) - \Phi_{k+1}(A) \geq \Phi_k(A)/4\} \geq 1/4.$$

Proof. We shall show that each of $D_k(A)$ and $E_k(A)$ decrease by at least a factor of $1/4$ with probability at least $1/4$. The lemma holds trivially for the last round of A, since the region completes.

Any deque that contributes potential to $D_k(A)$ has, at the beginning of the round, at least one node on top that can be stolen. By definition, every round except possibly the last has at least P steal attempts. Thus, Lemma 8 from [16] allows us to conclude directly that $D_k(A)$ decreases by a factor of $1/4$ with probability $1/4$. Any entering steals that occur only reduce the potential further.

To show that $E_k(A)$ reduces by at least $1/4$, we use the definition of rounds and contributing steals, which imply that on the first time step of round k, region A has no exposed deques. Thus, any deque q that contributes to $E_k(A)$ must be empty. An empty deque q without an assigned node contributes nothing to $E_k(A)$. An empty deque q with an assigned node u reduces q's contribution to $E_k(A)$ by more than $1/4$, since u is executed in the round's first time step. □

Lemma 3.5. *In a computation C, for any region $A \in \mathtt{regions}(C)$, the expected number of rounds is $O(\tau_\infty(A))$, and the number of rounds is $O(\tau_\infty(A) + \ln(1/\varepsilon))$ with probability at least $1-\varepsilon$.*

Proof. The proof is analogous to Theorem 9 in [16]. Call round k ***successful*** if $\Phi_k(A) - \Phi_{k+1}(A) \geq \Phi_k(A)/4$, i.e., the potential decreases by at least a $1/4$ fraction. Lemma 3.4 implies that $\Pr\{\Phi_{k+1}(A) \leq 3\Phi_k(A)/4\} \geq 1/4$, i.e., a round is successful with probability at least $1/4$. The potential for region A starts at $3^{2\tau_\infty(A)-1}$, ends at 0, and is always an integer. Thus, a region A can have at most $8\tau_\infty(A)$ successful rounds. Consequently, the expected number of rounds needed to finish A is at most $32\tau_\infty(A)$.

For the high probability bound, we can use Chernoff bounds as in [16]. In general, using Chernoff bounds, one can show that after $(2R + 4\ln(1/\varepsilon))/\beta$ coin flips, where each coin flip comes up heads with probability β, the probability of having fewer than R heads is less than $1-\varepsilon$. Since each round succeeds with probability at least $\beta = 1/4$ and we can have at most $R = 8\tau_\infty(A)$ successful rounds, the probability of having more than $64\widetilde{T}_\infty + 16\ln(1/\varepsilon)$ rounds is less than $1-\varepsilon$. □

Entering Steals

To bound the number of entering steals, we require some definitions. Intuitively, we divide the entering steals for a particular region A into "intervals" and subdivide intervals into "phases". For every region A, we divide the time steps when A is active into ***entering intervals***, which are separated by leaving steps for child regions B. More precisely, entering interval 1 for A, denoted $I_1(A)$, begins with the first entering steal in A and ends with the next leaving step belonging to any region B such that (A,B) is an edge in the region graph for $\mathcal{G}$, or if A completes. Similarly, $I_k(A)$ begins with the first entering steal in A after $I_{k-1}(A)$ completes.

We also subdivide an interval $I_k(A)$ into ***entering phases***, separated by successful entering steals in A. In general, phase j of $I_k(A)$, denoted $I_{k,j}(A)$, begins with the first entering steal of $I_k(A)$ after phase $j-1$ and ends with the next successful entering steal, or at the end of $I_k(A)$.[8] We say that a phase $I_{k,j}(A)$ has ***rank*** j. Define an entering phase as ***complete*** if it ends with a successful entering steal. Every interval has at most one incomplete phase (the last).

Intuitively, entering intervals and phases are constructed so that during an interval for A, the number of exposed deques only increases and the probability of a successful entering steal increases with the rank of the current phase.

Lemma 3.6. *For all regions A, the entering interval $I_k(A)$ satisfies the following properties:*

1. *Interval $I_k(A)$ has at most $P-1$ phases.*
2. *During any phase of rank j, the probability that a given steal attempt succeeds is at least j/P.*

Proof. At the beginning of the interval, there is at least one exposed deque in A, and therefore, at least one worker has moved from A to some child region of A. Every complete phase ends with a successful entering steal and a successful entering steal causes a worker to move from A to some child region. Therefore, after j phases, at least $j+1$ workers have moved to some child region. Moreover, an interval $I_k(A)$ ends with any leaving step for any child region of A. Therefore, no worker reenters region A from its child region during an interval. Due to these two facts, after $P-1$ complete phases, no processors are working in region A, and there can be no more entering steals. On the other hand, if the last phase is incomplete, then the number of complete phases is less than $P-1$. In either case, the interval has at most $P-1$ phases.

No exposed deque is eliminated during an interval since no processor reenters A, and thus the number of exposed deques in A never decreases. At the beginning of $I_{k,1}(A)$, the first phase of every interval, region A has at least one exposed deque. Therefore, during phase 1, the probability of any entering steal succeeding is at least $1/P$. As we argued above, after $j-1$ phases complete, at least j workers have entered a child region and each of these workers leave behind an exposed deque in region A (since their deque $dq(p,A)$ is empty). Hence in phase j, interval $I_{k,j}(A)$ has at least j exposed deques,[9] and the probability of hitting one of these exposed deques on any steal attempt is at least j/P.

□

Lemma 3.7. *Let C be a computation executed on P processors whose region graph has N regions and M edges, and let r_m be the number of phases of rank m. The following properties hold:*

1. $N-1 \leq M \leq PN$.
2. *The number of entering intervals over all regions is at most M.*
3. $M \geq r_1 \geq r_2 \cdots \geq r_{P-1}$.

[8] The end of $I_k(A)$ and the beginning of $I_{k+1}(A)$ occur on different time steps, but the end of phase j and beginning of phase $j+1$ within an interval can occur within the same time step. If multiple workers try to steal from the same exposed deque in a time step, they all succeed, and multiple entering phases end in that time step.

[9] Phase j might have more than j such deques, since help_region calls into A or steals within A can exposes new deques.

4. *The total number of complete entering phases over all regions is at most $N(P-1)$, and the total number of incomplete entering phases is at most M.*
5. *For any integer $K \geq r_1$, let $\alpha(K) = \left\lceil \sum_{j=1}^{P-1} r_j/K \right\rceil$. Then, for any nonincreasing function f, we have*
$$\sum_{j=1}^{P-1} r_j f(j) \leq K \sum_{j=1}^{\alpha(K)} f(j).$$

Proof. (1) Since only P workers total can enter a region B, and in the worst case each worker enters along a different edge, every region B can have in-degree at most P, and thus we have $M \leq PN$. Moreover, every region except $\mathcal{G}$ also has in-degree at least 1, which implies that $N-1 \leq M$.

(2) For any region A, let d_A be the out-degree of A in the region graph for $\mathcal{G}$. Intervals for region A end only when some child B completes and takes a leaving step. Also, the time after the leaving step for the last child region B does not form an interval because there can be no more entering steals in A. Thus, A has at most d_A intervals. Summing over all regions, we can have at most M intervals total.

(3) By construction, every interval can have at most one phase of a given rank m, and thus we have $M \geq r_1$. Also, an interval can have a phase of rank $m+1$ only if it has a phase of rank m, which implies $r_m \geq r_{m+1}$.

(4) Every complete entering phase has a successful entering steal. We can have at most $P-1$ successful entering steals for each region A, since one worker enters the region when A is started and at most $P-1$ can enter through stealing. Every interval can have at most 1 incomplete phase.

(5) The quantity $\sum_{j=1}^{P-1} r_j f(j)$ can be viewed as the sum of entries in a M by $P-1$ grid, where row z contains phases for the zth interval and the jth column corresponds to phases of rank j. Each entry (z, j) has a value that is $f(j)$ if interval z has a phase of rank j, or 0 otherwise. This grid contains at most r_j nonzero entries in each column and $\sum_j r_j$ entries total. Since the function $f(j)$ and r_j are both nonincreasing and $K \geq r_1$, conceptually, by moving entries left into smaller columns, we can compress the nonzero entries of the grid into a compact K by $\alpha(K) = \left\lceil \sum_j r_j/K \right\rceil$ grid without decreasing the sum. The value of the compact grid is an upper bound for the value of the original grid. The compact grid has at most $\alpha(K)$ columns, with each column j having value at most $Kf(j)$, giving the desired bound. □

We can now bound the expected number of entering steals.

Theorem 3.8. *Consider a computation C executed on P processors, and suppose that the region graph of C has N regions and M edges. The expected number of entering steal attempts is*
$$O\left(PM\ln\left(1+\frac{PN}{M}\right)\right).$$

Proof. Suppose that $\mathcal{G}$ requires r_m phases of rank m. Number the intervals of $\mathcal{G}$ arbitrarily, and let $Q(z,j)$ be the random variable for the number of entering steals in phase j of the zth interval, or 0 if the interval or phase does not exist. By Lemma 3.7, there can be at most M intervals. Thus, the total number Q of entering steals is given by

$$Q = \sum_{z=1}^{M} \sum_{j=1}^{P-1} Q(z,j) .$$

Lemma 3.6 implies that $E[Q(z,j)] \le P/j$, since each entering steal in the phase succeeds with probability at least j/P. Thus, linearity of expectation gives us

$$E[Q] = \sum_{z=1}^{M} \sum_{j=1}^{P-1} E[Q(z,j)] \le \sum_{z=1}^{M} \sum_{j=1}^{P-1} \frac{P}{j} = \sum_{j=1}^{P-1} r_j \left(\frac{P}{j} \right) . \tag{3.3}$$

We can apply Fact 5 of Lemma 3.7 with $K = M$ and $f(j) = 1/j$, since $M \ge r_1$. Then, we have $\alpha(K) = \left\lceil \sum_j r_j / M \right\rceil \le 1 + \lceil PN/M \rceil$, since $\sum_j r_j$ is the total number of phases, which is bounded by $(P-1)N + M$. Thus, we have

$$E[Q] \le PM \sum_{j=1}^{\alpha(K)} \frac{1}{j} = PMH_{\alpha(K)} ,$$

where $H_n = \sum_{i=1}^{n} 1/i = O(\ln n)$ is the nth harmonic number, which completes the proof. □

We can also bound the number of entering steals with high probability.

Theorem 3.9. *Consider a computation $\mathcal{C}$ executed on P processors, and suppose that the region graph for $\mathcal{C}$ has N regions and M edges. The number of entering steal attempts is*

$$O\left(PM \ln\left(1 + \frac{PN}{M}\right) + P\ln^2 P + P \ln\left(\frac{1}{\varepsilon}\right) \right)$$

with probability at least $1 - \varepsilon$.

Proof. Let r_j be the number of phases of rank j in $\mathcal{C}$. Conditioned on a particular computation $\mathcal{C}$ with fixed values for r_j, we want to bound the number of entering steals as a function of the r_j. Our argument involves dividing the entering steals into $\lceil \lg P \rceil$ "classes" of entering steals, where each class counts only entering steals from phases of similar rank. We then employ a union-bound over all classes to obtain the final bound.

Define a ***class-m entering steal*** as any entering steal that occurs in a phase of rank j satisfying $2^{m-1} \le j < 2^m$. Thus, class 1 contains entering steals in rank-1 phases, class 2 contains rank-2 and rank-3 steals, class 3 includes rank-4 through rank-7 phases, and so forth. Let $R_m = \sum_{j=2^{m-1}}^{2^m-1} r_j$ be the number of phases in class m. A computation can have at most R_m successful class-m steals, since each successful steal ends a phase. We shall show that it is unlikely to have too many unsuccessful steals in a class.

As in Lemma 3.5, we can use Chernoff bounds to calculate the probability of having more than R_m class-m steals. By Lemma 3.6, each class-m entering steal succeeds with probability $\beta \geq 2^{m-1}/P$. Thus, after

$$\frac{PR_m}{2^{m-2}} + \frac{4P}{2^{m-1}} \ln\left(\frac{\lceil \lg P \rceil}{\varepsilon}\right)$$

class-m steals, the probability of having fewer than R_m successful steals is at most $\varepsilon / \lceil \lg P \rceil$.

Let Q be the total number of entering steals. Summing over all $\lceil \lg P \rceil$ classes and using a union bound, we know that with probability at least $1 - \varepsilon$, we have

$$\begin{aligned} Q &\leq \sum_{m=1}^{\lceil \lg P \rceil} \left(\frac{PR_m}{2^{m-2}} + \frac{4P}{2^{m-1}} \ln\left(\frac{\lceil \lg P \rceil}{\varepsilon}\right)\right) \\ &< \sum_{j=1}^{P-1} \left(\frac{P \cdot r_j}{2^{\lfloor \lg j \rfloor + 1 - 2}} \right) + 8P \ln\left(\frac{\lceil \lg P \rceil}{\varepsilon}\right) . \end{aligned}$$

Using Fact 5 from Lemma 3.7 and choosing $K = M$ and $f(j) = 1/2^{\lfloor \lg j \rfloor}$ yields

$$Q < 2PM\left(\lg\left(1 + \left\lceil \frac{PN}{M} \right\rceil\right) + 1\right) + 8P \ln\left(\frac{\lceil \lg P \rceil}{\varepsilon}\right) .$$

□

Bounding Completion Time

We can now use the analysis of steal attempts in HELPER to bound the completion time of a computation.

Theorem 3.10. *Let C be a computation executed on P workers, let N be the number of regions in $\mathcal{G}$, and let M be the number of edges in $\mathcal{G}$'s region graph. If the computation DAG $\mathcal{G}(C)$ has work T_1 and aggregate span $\tilde{T}_\infty$, then HELPER completes C in expected time*

$$T_P(C) = O\left(\frac{T_1}{P} + \tilde{T}_\infty + b(C) + M \ln\left(1 + \frac{PN}{M}\right)\right) .$$

Moreover, for any $\varepsilon > 0$, the completion time satisfies

$$T_P(C) = O\left(\frac{T_1}{P} + \tilde{T}_\infty + b(C) + M \ln\left(1 + \frac{PN}{M}\right) + \ln^2 P + \ln\left(\frac{1}{\varepsilon}\right)\right)$$

with probability at least $1 - \varepsilon$.

Proof. On every time step, each of the P processors is working, entering, leaving, waiting, or stealing. Executing a computation requires exactly T_1 work steps. The number of entering steps and leaving steps is at most $O(PN \lg P)$, since every worker waits at most $O(N)$ times (at most once for entering and leaving every region A), and each worker spends only $O(\lg P)$ steps waiting each time.

Also, for a computation $\mathcal{C}$ with bondage $b(\mathcal{C})$ executed on P processors, we can show that the total number of waiting steps (due to serial acquires) is at most $Pb(\mathcal{C})$. When any worker takes a waiting step, there exists a p holding a lock L and executing some node in a critical region protected by a serial acquire. Therefore, the remaining bondage decreases by at least 1 during that time step. In the worst case, all $P-1$ other workers might be taking waiting steps during that time step, all waiting on lock L.

To bound the number of steals in expectation, by Lemma 3.4, we expect $O(P\widetilde{T}_\infty)$ contributing steals, and by Theorem 3.8, we have $O(M\ln(1+PN/M))$ entering steals. Finally, the number of leaving steals is bounded by P times the number of time steps that any worker spends on a leaving step. Thus, we have only $O(PN\lg P)$ leaving steals. The $PN\lg P$ terms are asymptotically absorbed into the $M\ln(1+PN/M)$ term, since $M \geq N-1$.

Since P workers are active on every step, adding up the bounds on all the steps and dividing by P gives us the expectation bound. To obtain a high-probability bound, we choose $\varepsilon' = \varepsilon/2$ for both Lemma 3.4 and Theorem 3.9 and then union-bound over the two cases of contributing and entering steals. □

3.7 Space Bounds

This section presents the space bound provided by HELPER. This section first extends the computation tree model to account for the space used by each function, and then states HELPER's space bound and sketches its proof.

Definitions

To state space bounds for HELPER, we extend some of the definitions from Section A.5. For any function instance F, let $\texttt{frameSize}(F)$ be the stack space used by the function F. On any step t, let $\texttt{rgVFunc}^{(t)}(A)$ denote the set of ***active functions*** within region A, i.e., the active functions F which belong to region A. Then, let $\texttt{rgStackSpace}^{(t)}(A)$ be

$$\texttt{rgStackSpace}^{(t)}(A) = \sum_{F \in \texttt{rgVFunc}^{(t)}(A)} \texttt{frameSize}(F) ,$$

the stack space used by A on step t. Define the ***region stack-space usage*** for A, denoted by $\mathcal{S}_P(A)$, as the maximum over all steps t of $\texttt{rgStackSpace}^{(t)}(\mathcal{C})$, assuming that region A is executed using P processors. Note that $\mathcal{S}_1(A)$ is the serial stack-space usage of region A. Finally, define the ***aggregate serial-space usage*** as

$$\widetilde{\mathcal{S}}_1 = \sum_{A \in \texttt{regions}(\mathcal{C})} \mathcal{S}_1(A) .$$

Space Bound

HELPER's space bound generalizes the space bound for Cilk given in [30].[10]

[10] The bound for Cilk is restated by Theorem A.4 in Appendix A, Section A.5.

Theorem 3.11. *For a computation C, HELPER executes C on P processors using at most $P\widetilde{\mathcal{S}}_1$ stack space.*

Proof. For an arbitrary computation C executed using HELPER, we can generalize the busy-leaves property from Theorem A.3 to apply to the subtree of the active tree that corresponds to a single region A. Thus, for a region A, using the same analysis as in the proof of Theorem A.4, we can show that

$$\texttt{rgStackSpace}^{(t)}(A) \leq \mathcal{S}_P(A) \leq P\mathcal{S}_1(A) .$$

The overall stack-space usage is at most the sum over all regions A of $\texttt{rgStackSpace}^{(t)}(A)$. Therefore, we have the bound on stack-space usage of $P\widetilde{\mathcal{S}}_1$.

□

Intuitively, the stack space required for each worker p roughly corresponds to the function frames for regions along p's deque chain. In general, however, `help_region` calls make it difficult to bound stack-space usage in terms of $\mathcal{S}_1$ for the entire computation C, because a deque q_B for a region B can be a child of deque q_A even though B is not a region nested inside A.

For some computations, a tighter bound than Theorem 3.11 is possible. Technically, we only need to sum over all regions A which can be active on any given time step t. For example, in the hash-table example from Section 3.1, HELPER requires only $O(P\lg n)$ space, because at most two regions can be active at once, namely the outer region and the resize region. Also, each region uses only $O(\lg n)$ space on a single worker.

On the other hand, there exist computations for which the bound in Theorem 3.11 is tight. For example, suppose that the region graph is a chain of regions $A_1, A_2, \ldots A_N$, and all P workers enter every region A_i (i.e., one worker starts region A_{i+1} and has all other workers make `help_region` calls to enter A_{i+1}).

3.8 HELPER Implementation

This section describes HELPER, a prototype implementation of helper locks and parallel regions created by modifying MIT Cilk [51]. In this section, I first discuss how HELPER implements deque chains and deque pools, the two major additions to the Cilk runtime. Then, I describe how HELPER compiles parallel regions.

Deque Chains

To implement parallel regions, HELPER must conceptually maintain a chain of deques for each worker p. In ordinary Cilk, each deque is represented by pointers into a ***shadow stack***, a per-worker stack which stores frames corresponding to Cilk functions. HELPER maintains the entire deque chain for a worker on that worker's shadow stack.

Normally, Cilk uses the THE protocol [51] to manage deques. Each deque consists of three pointers that point to slots in the shadow stack. The ***tail pointer*** T points to the first empty slot in the stack. When a worker pushes and pops frames onto its own deque, it modifies T. The ***head pointer*** H points to the frame at the top of the deque. When other workers steal from this deque, they remove the frame pointed to by H and decrement H. Finally, the ***exception pointer*** E represents the point in the deque above which the worker should not pop. If a worker working on the tail end encounters $E > T$, some exceptional condition has occurred, and control returns to the Cilk runtime system.

In order to avoid the overhead of allocating and deallocating shadow stacks at runtime, HELPER maintains the entire chain of deques for a given worker p on the same shadow stack, as shown in Figure 3-9. Each deque for a given worker p maintains its own THE pointers, but all point into the same shadow stack. When a worker enters a region, it only needs to create the THE pointers for a new deque and set all these pointers equal to the tail pointer for the parent deque in the shadow stack. The correctness of this implementation relies on the property that two deques in the same shadow stack (for a worker p) can not grow and interfere with each other. This property holds for two reasons. First, in HELPER every worker p works locally only on its bottom active deque $p \rightarrow activeDQ$, and thus, only the tail pointer T of the active deque at the bottom of the chain can grow downward. Second, for any deque, the head H never grows upward, since H only changes when steals remove frames. Figure 3-9 shows an example arrangement of a deque chain on a worker p's stack. In this example, no worker has stolen any frames from region B, whereas two frames have been stolen from region C. The deque $dq(p,E)$ is empty, and we have $p \rightarrow activeDQ = dq(p,F)$.x

Implementation of Deque Pools

HELPER implements a deque pool as a single array of deques (i.e., THE pointers). An array of P slots is statically allocated when the user creates a helper lock. When a region is active, one or more slots of this array are occupied by workers. Instead of dedicating a slot for every worker p as described in Section 3.3, however, the implemented prototype maintains a packed array. If k workers are assigned to a region, the first k slots of the deque-pool array contains those workers. It also maintains a shared counter to track whether the pool is empty.

Theoretically, with this packed array implementation, each worker p might spend $\Theta(P)$ time entering and leaving (as opposed to $O(\lg P)$ time with the sparse array described in Section 3.3), if there is worst-case contention and p waits for all other workers. Thus, in theory, the time bound could get slightly worse, while the space bound remains the same. On the other hand, with this packed array implementation, workers might in principle spend less time finding work if on average, the number of processors that enter each region is significantly less than P. For example, in a region with only k processors, if work is available to be stolen, each steal succeeds with probability at least $1/k$, instead of $1/P$.

In practice, we expect that the difference in contention between the two schemes not likely to be significant, particularly for modest values of P. Moreover, even with worst-case contention, using this scheme does not change the worst-case theoretical bounds for the case when $M = PN$. In this case, from the proof of Theorem 3.10, we see that when $M = PN$, the term due to entering steals becomes PN, which is asymptotically the same as having P workers each take P steps to enter and leave every region.

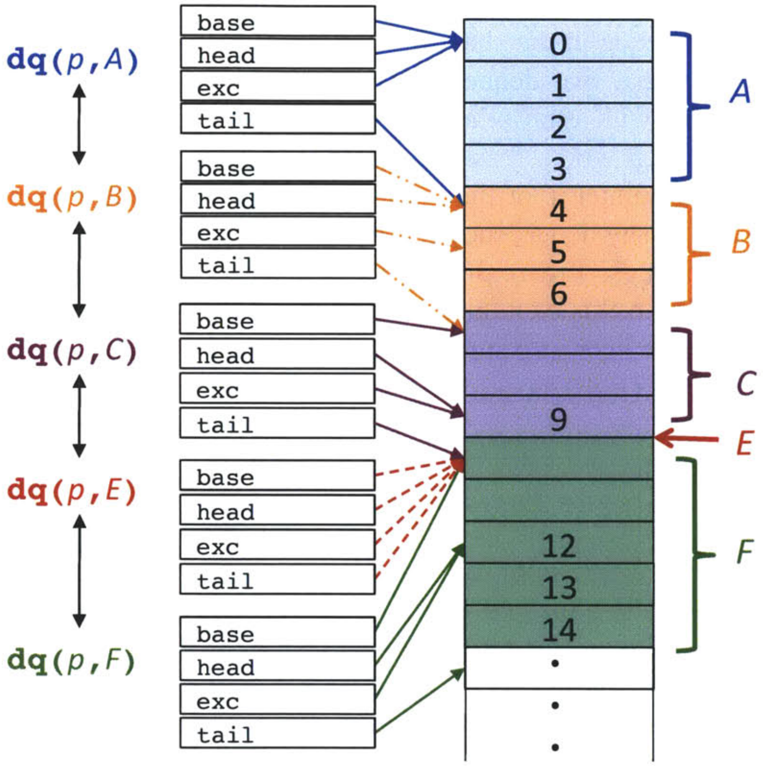


Figure 3-9: A chain of deques for a given worker p. Each deque $dq(p,A)$ consists of THE pointers (tail, head, and exception) into p's stack of Cilk frames. For clarity, the figure shows a pointer for the base of each deque q. In practice, q's base always equals the tail pointer of q's parent deque.

The prototype HELPER implementation uses a simple locking protocol which associates a lock with every deque. A worker must lock a deque before trying to steal from it. When a worker p with active deque $q = p \rightarrow activeDQ$ tries enters another region B, it locks q, creates a new deque q_B with $q_B \rightarrow region = B$ and $q \rightarrow child = q_B$, locks q_B, releases the lock on q, and finally releases the lock on q_B. Similarly, when a worker p leaves a region B, it locks $q = p \rightarrow activeDQ \rightarrow parent$, then locks $q_B = p \rightarrow activeDQ$, deallocates q_B, and finally releases the lock on q.

Compiling Parallel Regions

I implemented a small amount of compiler support in HELPER to enable the calling or spawning of an arbitrary function `F` as a parallel region protected by a helper lock.

The existing MIT Cilk compiler [51] uses source-to-source translation to convert every Cilk function `F` into two C functions, a fast and a slow clone. In ordinary Cilk, the runtime begins executing a spawned function `F` via its fast clone. The slow clone of `F` executes only after a steal from `F` has occurred.

For HELPER, I modified the source-to-source translator to generate an additional clone, the ***region clone*** of `F`, which takes in a helper lock object as an extra argument, and starts `F` as a parallel region. In this prototype, the region clone of `F` is a wrapper function that wraps the arguments of `F` into a generic "closure" object that the Cilk runtime can execute on any processor. Then, the Cilk runtime executes this closure, which eventually calls the fast clone for `F`. As in ordinary Cilk, all spawned functions inside `F` begin as calls to fast-clone functions, with slow-clone functions executing only after steals have occurred.

3.9 Experimental Results

This section presents a small experimental study of HELPER. First, a simple resizable hash table was implemented using HELPER, which demonstrates that HELPER enables users to exploit parallelism inside a critical region. Although neither the hash-table implementation nor the HELPER prototype were heavily optimized, these experimental results suggest that HELPER is not merely a theoretical construct, and that a practical and efficient implementation of HELPER is feasible. Also, the overheads of parallel regions were measured to estimate the impact of HELPER's runtime modifications on existing benchmarks in MIT Cilk. The results indicate that starting each parallel region incurs somewhat significant overhead, but that the modifications required for HELPER produce a negligible impact on existing Cilk code that does not make use of parallel regions.

Resizable Hash Table

This section describes the implementation of a simple hash table microbenchmark, as motivated by the example in Section 3.1.

The resizable hash table maintains an array of hash buckets, with each bucket implemented as a linked list plus a lock. Each element in the linked list stores a 64-bit integer key and a 64-bit integer value. The hash table supports two functions, `search` and an atomic insert_if_absent. Both functions requiring locking the appropriate bucket in the table. If concurrent table resizes are possible, then these functions also must hold the reader-writer lock on the table in reader mode.

The resize operation must acquire the reader-writer lock in writer mode. The table hashes an element x using hash functions of the form $f(x)$ mod n, where $f(x)$ is a pseudorandom function of x and n is the current table size. Thus, rebuilding the table is done by doubling the size of the table and splitting each bucket into two buckets.

The microbenchmark performs n inserts (of the keys 1 through n) into the table in parallel. Inserts were performed in a divide-and-conquer fashion with a serial base case of 1000 inserts. To reduce contention, the benchmark performs a batch of 250 inserts for every acquisition of the reader-writer lock. Also, after every batch, it updates a global counter which tracks the number of inserts into the table. A resize operation is triggered if the number of elements in the table is 5% more than the number of buckets.[11]

Finally, for a reference comparison, an analogous benchmark of n parallel inserts was implemented using the concurrent hash map and task scheduler in Intel TBB [71], version 3.0. Both the HELPER implementation and the TBB implementation use a hash function $f(x)$ that multiplies x by a large 64-bit integer.

Figure 3-10 shows results from performing insertions into the resizable hash table. These results demonstrate that executing the resize operation serially creates a scalability bottleneck. In contrast, HELPER is able to exploit parallelism in the resize operation and increase throughput up to $P = 6$ cores. The HELPER implementation also exhibits throughput for inserts comparable to the reference TBB implementation. This experiment demonstrates that helper locks can improve performance by exploiting parallelism inside critical regions.

Overheads of Parallel Regions

Next, I also compared the overhead of using parallel regions in HELPER to normal Cilk code without regions. In particular, I performed two different experiments on the suite of MIT Cilk benchmarks. First, to measure the overhead of starting parallel regions, I compiled the MIT Cilk benchmarks in HELPER, converting every Cilk function into a parallel region. Second, to determine the effect of HELPER's runtime modifications on MIT Cilk code that does not use regions, I compiled and ran the MIT Cilk benchmarks without modification, but using HELPER instead of MIT Cilk.

Figure 3-11 shows our overhead measurements for HELPER on the MIT Cilk benchmarks. The first experiment, with all Cilk functions converted into parallel regions, shows that there is somewhat significant overhead in our prototype for starting a region as compared to a spawn of an ordinary Cilk function. From the fib benchmark, which repeatedly spawns function calls, we see that starting a region is about 26 times the cost of a spawning a function. This overhead is not unexpected, however, since in the current prototype,

[11] For a more realistic use of a hash table, batching inserts may not be possible. Also, in practice, many applications that use a concurrent hash table spend a majority of time on searches, not inserts. This benchmark is not intended to model a good hash-table implementation for a real application, but rather is designed to trigger as many resize operations as possible to test whether HELPER can effectively exploit parallelism within the resize operation.

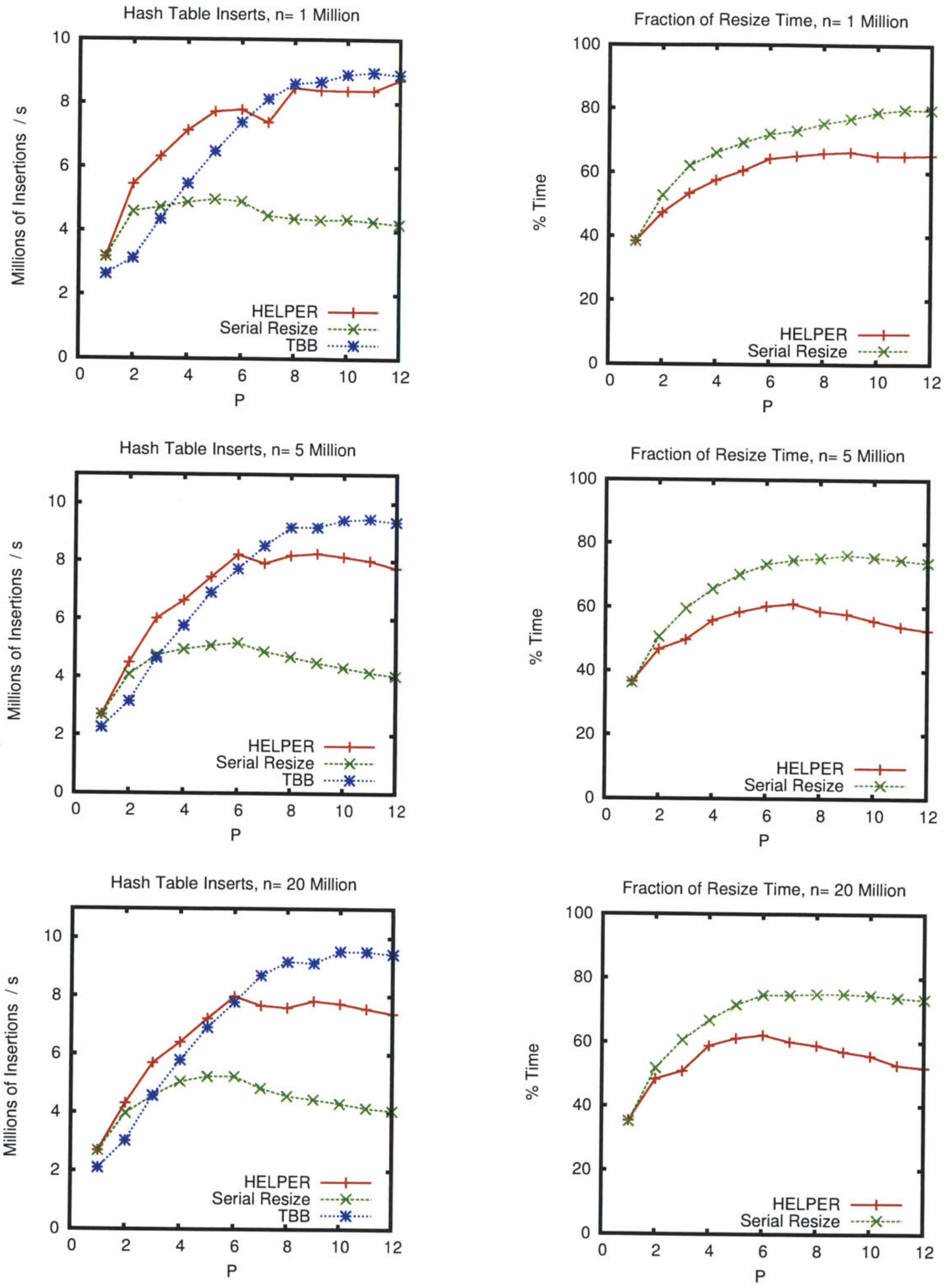


Figure 3-10: Performance of n inserts into a concurrent resizable hash table using HELPER. The plots on the left show the throughput of inserts. The plots on the right show the fraction of time spent on resize operation. Each data point represents the average of 25 runs with the same parameters. This experiment was run on a single-socket 12-core AMD Opteron 6168 processor (clocked at 1.9 GHz) on a machine with 16 GB RAM.

		MIT Cilk	HELPER, All Regions		HELPER, No Regions	
Application	*Description*	Time	Time	Overhead	Time	Overhead
`fib 37`	Fibonacci number calculation	2.63	69.3	26x	2.62	-0.4%
`knapsack`	Knapsack combinatorial optimization	5.22	73.7	14x	5.29	1.3%
`cholesky`	Sparse Cholesky matrix decomposition	2.97	19.2	6.6x	2.93	-1.3%
`heat`	Jacobi-type stencil computation	2.50	3.34	33%	2.52	0.8%
`fft`	Fast Fourier transform	1.68	2.09	24%	1.60	-4.7%
`lu -n 2048`	LU matrix decomposition	7.87	8.59	9%	7.87	0%
`matmul 1000`	Cache-friendly matrix multiplication	2.84	3.06	8%	2.84	0%
`queens 24`	n-queens puzzle	4.60	4.97	8%	4.60	0.2%

Figure 3-11: Serial overhead of parallel regions in HELPER for MIT Cilk benchmarks. The "All Regions" column corresponds to a modification of the MIT Cilk benchmarks where every Cilk function is converted into a parallel region. The "No Regions" column corresponds to unmodified benchmarks compiled with HELPER. All times are in seconds, and represent the average of 10 runs. This experiment was run on a single-socket 12-core AMD Opteron 6168 processor (clocked at 1.9 GHz), on a machine with 16 GB RAM.

starting a parallel region requires acquiring multiple locks, allocating memory, copying arguments into temporary storage, and other initialization for deque pools.

The second experiment, which has no parallel regions, shows that there the modifications required by HELPER have no significant impact on code that does not require parallel regions. In fact, HELPER was slightly faster than MIT Cilk on some of the benchmarks, which suggests that the performance differences between the systems may be dominated by measurement noise. This result is not surprising, since the only extra action that HELPER requires over ordinary MIT Cilk on code without regions is an extra check on a (failed) steal attempt of whether a deque has a child and a nested region to enter. This second benchmark indicates that one can support parallel regions in Cilk with virtually no impact on code that does not care about parallel regions — a significant advantage for a dynamic-threading platform, since we do not want to sacrifice the performance of the common case (programs without regions) for those programs that might want to utilize parallel regions.

3.10 Conclusions

I conclude by briefly reviewing some related work and discussing future research directions.

Related Work

OpenMP uses a `parallel` construct to support nested parallelism [106]. HELPER exhibits some similarities to the implementation of this construct, although the design goals differ. For convenience in implementation, in the HELPER prototype, as in OpenMP, every parallel region has one (worker) thread which is the first to enter the region and which is guaranteed to resume execution after the region completes. Also, unlike in OpenMP, HELPER executes parallel regions ***asynchronously*** — the number of workers is not fixed when a region begins, and additional workers can enter the region after it has started, either through random work stealing or because they are blocked on the lock for the region. Also, the `critical` construct in OpenMP, which provides mutual exclusion for critical sections, is generally not designed to allow nested `parallel` constructs.

The idea of parallelism inside critical sections was explored by Kessler and Seidl in Fork95 [79], a parallel language for PRAM machines. Like OpenMP, however, they adopt a synchronous approach. To start a parallel region, programmers invoke an "join" construct which waits a specified time (or until some other condition is satisfied) before starting execution using however many threads have joined the region.

Microsoft TPL [92] provides a "replicable task" API which allows users to implement their own parallel abstractions. In principle, a programmer could use this API to construct parallel regions as in HELPER by creating a replicable task which behaves as a worker thread for a region. Multiple threads executing the replicable task would correspond to multiple workers executing in the parallel region. To exploit asynchronous task parallelism within the region, however, the programmer would need to implement their own work-stealing task scheduler.

Database systems often exploit nested parallelism inside critical sections protected by transactions [56]. More recently, new dynamic-threading languages such as Fortress [13] and X10 [37] also provide an `atomic` (or similar) construct which allows users to specify blocks of code as transactions. Also, the specification for these languages allows users to `spawn` and expose task parallelism inside transactions. Transactions with nested parallelism are generally more complicated to support than locked critical sections because the runtime needs to detection conflicts between transactions and rollback transactions when they abort. In Chapter 6, I discuss support for transactions with nested parallelism in Cilk.

Cooperative techniques, where one thread helps another thread complete its work, have previously been proposed in a variety of contexts. In the context of nonblocking algorithms, researchers [22, 75, 118] describe algorithms where threads cooperate to complete an operation when they would otherwise block for synchronization. In the area of databases, Lim, Ahn, and Kim [94] describe a concurrent B^{link} tree algorithm which uses cooperative locking to handle nodes with concurrent underflow.

Directions for Future Research

One interesting direction for future research is to try to develop more efficient runtime and compiler support for helper locks. The HELPER prototype described in this chapter is not heavily optimized. In particular, executing a region requires multiple lock acquisitions to add and remove workers from the deque pool for each region, even in the case when no other workers block or steal into the region. One way to improve performance would be to try to optimize the runtime for the case when only a single worker executes the region. Also, one might add compiler support to generate more efficient code for starting and finishing each parallel region which is specialized to the region being executed.

Another avenue for future work is to implement a more sophisticated hash table using helper locks for resizing and to compare it to other optimized concurrent hash-table implementations. Most concurrent hash tables (e.g., hopscotch hashing [67]) are optimized for the case when most operations are queries. It would be interesting to explore whether one can efficiently parallelize resize operations using helper locks in phases of a program where updates are frequent without affecting the performance in the phases where queries dominate.

It would also be interesting to identify other applications which might benefit from helper locks and benchmark these applications on an improved prototype of HELPER. For example, one might conceivably use HELPER locks to support computations written using futures [57,58].

Finally, one may be able to find other uses for HELPER's parallel region construct besides helper locks. For example, one might use parallel regions to try to optimize for locality in a region, i.e., by limiting the worker threads allowed to execute a region to the set of workers which are executing on processors that share a cache. In Chapter 4, I describe another use case for parallel regions, namely enabling legacy-C functions to call back to parallel Cilk code.

Chapter 4

Parallel Regions for Legacy Callbacks

In dynamic-threading platforms such as MIT Cilk [51] and Cilk++ [93], it can be difficult to efficiently compose parallel dthreaded code with legacy code that make use of callback functions. For example, MIT Cilk prohibits a C function from calling back to a Cilk function, which prevents users from composing a parallel function inside a callback function for a legacy library. More generally, MIT Cilk and Cilk++ compile parallel functions using a special linkage that is incompatible with the linkage for ordinary legacy-C code. Thus, these platforms lack the property of ***serial-parallel reciprocity*** (or ***SP-reciprocity*** for short) [91], the ability to support arbitrary calling between parallel and serial code, including legacy serial binaries.

To address this issue, dynamic-threading platforms such as Intel TBB [110] often support SP-reciprocity by using a modified work-stealing scheduler that differs from traditional Cilk-like work-stealing. More specifically, TBB uses a ***restricted work-stealing*** scheduler, one that prevents a worker from stealing certain ready tasks, to avoid using excessive stack space. Although these schedulers often work acceptably in practice, they generally are unable to guarantee worst-case bounds on time or space usage which are as strong as the bounds for Cilk. Moreover, the theoretical implications of restricted work-stealing have generally not been analyzed and documented in the literature.

Contributions

In this chapter, I investigate the use of restricted work-stealing schedulers in dynamic-threading platforms to support SP-reciprocity. In particular, I discuss the following contributions:

- I show that a dynamic-threading platform that uses ***subtree-restricted work-stealing*** can support SP-reciprocity, achieve the same theoretical space bound as Cilk, and still guarantee a nontrivial upper bound on completion time.
- I describe a worst-case computation that provides a lower bound on completion time for restricted work-stealing. For this worst-case computation, a scheduler using restricted work-stealing can only achieve a constant factor speedup using P processors, whereas a platform using ordinary (unrestricted) work-stealing can achieve linear speedup. This lower bound applies to subtree-restricted work-stealing, as well as ***depth-restricted work-stealing***, the restricted work-stealing scheduler used by TBB.

To show the feasibility of subtree-restricted work-stealing, I describe PR-Cilk, a design of a dynamic-threading platform that uses HELPER's parallel region construct (described in Chapter 3) to enable legacy callbacks from C code back to MIT Cilk code.[1] PR-Cilk demonstrates that a platform can use a single mechanism to support both helper locks and legacy callbacks.

PR-Cilk also offers theoretical bounds on space and time usage. PR-Cilk provides the same guarantee on stack-space usage as MIT Cilk. It also guarantees a completion-time bound which is analogous to the bound for HELPER when callbacks from C to Cilk functions generate parallel regions. This bound is worse than the bound for MIT Cilk. If the number of callbacks from C to Cilk is small, however, or if each of these Cilk functions called by C functions has sufficient parallelism, PR-Cilk still guarantees linear speedup.

Chapter Outline

This chapter is organized as follows. Section 4.1 discusses the challenges in supporting SP-reciprocity in Cilk-like dynamic-threading platforms. Section 4.2 describes PR-Cilk, the design of a dynamic-threading platform that supports SP-reciprocity using subtree-restricted work-stealing. Section 4.3 presents the provable bounds on stack space usage and completion time guaranteed by PR-Cilk. Section 4.4 presents a worst-case lower-bound computation for completion time using a subtree-restricted work-stealing or depth-restricted work-stealing scheduler. Finally, Section 4.5 concludes with a discussion of ideas for future work.

4.1 Difficulty of SP-Reciprocity

This section discusses the challenge of supporting SP-reciprocity in dynamic-threading platforms and describes of the approaches adopted taken by various dynamic-threading platforms to address this problem. In particular, this section focuses on the approach adopted by TBB, which uses the heuristic of "depth-restricted work-stealing" to provide SP-reciprocity while still bounding stack-space usage.

An Example of Legacy Callbacks

In a program written for a dynamic-threading platform supporting SP-reciprocity, an ordinary serial function `foo` is able to seamlessly call a parallel function `A`, even though `A` is written and compiled for the platform and `foo` is written and compiled expecting `A` to be a serial function. Without SP-reciprocity, it can be hard to integrate a parallel library into existing code.

As a concrete example, consider the parallelization of a stencil-computation visualization using Cilk++ and OpenGL, a demo from a recent class at MIT on multicore computing [1]. This interactive demo repeatedly evaluates a stencil computation for a 2-d heat equation on an $n \times n$ grid, updating a window displaying the grid periodically and capturing mouse inputs from the user to introduce heat sources into the grid. In the original serial

[1] PR-Cilk represents joint work [6] with Kunal Agrawal and I-Ting Angelina Lee.

code for the visualization, a draw method calls a compute method to perform the stencil computation, and then it displays the result using OpenGL. In the main program, this draw method is registered as a callback method with OpenGL, which handles the visualization.

The lack of SP-reciprocity in Cilk++ complicates the parallelization of this demo. Ideally one would like parallelize the compute method using Cilk++. Since draw is a callback, and OpenGL is binary compiled using an traditional compiler (e.g., GCC), however, draw must use a standard C linkage and cannot be a ***Cilk function*** — a function that can contain spawn and sync statements. Thus, the draw method cannot directly call the compute method, which must be compiled using special Cilk linkage. MIT Cilk [51] and Cilk++ [93] require special linkage because they use "shadow frames" to support the "cactus-stack" abstraction. For more details, see Section A.1.

In some cases, one can work around the lack of SP-reciprocity in MIT Cilk and Cilk++ by creating and managing Cilk contexts whenever a C function needs to call back to Cilk. The stencil demo was parallelized using a global Cilk context object, which contains some number P of associated worker threads. Inside the draw method, the code uses the context object to run the Cilk method compute using the P threads. This workaround was sufficient because the stencil computation was the only section of code being parallelized, and P for the context could be set to the number of cores on the target machine.

Explicit management of Cilk contexts is not composable, however. If one had other code or other callbacks running in parallel with compute, then the programmer would need to create separate contexts for each. If the programmer allocates P threads to each of many parallel contexts, then he or she will oversubscribe the machine, thereby degrading performance with respect to both time and space. Conceptually, each context can be thought of as a separate instance of the dynamic-threading platform. Unfortunately, there can often be significant overhead for creating new contexts. Also, a programmer must carefully manage context sizes in order to obtain good performance. Thus, this solution is difficult to apply to applications that have multiple callbacks from C back to Cilk that can execute in parallel.

Alternatively, one can use special operating-system support to implement a cactus-stack abstraction, as in the Cilk-M system described by Lee et al. in [91]. Cilk-M system uses operating-system support for thread-local memory mapping to allow each worker to have its own local view for memory addresses that correspond to the ordinary linear stack. In a function A that spawns a function B, when the continuation of B (in A) is stolen by a thief worker p in Cilk-M, p maintains the illusion of building off the same stack for A by memory-mapping an appropriate range of stack addresses to a new page in memory. Unfortunately, this memory map must be thread-local, whereas traditional memory-mapping system calls only create process-local mappings. Thus, using the Cilk-M runtime system also requires using a special patched kernel.

SP-Reciprocity Using Linear Stacks

Since neither explicit context management nor Cilk-M are ideal for large-scale commercial applications, existing dynamic-threading platforms[2] often provide support for SP-

[2] Platforms that execute in virtualized environments, such as Fortress [13], Java Fork/Join Framework [90], X10 [37] and TPL [92], do not typically have issues with callbacks from legacy binaries.

reciprocity by using linear stacks to simulate the cactus-stack abstraction. When one implements the cactus-stack abstraction using ordinary linear stacks however, one generally faces a tradeoff between the number of stacks needed, the space that may be potentially be wasted on each stack, and the time bound that the platform can guarantee. As a concrete example, suppose that a worker p is executing a Cilk function A using a stack K, and suppose that A has multiple extant children. The other workers executing these extant children may share a single view of A's frame sitting on a linear stack of worker p. Once this frame for A has been allocated, its location in virtual memory cannot be changed, because there may be a pointer to a variable in the frame elsewhere in the system.[3] Thus, even if p has finished its child of A, since A's frame is shared, p cannot free the stack K where A resides until *all* of A's extant children return. Thus, after p finishes its child of A, it generally has three options:

1. Steal another frame B, but execute B reusing space on K (by putting the frame for B below the frame for A).[4]
2. Steal another frame B, and execute B using a new stack K′.
3. Limit the frames that p is allowed to steal until all of the extant children of A are finished.

Each of these approaches faces several obstacles.

In the first option, each worker p tries to reuse space on its stack K when stalling at a sync. This option can potentially waste stack space, because once A (and possibly its ancestors on K) finish, their space on K cannot be reused if B is still active. If A is already deep in the stack and the stolen frame B is close to the top of the ***invocation tree***, i.e., the tree of function calls and spawns, then the stack K can grow twice as deep as what it would be in a serial execution.[5] Furthermore, this scenario, which increases stack-space usage, could occur repeatedly every time p steals. Thus, this option could theoretically waste significant stack space.

The second option requires switching stacks whenever p returns and stalls at a sync. With this option, the number of stack switches can generally be proportional to the number of successful steal attempts in the computation. Also, the number of stacks that may be "active" at a time in a worst-case computation is at least $\Omega(Pd)$ stacks, where d is the spawn depth of the computation. In the worst case, two workers might keep stalling at a sync and stealing from each other at progressively deeper levels in the tree, causing a stack switch at every spawn depth. More precisely, if a worker p is working on a stack frame F, then in the worst case, each function F' which is an ancestor of F might have required a stack switch. Switching between multiple stacks can also waste space because the runtime must reserve enough virtual-address space for each stack to grow without any overflow when nested function calls occur.

Intel Cilk Plus [73] utilizes this second option, implementing the cactus-stack abstraction by using a pool of ordinary linear stacks. In this system, for a function A that spawns a function B, when the continuation of B (in A) is stolen, the thief worker resumes A on a

[3] This constraint applies more generally to any strategy that allocates frames in shared virtual-address space.

[4] We assume that the stack grows downward.

[5] Using the notation described in Section A.5, the space required can increase to $2\mathcal{S}_1$.

new stack, taken from the pool of stacks. When a worker p resumes the continuation of a `sync` statement, it may also need to switch linear stacks if there was a successful steal in that sync block.[6]

Finally, the third option, limiting the ability of p to steal, avoids the problem of excessive stack-space usage, but at the cost of sacrificing the provable bounds on completion time that randomized work-stealing provides. As reviewed in Section A.3, Cilk-like work-stealing schedulers provide provable bounds on completion time (i.e., Theorem A.1), because whenever a worker p steals, p is likely to steal a task whose execution is likely to reduce the span (critical path) of the computation. When p cannot steal arbitrarily, Theorem A.1 no longer holds true. In the remainder of this section, I discuss two approaches to supporting SP-reciprocity which are based on this third option, namely depth-restricted work-stealing as implemented by TBB, and subtree-restricted work-stealing, which is used by the PR-Cilk design.

Depth-Restricted Work-Stealing in TBB

Intel Threading Building Blocks (TBB) [110] is a dynamic-threading platform that enables programmers to exploit task-based parallelism as a library. Because TBB is designed as a library rather than a language, programs written using TBB can be compiled with an ordinary (serial) C++ compiler. Thus, TBB automatically provides programmers with SP-reciprocity.

Unfortunately, because TBB has no special compiler support, it cannot automatically generate function continuations. Continuations allow a function to begin execution on one processor, but resume execution on another processor (e.g., after a steal occurs). A library such as TBB typically can utilize continuations only when they are explicitly provided by the programmer. Programming with explicit continuations can be tricky because the cactus-stack abstraction is unavailable. Said differently, if a task wants to initialize a variable `x` and pass the address of `x` down to a child subtask, because there is no stack to allocate `x` from, `x` must be allocated from the heap.

Thus, libraries such as TBB usually support additional, more convenient interfaces which do not require coding explicit continuations. For example, TBB allows programmers to write ***blocking-style*** code, as shown in Figure 4-1. In this code, `BTask` is a task which computes two values, x and y (using two parallel subtasks, `XTask` and `YTask`), and then computes the sum $x+y$. To understand the behavior of blocking-style code, suppose that worker p_1 begins executing `BTask`. In line 9, p_1 spawns `YTask` for other workers to potentially steal and then starts working on `XTask` in line 10. Suppose that another worker p_2 steals `YTask` from p_1, and then p_1 finishes `XTask` before p_2 completes `YTask`. Then p_1 stalls at line 10, and p_1 begins trying to randomly steal work. If `BTask` was coded by the programmer using an explicit continuation, then p_1 could then clear the stack space used by `BTask`, since p_2 could resume the continuation of `BTask` after p_2 completes `YTask`. With

[6] Conceptually, one could think of MIT Cilk and Cilk++ as a degenerate case of using multiple linear stacks, with each linear stack only containing a single frame. Because each stack contains only a single frame, instead of storing that frame on a linear stack that can grow downward, the Cilk compiler optimizes the linear stack into a shadow frame (which is allocated from heap memory), at the cost of requiring a special calling convention.

```
 1 class BTask: public task {
 2   int* sum;
 3   int x, y;
 4   BTask(int* sum_) : sum(sum_)
 5   task* execute() {
 6     XTask& t_x;  YTask& t_y;
 7     t_x = *new(allocate_child()) XTask(&x);
 8     t_y = *new(allocate_child()) YTask(&y);
 9     spawn(t_y);
10     spawn_and_wait_for_all(t_x);
11     *sum = x + y;
12   }
13 }
```

Figure 4-1: A blocking-style computation using TBB pseudocode. Code for reference counting tasks has been omitted.

blocking-style code, however, p_1 cannot clear this stack space, since p_1 must eventually resume BTask at line 11.

Blocking-style code is convenient because it can support the cactus-stack abstraction. In blocking-style code, the programmer can allocate local variables on the stack of the execute method and pass pointers to these variables to XTask and YTask. Using explicit continuations, however, the programmer must declare these variables elsewhere (e.g., as member variables of BTask), because control can return from the execute function before both children complete.

To avoid a significant growth in stack space due to a worker repeatedly blocking and stealing, TBB uses ***depth-restricted work-stealing*** [115], that is, TBB constrains a worker to only steal tasks which are deeper than the worker's deepest blocked task. As the TBB documentation states, this restriction on work-stealing may limit the available parallelism and impact performance [110].

How restrictive is depth-restricted work-stealing, as compared to unrestricted work-stealing? Unfortunately, it seems difficult to prove a nontrivial upper bound for depth-restricted work-stealing. The fact that a thief can steal from arbitrary part of the invocation tree (as long as the depth restriction is not violated) greatly complicates a theoretical analysis. More precisely, with depth-restricted work-stealing, even if a frame B is below A in a worker's stack, A need not be an ancestor of B in the invocation tree. Thus, unlike in Cilk or HELPER, it is not obvious how to count a steal as making progress on any "span" of the computation.

The PR-Cilk design explores the notion of "subtree-restricted work-stealing," a stronger version of depth-restricted work-stealing that is more amenable to a worst-case theoretical analysis. As I discuss in Sections 4.2 and 4.3, HELPER's parallel region construct can be used to implement callbacks from C to Cilk, thereby providing support for SP-reciprocity.

Section 4.4 also explores the limits of restricted work-stealing. In particular, I construct a computation which, when executed using depth-restricted work-stealing or subtree-restricted work-stealing on P processors, runs $\Omega(P)$ times slower than when executed using unrestricted work-stealing. Thus, there exists a computation which could exhibit linear speedup when run on *P* processors, but which is asymptotically serialized by restricted work-stealing.

4.2 PR-Cilk Design

PR-Cilk supports SP-reciprocity and provable space and time bounds by using a strategy called subtree-restricted work-stealing. Instead of exclusively using shadow frames or activation frames, PR-Cilk uses shadow frames of Cilk functions and the regular C activation frames for C functions. At the high-level, PR-Cilk must add three functionalities to Cilk. First, it must implement subtree-restricted work-stealing. Second, it must modify the runtime system to implement the call semantics for invoking a Cilk function, which differs from the spawn semantics. Finally, it must modify the compiler to perform a linkage transition automatically when a C function calls a Cilk function. This section describes the design of these functionalities.

Subtree-Restricted Work-Stealing Using Parallel Regions

Like depth-restricted work-stealing, subtree-restricted work-stealing maintains the same space bound as Cilk by restricting the frames that some processors can steal. Restricted work-stealing is closely related to the idea of ***leapfrogging***, a technique for implementing futures described in [123]. Subtree-restricted work-stealing is in fact more restrictive than TBB's depth-restricted work-stealing, but the stronger restriction enables us to prove a stronger completion-time bound, because a stronger restriction eliminates some bad schedules that the weaker restriction can allow.

To recall the problem, suppose that a worker *p* executes a C function `foo` which calls a Cilk function `B`. Since the C function uses the activation frame, the stack space associated with `foo` cannot be removed from the *p*'s stack until all the descendants of `B` in the invocation tree are completed. If *p* runs out of the work before all children of `B` finish, then, as mentioned earlier, it must either block and wait for its extant children to complete, thereby sacrificing the time bound, or steal, potentially consuming excessive space if it steals a frame close to the top of the invocation tree.

Subtree-restricted work-stealing solves the problem by forcing *p* to steal from only within `B`'s subtree in the invocation tree. Conceptually, we refer to `B`'s subtree as a ***callback region***, i.e., it is a subcomputation (potentially with nested parallelism) whose root is a Cilk function called by a C function. In any serial execution, the stack depth of any frame within `B`'s subtree is greater than the stack depth of `B`, where *p* is stalled. Therefore, with subtree-restricted work-stealing, no processor can use more stack space than the serial execution, and we maintain the Cilk stack-space bound. Furthermore, any work *p* steals is work that must be completed in order for `B` to return, and *p* is (in some sense) helping to complete its own work.

PR-Cilk implements subtree-restricted work-stealing for callback regions using the parallel region construct described earlier in Chapter 3. Normally, a worker in Cilk is only allowed to steal from the top of a deque. The parallel region construct in HELPER provides, however, a mechanism for limiting work-stealing to a particular region, namely a subtree of the invocation tree.

PR-Cilk implements each callback region as a parallel region, and thus automatically uses subtree-restricted work-stealing. When worker p calls a Cilk function B from a C function foo, it implicitly invokes start_region (the compiler and the runtime support for this is described later). The start_region call causes p to start a new region, which involves p creating a new deque pool $B.dqpool$ and creating a new deque q for itself in $B.dqpool$. After creating the region, p continues to execute B, which may spawn more functions under region B, and the frames associated with these functions are added to the q. Other workers may later be assigned to this region and steal from q. Any additional work created by these workers within B is added to some deque in $B.dqpool$ as well. If p later stalls on a sync in B, it can now steal work from any deque in the pool $B.dqpool$, since such work belongs to the subtree rooted at B. Since PR-Cilk regions are not associated with locks, PR-Cilk does not need to use the help_region construct from HELPER.

Compiler and Runtime Support for SP-Reciprocity

SP-reciprocity requires support for four kinds of invocations involving Cilk functions: (1) a Cilk function can spawn another Cilk function, (2) a Cilk function can call a C function, (3) a Cilk function can call another Cilk function, and (4) a C function can call a Cilk function. MIT Cilk supports the first and the second functionality but not the third and fourth. In order to support the third type of invocation, allowing one Cilk function to call another, one can modify Cilk runtime system to store ***stacklets***, i.e., sequences of frames in a deque, corresponding to Cilk functions connected by call relations, instead of single frames in deques. This approach resembles the technique used in both the Cilk++ and the Cilk-M runtime systems (for details, see the description in [49]).

The most interesting case for PR-Cilk is the fourth kind of invocation, allowing a C function to call a Cilk function. PR-Cilk must modify Cilk to handle two issues. First, since all Cilk functions still use the Cilk linkage (and shadow frames), the runtime must transition between two different kinds of linkages when a C function calls a Cilk function. Second, the compiler must generate code for automatically creating a parallel region whenever a C function calls a Cilk function.

The PR-Cilk compiler generates a special "callable clone" and a "parallel clone" for each Cilk function B, where the callable clone is a C function and the parallel clone is the Cilk function. Conceptually, the ***callable clone*** is a wrapper function for B that invokes the parallel clone of B as a region. Since the callable clone uses the ordinary C linkage (i.e., uses a linear stack), an ordinary C function foo can call the callable clone without recompilation, as long as the callable clone has the same function header expected by foo. In the callable clone, a start_region call is made to create a parallel region. The ***parallel clone***, on the other hand, is essentially equivalent to the user-defined Cilk function B that can be spawned, but is renamed to avoid a naming conflict. The compiler can perform this renaming for the parallel clone because the parallel clone is always spawned and the Cilk compiler compiles all the spawn statements. A parallel region is *not* created when another Cilk function A spawns B. In a chain of nested calls, the number of regions created is exactly the number of times a C function calls a Cilk function.

```
cilk void A () {
  foo(1, 2);
  spawn E();
  sync;
}

cilk void B (int* x, int* y) {
  spawn C(x);
  spawn D(y);
  sync;
}

void foo(int x, int y) {
  B(&x, &y);
}
```

Figure 4-2: Cilk code illustrating the example where A calls foo and then a C function foo calls back to a Cilk function B. In this example, Cilk functions are explicitly labeled with the cilk keyword as a type qualifier.

```
1 void B(int* x, int* y) {
2   start_region B_par(x, y);
3 }

4 cilk void B_par(int* x, int* y) {
5   spawn C_par(x);
6   spawn D_par(y);
7   sync;
8 }
```

Figure 4-3: The callable and parallel clones for B conceptually generated by PR-Cilk. This figure shows valid Cilk code for each clone. In practice, the compiler also translates this Cilk code (along with the rest of the program) into C code with calls into the Cilk runtime.

To illustrate the use of the callable and parallel clones more concretely, consider the Cilk function B from user code shown in Figure 4-2. Figure 4-3 shows the corresponding clones that PR-Cilk conceptually generates for B. The callable clone, which is shown in lines 1–3 of Figure 4-3, has the same header that foo uses to call B. The parallel clone, which is shown in lines 4–8, is essentially the original Cilk function B, except that it has been renamed to B_par. The spawn statements are also compiled accordingly, as shown in lines 5–6 of Figure 4-3. In the callable clone B, the parallel clone B_par is invoked as a parallel region in line 2.

PR-Cilk uses the Cilk linkage for all Cilk functions that are spawned, but it uses the C linkage for C functions. The transition between different linkages occurs in the callable clone. That means, however, that a worker *p* which executes a callable clone must be the worker that finishes the region and resumes back to foo. PR-Cilk imposes this restriction on foo because the activation frame for foo is still sitting on *p*'s linear stack, and the activation frame is the only way to resume foo since foo has no shadow frame.

4.3 Bounds for PR-Cilk

Since PR-Cilk uses the same parallel region mechanism as HELPER [10], the stack-space and completion-time bounds for HELPER in Section 3.6 can be simplified and applied directly to PR-Cilk. This section presents the stack-space and completion-time bounds for PR-Cilk. PR-Cilk guarantees the same stack-space bound as ordinary Cilk, namely that an execution on P processors uses only P times as much space as a serial execution. It also guarantees linear speedup for programs where each callback region has sufficient parallelism.

Space Bounds

PR-Cilk's bound on stack-space usage is an extension of the Cilk's space bound [30].

Theorem 4.1. *For a computation $\mathcal{C}$, let $\mathcal{S}_1$ be the stack space requiredfor a serial execution of $\mathcal{C}$. PR-Cilk executes $\mathcal{C}$ on P workers using at most $O(P\mathcal{S}_1)$ stack space.*

Proof. In a sequential execution, the stack space used while executing a particular function A is the space occupied by the frames corresponding to all of A's ancestors in the invocation tree. Therefore, the total stack space $\mathcal{S}_1$ the maximum over all the root to leaf paths, the sum of the stack space used by all the frames on the path. PR-Cilk, like ordinary Cilk, follows the busy leaves property, which states that all the frames on a processors stack fall on a single root to leaf path in the invocation tree. Since there are P processors, the total space used by PR-Cilk is at most $P\mathcal{S}_1$. Note that in ordinary Cilk, consecutive frames on a worker's stack have a parent-child relation, but in PR-Cilk, frames may have only an ancestor-descendant relation. Intuitively, skipping over ancestors when walking up a single worker's stack only helps the worst-case bound, and thus in PR-Cilk the frames on a single worker's stack uses less than $\mathcal{S}_1$ space. □

Theorem 4.1 provides a stronger bound on stack space than directly applying Theorem 3.11 for HELPER. HELPER has a weaker bound than PR-Cilk because in HELPER, a worker in region A can make an explicit call to help into a region B which is not nested in A. Thus, in the worst case for HELPER, a single worker could have stack frames from every region.

Completion-Time Bounds

To bound completion time, we apply the existing time bound for HELPER (Theorem 3.10 in Chapter 3) directly. We assume that the computation executes on hardware with P processors, with one worker assigned to each processor.

Theorem 4.2. *Let T_P be the running time of a computation $\mathcal{C}$ using PR-Cilk running on P processors. A computation $\mathcal{C}$ with N callback regions, work T_1, and aggregate span*

$\widetilde{T}_\infty$ runs in expected time $\mathrm{E}[T_P] = O\left(T_1/P + \widetilde{T}_\infty + N \ln P\right)$. *Moreover, for any* $\varepsilon > 0$, *with probability at least* $1-\varepsilon$, *the execution time is* $T_P = O(T_1/P + \widetilde{T}_\infty + N \ln P + \lg(1/\varepsilon))$.

Proof. An execution of a PR-Cilk computation is equivalent to a HELPER computation with no `help_region` calls, assuming PR-Cilk uses the same entering policy as HELPER. Thus, we can apply Theorem 3.10 with $M = N-1$. □

In general, Theorem 4.2 is worse than the bound for Cilk, because $\widetilde{T}_\infty$ can potentially be larger than T_∞. Section 4.4 presents a pessimal computation for both depth-restricted work-stealing and subtree-restricted work-stealing that illustrates the adverse effect that restricting work-stealing might have. On this example, PR-Cilk can asymptotically serialize a program while Cilk achieves linear speedup.

Using Theorem 4.2, however, one can derive Corollary 4.3, which gives a sufficient condition for PR-Cilk to execute a program with linear speedup.

Corollary 4.3. *Let T_P be the running time of a computation C using PR-Cilk running on P processors. Suppose that every region $A \in \texttt{regions}(C)$ satisfies $\tau_1(A)/(\tau_\infty(A) + \ln P) > P$. Then $E[T_P] = O(T_1/P)$, i.e., PR-Cilk achieves linear speedup.*

Proof. One can rewrite the completion-time bound in Theorem 4.2 in an alternative form, i.e.,

$$\mathrm{E}[T_P] = O\left(\sum_{A \in \texttt{regions}(C)} \left(\frac{\tau_1(A)}{P} + \tau_\infty(A) + \ln P\right)\right). \tag{4.1}$$

If for each region A, we have $\tau_1(A)/(\tau_\infty(A) + \ln P) > P$, then the quantity $\tau_\infty(A) + \ln P$ is asymptotically dominated by $\tau_1(A)/P$ term. □

Corollary 4.3 applies roughly when every region A has sufficient parallelism. This condition is, of course, not a necessary one for achieving linear speedup. One can derive stronger conditions by considering combinations of regions instead of considering each region separately. For example, one may be able to amortize the time spent in small regions with insufficient parallelism against the time spent in a large region A which does have sufficient parallelism.

4.4 A Pessimal Example for Restricted Work-Stealing

In this section, I present a parallel computation which exhibits linear speedup on P processors when executed by a runtime with an ordinary, unrestricted work-stealing scheduler, but which achieves only constant speedup when the runtime uses depth-restricted work-stealing. This computation can also be modified to apply to subtree-restricted work-stealing and PR-Cilk. First, I outline the general structure of the lower-bound computation. Then, I analyze the runtime of the computation using both depth-restricted and unrestricted work-stealing. This lower-bound computation demonstrates an asymptotic difference in performance between restricted and unrestricted work-stealing and suggests that restricting work-stealing can adversely impact application performance in practice.

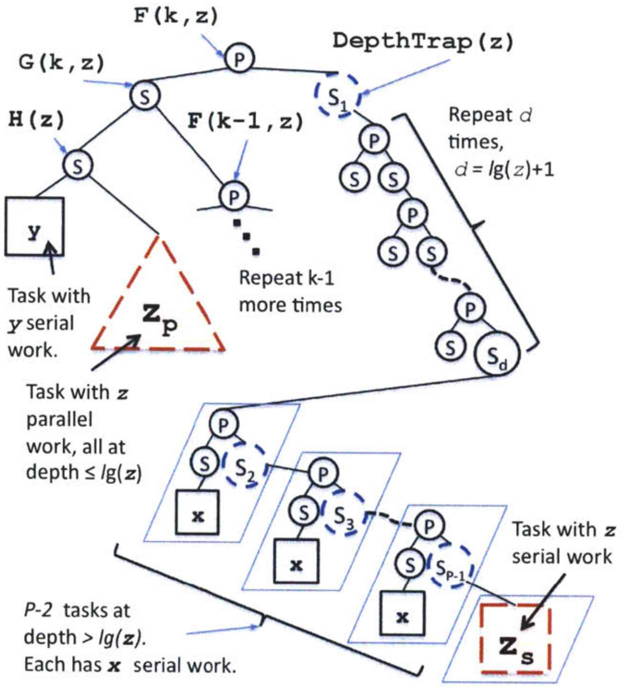


Figure 4-4: A series-parallel parse tree for the F, a lower-bound computation for restricted work-stealing. Squares represent serial tasks, labeled with the work of each task.

The pessimal example is generated by a method $\texttt{F}(k,z)$, which conceptually chains together k instances of $\texttt{F}(1,z)$. I describe this example using a series-parallel tree [45] representation, as reviewed in Section A.4. Figure 4-4 shows the a series-parallel parse tree representation of F. A parallel traversal of a series-parallel parse tree models an execution of a computation on multiple processors. The child subtrees of an S-node must be traversed serially, from left to right, while the child subtrees of a P-node can be traversed in any order. In this tree, a P-node corresponds to a blocking spawn. When a processor reaches a P-node, it begins work on the left child subtree. If the right subtree is stolen and the processor finishes the left subtree, then the processor is blocked on the (S-node) root of the right subtree. I assume that the depth of a task is measured as the depth of P-nodes (i.e., nesting depth of spawns) in the tree.

The subroutine $\texttt{F}(1,z)$ forms the core of the example. When executed on P processors using depth-restricted work-stealing, $\texttt{F}(1,z)$ runs for at least z time, but completes only about $2z$ work. Intuitively, F spawns two tasks: one task G contains z potentially parallel work (subtask $\texttt{z}_p$), and the other task DepthTrap contains z serial work (subtask $\texttt{z}_s$). Ideally, $P-1$ processors should work on G and one should work on DepthTrap. Instead, DepthTrap begins with enough parallel work ($P-2$ tasks with serial work x), and G begins with enough serial work (y) so that $P-1$ processors steal work from DepthTrap and only one works on G. Once $P-1$ processors steal from DepthTrap, $P-2$ processors block waiting for one processor to complete z serial work. Furthermore, since DepthTrap traps these $P-2$ processors at a depth greater than the depth of any work in $\texttt{z}_p$, the processors

```
void F(int k, int z) {
  spawn(DepthTrap(lg(z), z));
  spawn_and_wait_for_all(G(k, z));
}
void G(int k, int z) {
  H(z); if (k > 1) { F(k-1, z); }
}
void H(int z) {
  spawn(ParallelWork(z));
  spawn_and_wait_for_all(SerialWork(y));
}
void DepthTrap(int d, int z) {
  if (d == 0) { TrapProcessors(P-1, z); }
  else {
    spawn(DepthTrap(d-1, z));
    spawn_and_wait_for_all(SerialWork(1));
  }
}
void TrapProcessors(int i, int z) {
  if (i <= 1) { SerialWork(z); }
  else {
    spawn(TrapProcessors(i-1, z));
    spawn_and_wait_for_all(SerialWork(x));
  }
}
void ParallelWork(int n) {
  if (n <= 1) { doUnitWork(); }
  else {
    spawn(ParallelWork(n - n/2));
    spawn_and_wait_for_all(ParallelWork(n/2));
  }
}
void SerialWork(int n) {
  for (int i = 0; i < n; i++) { doUnitWork(); }
}
```

Figure 4-5: TBB pseudocode for the pessimal example F(k,z) from Figure 4-4. Code for reference-counting of tasks has been omitted.

remain idle, even after G creates additional parallel work.

To form the complete example, chain k repetitions of F$(1,z)$, arranged so that repetition j can begin only after the z$_p$ task of repetition $j-1$ is complete. F is designed so that with depth-restricted work-stealing, DepthTrap finishes before G enables the next repetition of F. Then the k instances of DepthTrap occur sequentially, and F(k,z) requires at least kz time to execute. Set the values of x and y in F to "sufficiently large" values. More precisely, x should be large enough to guarantee that $P-1$ processors complete $P-1$ successful steals with high probability, and $y > x$ should be large enough to ensure that H(z) does not complete before DepthTrap(z).

This example can be applied to other definitions of depth or other forms of restricted work-stealing as long as one can construct a DepthTrap method which prevents processors from stealing the work in z$_p$ once they enter DepthTrap. For subtree-restricted work-stealing using parallel regions, this restriction appears automatically if DepthTrap is declared as a parallel region, since workers do not leave the region until it is completed. Even if one modifies parallel regions to allow processors that steal into the region to leave early, one can still construct an appropriate trap for processors, since the processor that starts the region is still not allowed to leave the region until it completes. An appropriate chain of $P-1$ nested regions can force all but one processor to be idle waiting for one other processor to complete.

Theoretical Analysis

The properties of F can be stated and proved more formally. The next definition and lemma tell us appropriate values for setting x and y. In particular, the values $x > X(\varepsilon)$ and $y > Y(\varepsilon, z)$ in Definition 4.1 are chosen to be sufficiently large to make the behavior of random work-stealing predictable, i.e., to satisfy Lemma 4.4.

Definition 4.1. *Let c_s be the maximum time for any steal attempt (successful or unsuccessful). Define two functions:*

$$X(\varepsilon) = c_s(P + P\ln P)\ln\left(\frac{P}{\varepsilon}\right)$$

$$Y(\varepsilon,z) = X(\varepsilon) + c_s(1+\lg z) + P^2 + P\lg z\,.$$

Lemma 4.4. *Consider an execution of* F$(1,z)$ *using a depth-restricted work-stealing scheduler. If $x \geq X(\varepsilon)$, and $y \geq Y(\varepsilon, z)$, then, with probability at least $1-\varepsilon$, $P-1$ processors are stuck in* DepthTrap(z) *for at least z time.*

Proof. For $i \in \{1,2,\ldots P-1\}$, let t_i be the time step when some processor begins working on node S_i in Figure 4-4. Similarly, let t_d be the time when some processor begins work on node S_d, i.e., a processor has reached the bottom of the chain of length $d = 1 + \lg z$ in DepthTrap. Intuitively, we prove the lemma by showing that t_{P-1}, the time that a processor starts working z$_s$, is likely to satisfy $t_{P-1} \leq X(\varepsilon) + c_s d$. Then, since $y \geq Y(\varepsilon,z) > X(\varepsilon) + c_s d$, the subcomputation H$(z)$ does not generate any parallel work before time t_{P-1}, and processors must steal only from `DepthTrap`. Furthermore, if we have $t_{P-1} - t_d \leq X(\varepsilon) \leq x$, then it is impossible for a processor to finish a serial block of work x before t_{P-1}. Thus, each of the $P-2$ tasks with x serial work and the task $\mathtt{z}_s$ must be executed by one of $P-1$ distinct processors. Finally, once $P-1$ processors have stolen from `DepthTrap`, they are trapped waiting on $\mathtt{z}_s$ at a depth larger than the depth of any parallel work generated by $\mathtt{H}(z)$.

More precisely, to bound t_{P-1}, we construct events A_i which capture the notion that the t_i's are meeting their "likely" deadlines. Let A_1 be the event that

$$t_d \leq c_s d + c_s P \left(\frac{\ln(P/\varepsilon)}{P-1} \right) .$$

Similarly, for $j \in \{2, 3, \ldots P-1\}$, let A_j be the event that

$$t_j \leq c_s d + c_s P \left(\sum_{i=1}^{j} \frac{\ln(P/\varepsilon)}{P-i} \right) .$$

We can show by induction that $\Pr(\bigcap_{i=1}^{j} A_i) \geq (1 - \varepsilon/P)^j$. Substituting $j = P-1$ and simplifying the sum gives us $t_{P-1} \leq X(\varepsilon) + c_s d$ with probability at least $1 - \varepsilon$.

In the base case for induction, we compute $\Pr(A_1)$. Initially, one processor begins work on $\mathtt{G}$, and $P-1$ other processors attempt to steal S_1. Since processors steal randomly, the probability that S_1 has not been stolen after n steal attempts is at most $(1 - 1/P)^n < e^{-n/P}$. Thus, with $P-1$ processors stealing, S_1 is stolen before time $c_s P \ln(P/\varepsilon)/(P-1)$ with probability at least $1 - \varepsilon/P$. Once work begins at node S_1, we know some processor must reach S_d in at most $c_s d = c_s(1 + \lg z)$ time (i.e., $t_d - t_1 \leq c_s d$) since in the worst case, steals happen for every right S-node on the chain of length d in `DepthTrap`. Thus, we have $\Pr(A_1) \geq 1 - \varepsilon/P$.

To bound $\Pr(A_1 \cap A_2)$, we first condition on A_1 occurring. Once a processor reaches S_d, it then quickly spawns S_2 and begins working on a block with serial work x. For any time t in the range $t_d < t < t_2$, at least $P-2$ processors must be idle and trying to steal S_2. Thus, using the same analysis as for t_1, we know that $t_2 - t_d \leq c_s P \ln(P/\varepsilon)/(P-2)$ with probability at least $1 - \varepsilon/P$. Thus, we have $\Pr(A_2 | A_1) \geq 1 - \varepsilon/P$, and $\Pr(A_1 \cap A_2) = \Pr(A_2 | A_1) \cdot \Pr(A_1) \geq (1 - \varepsilon/P)^2$.

We complete the induction by repeating this analysis for the remaining A_j. Conditioned on the event that $\bigcap_{i=1}^{j-1} A_i$, we know for times t such that $t_{j-1} < t < t_j$, at least $P-j$ processors are trying to steal S_j. Thus, we have $t_j - t_{j-1} \leq c_s P \left(\frac{\ln(P/\varepsilon)}{P-j} \right)$ with probability at least $1 - \varepsilon/P$, and hence $\Pr\left(\bigcap_{i=1}^{j} A_i \right) \geq (1 - \varepsilon/P)^j$.

Finally, to show $t_{P-1} - t_d \leq X(\varepsilon)$ with probability at least $1 - \varepsilon$, note that if we ignore S_1 and compute deadlines for $t_i - t_d$ instead of t_i, the same inductive proof used to bound t_{P-1} applies. □

By using Lemma 4.4, we can bound the completion time of $\mathtt{H}(z)$ and `DepthTrap`(z).

Theorem 4.5. *For an execution of* $\mathtt{F}(1, z)$ *using depth-restricted work-stealing, let* T_H *and*

T_D be the completion times of `H(z)` *and* `DepthTrap(z)`*, respectively. If $x \geq X(\varepsilon)$ and $y \geq Y(\varepsilon, z)$, then $T_H - T_D \geq 0$, i.e.,* `H(z)` *does not finish before* `DepthTrap`*.*

Proof. By Lemma 4.4, since `DepthTrap` begins executing z_s at time t_{P-1} and `DepthTrap` requires at most $z + P + \lg z$ additional time to finish, we have $T_D \leq t_{P-1} + z + \lg z + P$. Also from Lemma 4.4, with probability at least $1 - \varepsilon$, we know `DepthTrap` keeps $P - 1$ processors occupied and `H(z)` executes serially for at least $t_{P-1} + z$ time. Since `H` has at least $y + z$ work, it cannot finish before time $T_H \geq t_{P-1} + z + (y - t_{P-1})/P$. Then, we know $T_H - T_D \geq (y - t_{P-1})/P - \lg(z) - P$. By substituting $t_{P-1} \leq X(\varepsilon) + c_s(1 + \lg z)$, we get that $T_H - T_D \geq 0$. □

Corollary 4.6. *Let $x = X(\varepsilon/k)$ and $y = Y(\varepsilon/k)$. With probability at least $1 - \varepsilon$, a depth-restricted work-stealing scheduler using P processors requires at least $\Omega(kz)$ time to execute* `F(k,z)`*.*

Proof. By Theorem 4.5, the execution of `F(1,z)`, `H(z)` does not finish before `DepthTrap(z)` with probability at least $1 - \varepsilon/k$. At least z time is required to execute `DepthTrap(z)`. Thus, using a union bound over k repetitions of `F`, we conclude that `F(k,z)` requires at least $\Omega(kz)$ time with probability at least $1 - \varepsilon$. □

A runtime using unrestricted work-stealing can complete `F(k,z)` quickly, however, because it can complete each z_p quickly (i.e., in $O(z/P)$ time) and overlap the executions of the serial z_s tasks from the k repetitions of `F`. Lemma 4.7 states this result more formally, and Theorem 4.8 compares depth-restricted work-stealing and unrestricted work-stealing for `F(k,z)`.

Lemma 4.7. *Let $x = X(\varepsilon/k)$ and $y = Y(\varepsilon/k)$. With probability at least $1 - \varepsilon$, an unrestricted work-stealing scheduler using P processors can execute* `F(k,z)` *in $O(kz/P + z)$ time, assuming that $z = \omega(kP^2 \lg(kP/\varepsilon))$.*

Proof. The proof follows from the analysis of Cilk [30], which has an unrestricted work-stealing scheduler. For a Cilk computation with work T_1 and span T_∞, the running time on P processors is $O(T_1/P + T_\infty + \lg(P/\varepsilon))$, with probability at least $1 - \varepsilon$.

The span of `F` is $T_\infty \leq k(y + \lg z) + z$. From our choices of x and y, we know that $x = O(P^2 \lg(kP/\varepsilon))$ and $y = x + O(P^2 + P \lg z)$. If we have $z = \omega(kP^2 \lg(kP/\varepsilon))$, then one can show that $ky = o(z)$ and $k \lg(z) = o(z)$. Thus, we have $T_\infty = O(z)$. Similarly, the work of `F` is $T_1 = k((P-2)x + y + 2z) + \Theta(k(P + \lg z))$. Since the $2kz$ term asymptotically dominates the other terms, we have $T_1 = O(kz)$. □

Theorem 4.8. *There exists a computation for which the ratio of the runtime using a depth-restricted work-stealing scheduler compared to the runtime using an unrestricted work-stealing scheduler is $\Omega(P)$.*

Proof. Choose $k = \Omega(P)$ and $z = \omega(kP^2 \lg(kP/\varepsilon))$. Then the computation `F(k,z)` satisfies Corollary 4.6 and Lemma 4.7, producing a competitive ratio of $\Omega(P)$. □

4.5 Conclusions

In this chapter, I have described PR-Cilk, a runtime system design that uses parallel regions to support SP-reciprocity using subtree-restricted work-stealing. The PR-Cilk design demonstrates that a dynamic-threading platform can support the composition of parallel functions inside legacy callbacks and still providing provable guarantees on performance. I conclude this chapter by describing some potential extensions to PR-Cilk and some avenues for future research.

I describe a design of PR-Cilk that uses the parallel regions from HELPER without much modification. HELPER already provides a correct implementation for PR-Cilk as well the completion-time bounds stated in Section 4.3. Since SP-reciprocity has slightly different semantics than helper locks, however, one may change some of the policies in HELPER for moving workers between regions. Using different policies may improve PR-Cilk's performance for some classes of programs. In this section, I discuss these potential optimizations. First, I consider ***adaptive callback regions***, regions in which helper workers that enter a region A can potentially leave before A completes. Second, I consider ***capacity-limited regions***, regions A which allow only a limited number of workers $P(A)$ to enter, where $P(A)$ can be less than P, the total number of worker threads.

The default implementation of HELPER described in [10] uses what we call an ***absorbing transition policy*** — namely, (1) a worker p in region A enters a nested region B immediately if p ever encounters an empty deque in $A.dqpool$ with a child deque q in $B.dqpool$, even if q is empty,[7] and (2) workers never leave a region A until A completes.

Alternatively, one might consider modifying this policy to support ***lazy entering*** and/or ***eager exits***. In lazy entering scheme, a worker p only enters a nested region A through random work-stealing if the deque that p encounters is nonempty. If A is a serial region, then this policy prevents workers from unnecessarily entering A. Similarly, in an eager exit scheme, workers assigned to a region can leave a region before it completes. In order to maintain good space bounds, the worker which starts a particular parallel region A must remain in A until A finishes, but any other worker p (which enters A due to random stealing) can leave whenever p's deque in $A.dqpool$ is empty. This strategy might provide better performance for programs with mostly sequential regions, since workers can leave the region and work on something else. On the other hand, this strategy might cause workers to repeatedly enter and leave a region, thereby incurring higher overhead. These modifications do not change the space bound guaranteed by PR-Cilk and are unlikely to improve the worst-case time bound, since the lower-bound computation from Section 4.4 still applies. These changes may, however, improve the performance of PR-Cilk in practice.

Another modification to the PR-Cilk scheme would be to allow programmers to specify a capacity limitation on callback regions. For example, if the programmer knows that a particular callback region has a parallelism of 4, they can specify it, and the runtime system would then allow only 4 workers to enter the region. This scheme may improve performance for programs with callback regions that have limited but predictable parallelism.

Although parallel regions were originally designed for supporting helper locks, with

[7] With an absorbing transition policy, it is sufficient for a worker p to look only one level deep in the victim's deque chain, because if p finds an empty deque q, it enters the region for q and is then able to steal from q's child deque.

minimal change, PR-Cilk can use regions to support SP-reciprocity. An interesting research question is whether parallel regions have other use cases, and how these uses interact with the existing scheduler. For example, in the stencil computation demo, the programmer might wish to execute the visualization as a parallel "I/O region" which has a static number of worker of threads dedicated to it. One natural question is whether one can adapt a Cilk-style scheduler and programming model to handle this type of interaction between static and dynamic threading. Conceivably, one might be able to use techniques of providing adaptive parallelism feedback as described in [7] to achieve provable completion-time bounds for such a system.

Chapter 5

Memory Models for Nested Transactions

Transactional memory (TM) is a term that describes a collection of hardware and software mechanisms that provide a transactional interface for accessing memory. Atomic transactions are a well-known and useful abstraction for programmers writing parallel code which have been utilized in database systems [56] for decades. More recently, however, transactional memory [65] has become an active area of study as researchers have observed that transactions might be used as a general-purpose mechanism for synchronizing access to shared memory in parallel programs. TM systems often provide programmers with a simple interface for specifying a block of code as a transaction. Conceptually, a TM system enforces atomicity by tracking the memory locations that each transaction accesses, finding "conflicts" between active transactions, and aborting and retrying transactions that conflict.

Researchers have argued that using TM for synchronization is more composable than using locks. As discussed in Section 1.4, programming with transactions is composable because TM systems can support nested transactions. Unfortunately, it turns out that nontrivial nesting of transactions can introduce significant complications into TM, particularly when transactions are allowed to contain nested transactions with nested parallelism, or when programmers uses the optimization of "open-nested" transactions. To provide composable synchronization using transactions in a dynamic-threading platform, it is helpful to have a precise specification of the semantic guarantees provided by TM with nested transactions.

Contributions

In this chapter, I present the ***transactional-computation*** framework, a formal model for understanding the semantics of TM systems with nested transactions.[1] Using this framework, one can formally describe "serializability," the classical correctness condition for transactions, and then generalize this condition to handle transactions with nested parallel transactions. Chapter 6 uses this transactional computation framework to describe how a TM runtime can support nested parallel transactions, and Chapter 7 uses the framework to explore the semantics of "open-nested" transactions.

[1] The transactional computation framework is derived from the formal model described in [8], which is joint work with Kunal Agrawal and Charles E. Leiserson.

Chapter Outline

The remainder of this chapter is organized as follows. Section 5.1 reviews the basic concepts of transactional memory (TM) from the literature and discusses some of the semantic complexities that nested parallelism introduces into TM. Section 5.2 presents the transactional-computation framework. Section 5.3 uses this framework to define several memory models for transactional memory — the model of serializability, as well as the model of "prefix-race freedom," a model which is seemingly weaker, but which more closely matches the behavior of a TM runtime system. In fact, Section 5.4 shows, the memory models of prefix-race freedom and serializability are equivalent for programs without open-nested transactions. Section 5.5 uses these memory models to define transaction conflict for TM with nested parallel transactions, and then it presents an operational model for TM based on this conflict definition. Finally, Section 5.6 shows that this operational model guarantees that transactions are serializable.

5.1 Overview of Transactional Memory

This section reviews the notions of serializability and conflict detection in transactional memory (TM) and discusses the complications that nested parallel transactions add to TM. In particular, this section presents example code illustrating why a TM system requires additional runtime mechanisms to support nested parallel transactions.

Serializability and Conflict Detection

TM systems typically guarantee that transactions are ***serializable*** [107], that is, transactions affect global memory as if they were executed one at a time in some order, even if in reality, several transactions executed concurrently.

As a concrete example, consider the transactions shown in Figure 5-1. This code assumes that programmers specify transactions using ***atomic blocks***, i.e., the code enclosed inside a block labeled with the `atomic` keyword represents a transaction. The transaction X reads two values from an array `A` at indices `a` and `b` and stores their sum into `A` at index `c`. X also inserts the values from `A` at indices `a` and `b` into their corresponding slots in array `B`. The transaction Y performs an analogous computation to X, except using indices `d`, `e`, and `f`. A TM system guarantees that a concurrent execution of X and Y is serializable, i.e., it appears as though either X executed serially before Y (as in Schedule 1), or vice versa, i.e., Y executes serially before X. In reality, the system may interleave memory accesses from X and Y, as in Schedules 2 or 3, but the TM system is responsible for guaranteeing that these schedules are equivalent to a serial schedule (e.g., Schedule 1).

To guarantee that transactions are serializable, a TM system tracks the memory locations accessed by each transaction, aborting transactions that might generate a conflict. For the example in Figure 5-1, if the sets $\{$`a`,`b`,`c`$\}$ and $\{$`d`,`e`,`f`$\}$ do not intersect, then a TM system will not detect any conflicts for X and Y since they access disjoint sets of memory. If the sets intersect, however, then the transactions X and Y may conflict if both transactions try to change the same location in the array `B` (e.g., `a` = `e`), or if one transaction tries to read

```
// Transaction X
atomic {
  int v1, v2;
  v1 = A[a];       // X1
  Insert(a, v1);  // X2
  v2 = A[b];       // X3
  Insert(b, v2);  // X4
  A[c] = v1+v2;   // X5
}
```

```
// Transaction Y
atomic {
  int v1, v2;
  v1 = A[d];       // Y1
  Insert(d, v1);  // Y2
  v2 = A[e];       // Y3
  Insert(e, v2);  // Y4
  A[f] = v1+v2;   // Y5
}
```

```
void Insert(int k,
            int v) {
  B[k] = v;
}
```

Schedule 1: $X_1, X_2, X_3, X_4, X_5, Y_1, Y_2, Y_3, Y_4, Y_5$
Schedule 2: $Y_1, X_1, X_2, X_3, X_4, Y_2, Y_3, Y_4, X_5, Y_5$
Schedule 3: $Y_1, X_1, X_2, Y_2, X_3, X_4, Y_3, Y_4, X_5, Y_5$

Figure 5-1: Two concurrent transactions X and Y. We assume X can be subdivided into fragments, namely $X_1, X_2, \ldots, X_5$, and similarly for Y. The fragments X_2, X_4, Y_2 and Y_4 represent inserts into a shared global table. Schedule 1 represents a serial execution order. If the indices a through f are all distinct, then Schedules 2 and 3 are interleaved execution orders which are equivalent to Schedule 1.

a location to which the other is writing (e.g., if c = d).

More formally, consider a TM system that operates on a set $\mathcal{M}$ of memory locations. For each transaction X, a TM system conceptually maintains a ***readset*** (denoted by $\texttt{R}(X)$), and a ***writeset*** (denoted by $\texttt{W}(X)$), which represent the set of memory locations (i.e., the subsets of $\mathcal{M}$) that the transaction X has read from and written to, respectively. Define a ***flat TM system*** as a TM without any nested transactions. In this dissertation, we consider TM systems with ***eager conflict detection***, that is, a TM that checks for "conflicts" on the first access to every memory location. More precisely, we consider TM with the following definition of conflict:

Definition 5.1. *A flat TM system with eager conflict detection reports a* ***conflict*** *between two active (i.e., currently executing) transactions X and Y if*

$$\{W(X) \cap (R(Y) \cup W(Y))\} \cup \{W(Y) \cap (R(X) \cup W(X))\} \neq \emptyset ,$$

that is, X and Y have conflicting readsets and writesets.

According to Definition 5.1, a conflict occurs when transaction X tries to write to a memory location ℓ that transaction Y has read from or written to, or vice versa.

With eager conflict detection, if the end of a transaction X is reached without detecting any conflicts, then assuming no other memory accesses from (other) transactions occur, then the TM runtime is free to ***commit*** the transaction X, i.e., make X's changes to memory locations permanent. Otherwise, if the runtime detects a conflict, then it may ***abort*** X, i.e., rollback any changes to memory performed by X. Thus, transaction X appears either to execute atomically if it commits, or not at all if it aborts.

Instead of eager conflict detection, some TM systems implement ***lazy conflict detection***, where in order to commit a transaction X, the TM runtime must first ***validate*** X — scan over the readset and writeset of X to determine whether X can commit. For a more extensive classification of conflict detection in TM, see [88, Section 2.3] or [111].

Eager conflict detection according to Definition 5.1 is "conservative" in the sense that a TM system with eager conflict detection may report a conflict between two transactions even though a more generic TM could theoretically finish both transactions without reporting a conflict. For example, consider Schedule 2 from Figure 5-1 for a generic TM, with all `a` through `f` distinct except `b = e`. In Schedule 2, after X_4 completes, the TM system has `A[b]` in `W`(X). Thus, when Y_3 executes, in theory, transaction Y could read the value of `A[e] = A[b]` written by transaction X. Then, both X and Y could both complete and commit at the same time. It can be difficult to implement this more generic TM system, however, because it must support ***cascading aborts*** of transactions — if Y reads a value written by X, but then later X needs to abort, then the abort of X would also force Y to abort.

More generally, it is not feasible to implement an ideal TM system, i.e. a system that permits every serializable schedule without aborting transactions. Papadimitriou [107] demonstrates that detecting whether a given schedule of reads and writes is serializable is NP-complete. Thus, a TM system must be somewhat conservative in conflict detection to be able to operate efficiently online.

Nested Transactions

As mentioned in Section 1.4, researchers have argued that using TM is composable because TM systems can handle nested transactions. TM systems can implement***flat nesting*** of transactions, which conceptually merges any nested transaction Z into its outer transaction X. Flat-nested transactions can allow users to call a function inside a transaction X without needing to know whether it will generate a nested transaction Z. For example, in the `Insert` method from Figure 5-1, suppose that we enclose line 18 within an `atomic` block. In this example, flat nesting does not change the correctness of the transactions calling the `Insert` because any transaction that would conflict with transaction Z must also conflict with transaction X.

To enhance the composability of TM, however, we would like to have support for nested parallel transactions. For example, consider the code in Figure 5-2, where the function `F` (line 11) is a transaction which executes two calls to `update2` (line 4) in parallel. Let X_1 and X_2 denote the transactions invoked in lines 13 and 14, respectively. With nested parallel transactions, a TM system guarantees that X_1 and X_2 execute atomically and thus do not interfere with each other, even though they are both nested transactions.

We would also like to have transactions nested to arbitrary depth. For example, in X_1, the `update2` function itself performs a `spawn` (line 6), creating two parallel transaction instances of the `atmUpdate` (line 1). Let Y_1 and Y_2 denote the instances of `atmUpdate` created by X_1 in lines 6 and 7, respectively.

When transactions can contain nested parallelism and nested transactions, it turns out that TM using only flat-nested transactions no longer guarantees that transactions are serializable. Flat nesting, which would elide all but the outermost transaction, does not guarantee a correct execution of this code. Normally, a call to `update2` has no visible effect when `x` and `y` point to the same variable, since the variable is incremented and then decremented.

```
 1 void atmUpdate(int* x, int v) {
 2   atomic { *x = *x + v; }
 3 }
 4 void update2(int* x, int* y) {
 5   atomic {
 6     spawn atmUpdate(x, 1);
 7     atmUpdate(y, -1);
 8     sync;
 9   }
10 }
11 void F(int* X, int a, int b, int c, int d) {
12   atomic {
13     spawn update2(&X[a], &X[b]);
14     update2(&X[c], &X[d]);
15     sync;
16   }
17 }
```

Figure 5-2: A code example which requires closed nesting. The `update2` increments the integer pointed to by `x` and decrements the integer pointed to by `y` using parallel transactions. The function `F` invokes two instances of `update2` in parallel, which modify the array `X` at the positions specified by the indices `a` through `d`.

With flat nesting, however, the transactions inside `update2` would be eliminated. Then, if `x` and `y` were equal, the transaction instances Y_1 and Y_2 could potentially interfere with each other, changing the variable's value.

Instead, to correctly support transactions with nested parallelism, a TM system must use what are referred to in the literature as ***closed-nested*** transactions [101]. One can define closed nesting operationally. Every transaction Y maintains its own readset and writeset, even if Y is nested inside another transaction X. When a transaction Y is closed-nested inside X and commits, it merges all memory locations in $\mathtt{R}(Y)$ into $\mathtt{R}(X)$, and similarly it merges $\mathtt{W}(Y)$ into $\mathtt{W}(X)$, thereby effectively committing Y with respect to X.

To consider closed nesting for the previous example, suppose that the input to X_1 has both `x` and `y` equal and pointing to some memory location ℓ. Then, while Y_1 is executing, ℓ will be added to $\mathtt{R}(Y_1)$ and $\mathtt{W}(Y_1)$. Thus, while Y_1 is active, the TM runtime would detect a conflict if Y_2 tries to concurrently to modify ℓ. If Y_2 tries to modify ℓ after Y_1 commits, however, ℓ would have been merged into the $\mathtt{R}(X_1)$ and $\mathtt{W}(X_1)$, and then Y_2 would not generate a conflict. Nevertheless, the transaction X_2 (line 14) running in parallel with X_1 can still cause a conflict if X_2 accesses ℓ.

It can be tricky to reason about the correctness of closed nesting using an operational definition, that is, a description of how TM maintains transaction readsets and writesets. With an operational definition, one needs to consider both how the TM runtime performs conflict detection and how it schedules code on different processors. To pose the problem more concretely, consider the function `X` shown in Figure 5-3, which performs parallel increments to a shared variable, and its corresponding computation DAG in Figure 5-4. The function `X` exhibits several race conditions on the variable `x`, as the function contains

```
void update(int* x, int v) {
  *x = *x + v;
}
void Y(int* x) {
  spawn update(x, 100);
  update(x, 1000);
  sync;
}
void X() {
  int x = 0;
  spawn update(&x, 1);
  update(&x, 10);
  Y(&x);
  sync;
  printf("x = %d", x);
}
```

Figure 5-3: A Cilk function X that updates a shared variable x in parallel, without synchronization. This program contains several data races. Assuming sequential consistency, the possible valid outputs for x are 1, 11, 101, 110, 111, 1001, 1010, 1011, 1101, 1110, or 1111.

several concurrent updates to x which are not guaranteed to execute atomically. Since every update to x can be broken up into two memory operations, namely a read from x and a write of the updated value to x, some updates might be lost, depending on how instructions interleave in a parallel execution.

Suppose that we wrap segments of the code from Figure 5-3 inside atomic blocks, as shown in Figure 5-5. How does a TM system execute this code, and what final values of x are possible in an execution? Fewer interleavings of the updates are possible because some updates occur inside transactions. Races still exist, however, because an update to x outside a transaction (e.g., line 10 in Figure 5-5) can run in parallel with an update to x inside a transaction (line 11). Unfortunately, the behavior of having "nontransactional"

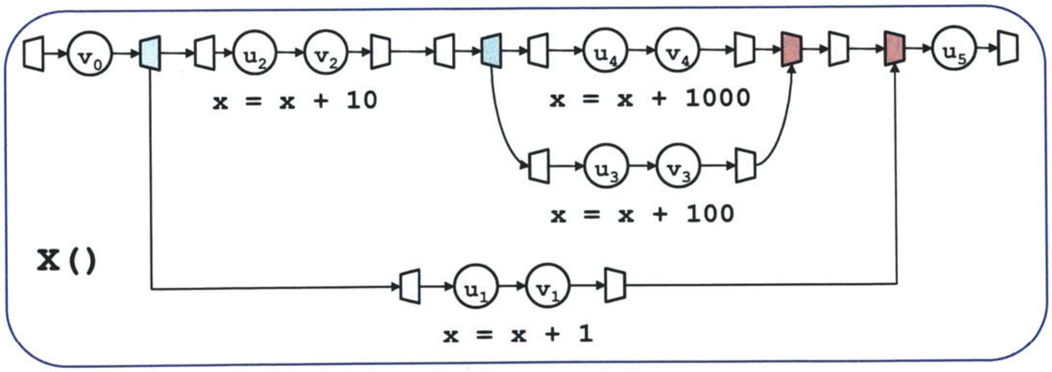


Figure 5-4: The series-parallel DAG for the sample program given in Figure 5-3.

```
void update(int* x, int v) {
  *x = *x + v;
}
void atmUpdate(int* x, int v) {
  atomic {
    *x = *x + v;
  }
}
void Y(int* x) {
  spawn update(x, 100);
  atmUpdate(x, 1000);
  sync;
}
void X() {
  int x = 0;
  atomic {
    spawn atmUpdate(&x, 1);
    atomic {
      update(&x, 10);
      Y(&x);
    }
    sync;
  }
  printf("x = %d", x);
}
```

Figure 5-5: The code from Figure 5-3 with transactions added.

code in parallel with a transaction is often dependent on the particular TM implementation.

Figures 5-2 and 5-5 illustrate some of the subtleties involved in understanding the correctness of a TM implementation. Although the notion of serializability [107] is intuitive in a static-threading programming model, where transactions execute serially on different threads, in an environment with dynamic threading, where transactions can have nested parallelism, we need a more precise model for understanding TM. Thus, the remainder of this chapter describes a framework for transactional computations, which can be used to understand the correctness guarantees provided by TM.

5.2 Transactional Computations

This section defines ***transactional computations***, a framework for modeling the properties of TM systems that support nested transactions. Our framework is inspired by a combination of the computation-tree framework for Cilk (reviewed in Section A.4), as well as the computation-centric memory models of Frigo and Luchangco in [48, 52, 96]. The transactional-computation framework uses computation trees to model both the parallel structure of a computation and the nesting structure of transactions. It also uses the notion of an "observer function" from [52] to model the behavior of read and write memory op-

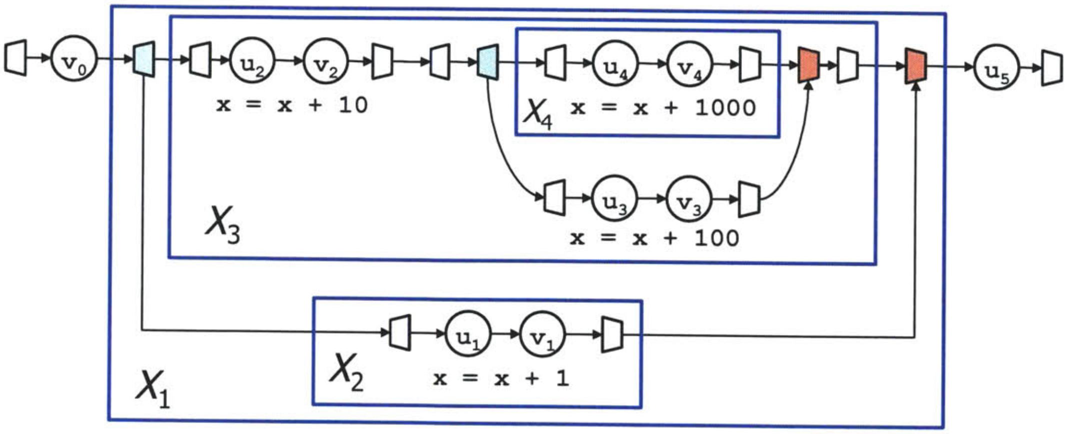


Figure 5-6: The computation DAG for the sample code given in Figure 5-5. Transactions are enclosed in rectangles.

erations. Section 5.3 uses this framework to define memory models for TM and explore serializability for TM systems with nested parallel transactions.

The computation-centric model focuses on an *a posteriori* analysis of a program execution. After a program completes, we assume that the execution has generated a ***trace*** which is abstractly modeled as a pair (C, Φ), where C is a computation tree (as defined in Section A.4), describing the memory operations performed and transactions executed, and Φ is an "observer function" describing the behavior of read and write operations. We shall define C and Φ more precisely below. We define $\mathcal{U}$ to be the set of all possible traces.

Within this framework, we define a memory model as follows:

Definition 5.2. *A **memory model** is a subset $\Delta \subseteq \mathcal{U}$.*

That is, Δ represents all executions that "obey" the memory model.

Computation Trees

This section reviews the computation tree framework for programs without transactions. This framework is described in more detail in Section A.4.

A ***computation tree*** C has two types of nodes: primitive operation nodes $\texttt{primOps}(C)$, and control nodes $\texttt{spNodes}(C)$. In transactional computations, a control node can either be an S-node or a P-node. For an S-node X, all the child subtrees of X must be executed in series, from left to right, but the child subtrees of a P-node are allowed to execute in parallel with each other in an arbitrary order. We assume that computation trees have a canonical form; more specifically, the set $\texttt{sNodes}(C)$ of S-nodes can be partitioned into task nodes $\texttt{tasks}(C)$ and function nodes $\texttt{functions}(C)$, and every P-node $X \in \texttt{pNodes}(C)$ has only two task nodes as children. Define $\texttt{nodes}(C) = \texttt{primOps}(C) \cup \texttt{spNodes}(C)$.

This chapter uses many of the structural notations for computation trees defined in Section A.4. Denote the ***root*** of the tree C as $\texttt{root}(C)$. For any node $B \in \texttt{nodes}(C)$, let $\texttt{parent}(B)$ denote the ***parent*** of B in C, or NULL if $B = \texttt{root}(C)$. Similarly, let

$\texttt{children}(B)$ denote the ordered set of B's ***children***, or NULL if B is a leaf. For any tree node $B \in \texttt{nodes}(\mathcal{C})$, let $\texttt{ances}(B)$ denote the set of all ***ancestors*** of B in $\mathcal{C}$, and let $\texttt{desc}(B)$ denote the set of all B's ***descendants***. Denote the set of ***proper ancestors*** (and proper descendants) of B by $\texttt{pAnces}(B)$ (and $\texttt{pDesc}(B)$). Denote the ***least common ancestor*** of two nodes $B_1, B_2 \in C$ by $\texttt{LCA}(B_1, B_2)$.

For any set of computation tree nodes $J \subseteq \texttt{nodes}(\mathcal{C})$, it is also useful to define the ***leaf set*** of J as

$$\texttt{leafSet}(J) = \{X \in J \; : \; \texttt{pDesc}(X) \cap J = \emptyset\} \; .$$

Conceptually, $\texttt{leafSet}(J)$ is the subset of J which form the "leaves" of J. All other nodes $Y \in J$ are on the path from some $X \in J$ to $\texttt{root}(\mathcal{C})$. For sets J which are guaranteed to have $|\texttt{leafSet}(J)| = 1$ (e.g., $\texttt{leafSet}(\texttt{ances}(X))$ for some X), we define $\texttt{leaf}(J)$ as the unique element in $\texttt{leafSet}(J)$.

Since subtrees of $\mathcal{C}$ are themselves valid computation trees, we shall sometimes overload notation and use a subtree and its root interchangeably. For example, if $X = \texttt{root}(\mathcal{C})$, then $\texttt{primOps}(X)$ refers to all the primitive operation nodes in $\texttt{primOps}(\mathcal{C}) \cap \texttt{desc}(X)$.

As reviewed in Section A.4, there is a direct correspondence between a computation tree $\mathcal{C}$ and its computation DAG $\mathcal{G} = (V(\mathcal{C}), E(\mathcal{C}))$. Intuitively, every node $X \in \texttt{spNodes}(\mathcal{C})$ maps to two vertices $\texttt{source}(X)$ and $\texttt{sink}(X)$ in the computation DAG $\mathcal{G}$, which enclose the subDAG corresponding to the subtree rooted at X. For any two vertices $u, v \in V(\mathcal{C})$, we say $u \prec_{\mathcal{G}} v$ if there is a path from u to v in $\mathcal{G}$.

Finally, we consider topological sorts of computation DAGs. For a DAG $\mathcal{G}$, define $\texttt{tsorts}(\mathcal{G})$ as the set of all topological sorts of $\mathcal{G}$. Any sort $\mathcal{S} \in \texttt{tsorts}(\mathcal{G})$ defines a precedence relation $<_{\mathcal{S}}$: for any two vertices $u, v \in V(\mathcal{C})$, we say $u <_{\mathcal{S}} v$ if u comes before v in the sort order $\mathcal{S}$.

Computations with Transactions

We can extend the computation tree model to add transactions. Unless otherwise specified, in this chapter, we consider a ***closed TM system*** — a TM system that supports only closed-nested transactions.

Since TM is concerned with accesses to memory, we require additional definitions for memory operations. We define a special subset of primitive operations — ***memory-operation nodes***, which we denote by $\texttt{memOps}(\mathcal{C})$. Also, we define $\mathcal{M}$ to be the set of all memory locations. Each leaf node $u \in \texttt{memOps}(\mathcal{C})$ represents a single memory operation on a memory location $\ell \in \mathcal{M}$. We say that node u satisfies the ***read predicate*** $R(u, \ell)$ if u reads from location ℓ. Similarly, u satisfies the ***write predicate*** $W(u, \ell)$ if u writes to ℓ.

For transactional computations, we also specify that particular function nodes $X \in \texttt{functions}(\mathcal{C})$ are transactions. Let $\texttt{xactions}(\mathcal{C}) \subseteq \texttt{functions}(\mathcal{C})$ denote the set of transactions in the computation $\mathcal{C}$. Conceptually, a node $X \in \texttt{xactions}(\mathcal{C})$ corresponds to defining a transaction X that contains the subDAG starting at $\texttt{source}(X)$ and ending at $\texttt{sink}(X)$. The nesting structure of transactions is specified by the computation tree $\mathcal{C}$. Define a transaction Y as ***nested*** inside a transaction X if $X \in \texttt{ances}(Y)$. We say that two transactions X_1 and X_2 are ***independent*** if neither is nested inside the other, i.e., $\texttt{LCA}(X_1, X_2) \notin \{X_1, X_2\}$.

Since we are only concerned with memory operations and transactions, without loss of generality, in the remainder of our discussion of TM, we assume that $\texttt{primOps}(\mathcal{C}) = \texttt{memOps}(\mathcal{C})$ and $\texttt{functions}(\mathcal{C}) = \texttt{xactions}(\mathcal{C})$, i.e., all primitive operations are memory operations, and all functions are transactions. Equivalently, we could treat every non-memory operation as a read from its own private memory location, since such an operation does not contribute to transaction conflicts. Similarly, since every function node $F \in \texttt{functions}(\mathcal{C})$ must have an S-node as a parent, we can conceptually elide any F which is not a transaction by setting the parent of F's children to $\texttt{parent}(F)$.

To illustrate these definitions more concretely, Figure 5-7 shows a computation tree $\mathcal{C}$. Figure 5-8 shows the corresponding computation DAG.

We also define notation that expresses the relationship of a node B to transactions in the computation tree. For any node $B \in \texttt{nodes}(\mathcal{C})$, we define the ***transactional parent*** of B, denoted by $\texttt{xParent}(B)$, as

$$\texttt{xParent}(B) = \begin{cases} \texttt{parent}(B) & \text{if } \texttt{parent}(B) \in \texttt{xactions}(\mathcal{C}) \\ \texttt{xParent}(\texttt{parent}(B)) & \text{if } \texttt{parent}(B) \notin \texttt{xactions}(\mathcal{C}) \end{cases} .$$

Define the ***transactional ancestors*** of B as $\texttt{xAnces}(B) = \texttt{ances}(B) \cap \texttt{xactions}(\mathcal{C})$. Similarly, define the ***transactional descendants*** of B as $\texttt{xDesc}(B) = \texttt{desc}(B) \cap \texttt{xactions}(\mathcal{C})$. Finally, define the ***least common ancestor transaction***, denoted by $\texttt{xLCA}(B_1, B_2)$, as $Z = \texttt{LCA}(B_1, B_2)$ if $Z \in \texttt{xactions}(\mathcal{C})$, and as $\texttt{xParent}(Z)$ otherwise.

For every transaction $X \in \texttt{xactions}(\mathcal{C})$, the model distinguishes transactions as either committed or aborted.[2] Let $\texttt{committed}(\mathcal{C}) \subseteq \texttt{xactions}(\mathcal{C})$ denote the set of ***committed transactions***. Similarly, let $\texttt{aborted}(\mathcal{C}) = \texttt{xactions}(\mathcal{C}) - \texttt{committed}(\mathcal{C})$ be the set of ***aborted transactions***. These sets are useful for the following definition:

Definition 5.3. *For any transaction $X \in \texttt{xactions}(\mathcal{C})$, define the **content** of X as*

$$\texttt{content}(X) = V(X) - \bigcup_{Z \in \texttt{aborted}(X) - \{X\}} V(Z) .$$

That is, the content of X is the set of all operations that belong to X but not to any of X's aborted subtransactions. We always have $\texttt{content}(X) \subseteq V(X)$, with equality holding when X's subtree contains no aborted transactions.

We can also define the inverse mapping for content.

Definition 5.4. *For any node $B \in V(\mathcal{C})$, define the **holders** of B as*

$$\texttt{holders}(B) = \{Y \in \texttt{xactions}(\mathcal{C}) : B \in \texttt{content}(Y)\} .$$

That is, the holders of a node B is the set of all transactions that contain B.

Although conceptually, aborted transactions have no effect, the framework of transactional computations explicitly models them to aid in reasoning about transactions that abort

[2] For now, we consider only an *a posteriori* analysis. We do not model transactions that are in progress until Section 5.5.

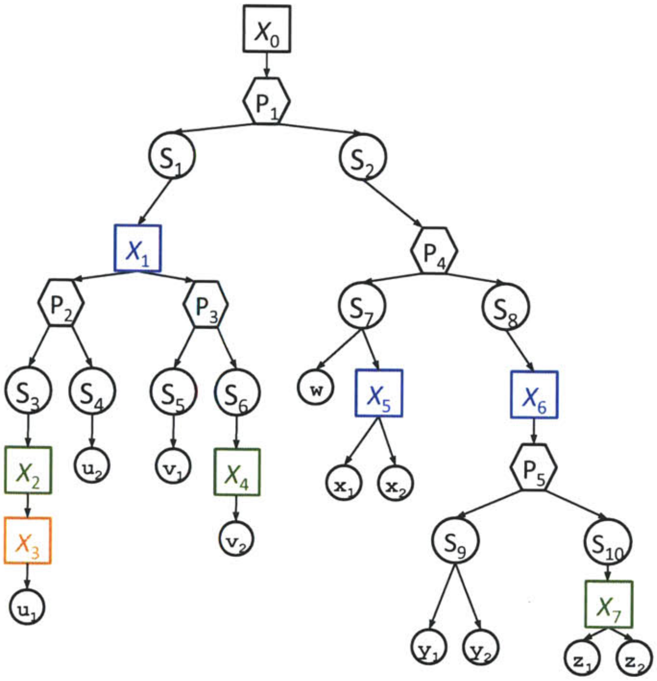


Figure 5-7: A computation tree $\mathcal{C}$ with transactions. More specifically, in this computation tree $\mathcal{C}$, we have the sets of transactions $\texttt{xactions}(\mathcal{C}) = \{X_0, X_1, \ldots, X_7\}$, P-nodes $\texttt{pNodes}(\mathcal{C}) = \{\mathsf{P}_1, \mathsf{P}_2, \ldots, \mathsf{P}_5\}$, task nodes $\texttt{tasks}(\mathcal{C}) = \{\mathsf{S}_1, \mathsf{S}_2, \ldots, \mathsf{S}_{10}\}$, and memory operations $\texttt{memOps}(\mathcal{C}) = \{u_1, u_2, v_1, v_2, w, x_1, x_2, y_1, y_2, z_1, z_2\}$.

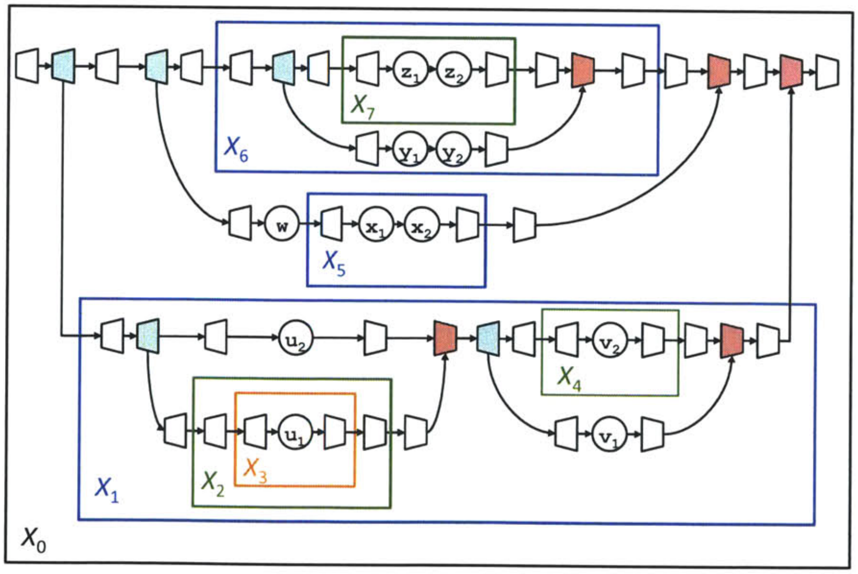


Figure 5-8: The computation DAG for the transactional computation from Figure 5-7. Transactions are enclosed in rectangles.

in an actual implementation. Also, modeling aborted transactions is useful in Chapter 7 for considering "open-nested" transactions, since open-nested transactions enable aborted transactions to have visible effects on memory.

Basic transactional semantics dictate that committed transactions should not "see" values written by vertices belonging to the content of an aborted transaction. One may argue whether one aborted transaction should be able to see values written by another aborted transaction. This dissertation takes the position that a transaction should be "well behaved" up to the point that it aborts. Thus, one aborted transaction should not see values written by other aborted transactions, although the values written by a vertex within an aborted transaction may be seen by other vertices within the same transaction.

The following definition describes which vertices are hidden from which other vertices according to these semantics.

Definition 5.5. *For any two vertices $u, v \in V(\mathcal{C})$, let $X = \texttt{LCA}(u, v)$. We say that u is **hidden** from v, denoted uHv, if*

$$u \in \bigcup_{Y \in \texttt{aborted}(X) - \{X\}} \texttt{content}(Y) .$$

In Figure 5-7, we have $v_2 H z_2$ if and only if one of X_1 or X_4 belongs to $\texttt{aborted}(\mathcal{C})$. The hidden relation H is not symmetric. For example, if we have $X_1 \in \texttt{committed}(\mathcal{C})$ and $X_6 \in \texttt{aborted}(\mathcal{C})$, then we have $y_1 H v_1$, but not $v_1 H y_1$.

Observer Functions

Instead of specifying the value that a vertex $v \in \texttt{memOps}(\mathcal{C})$ reads from or writes to a memory location $\ell \in \mathcal{M}$, we adopt the approach of the computation-centric framework of Frigo and Luchangco [48, 52], which abstracts away the values entirely. An ***observer function***[3] $\Phi(v) : \texttt{memOps}(\mathcal{C}) \to \texttt{memOps}(\mathcal{C}) \cup \{\texttt{source}(\mathcal{C})\}$ tells us which vertex $u \in \texttt{memOps}(\mathcal{C})$ writes the value of ℓ that v sees. For a given computation tree $\mathcal{C}$, if $v \in \texttt{memOps}(\mathcal{C})$ accesses location $\ell \in \mathcal{M}$, then a well-formed observer function must satisfy $\neg(v \prec_{\mathcal{G}(\mathcal{C})} \Phi(v))$ and $W(\Phi(v), \ell)$. In other words, v cannot observe a value from a vertex that comes after v in the computation DAG, and v can only observe a vertex if it actually writes to location ℓ. To define Φ on all vertices that access memory locations, we assume that the vertex $\texttt{source}(\mathcal{C})$ writes initial values to all of memory. When $\Phi(v) = \texttt{source}(\mathcal{C})$ for a memory operation v, sometimes for shorthand we write $\Phi(v) = \perp$.

Traces and Memory Models

Recall that Definition 5.2 states that a memory model Δ is a subset of $\mathcal{U}$, the universe of all possible traces. Later, when we consider some basic memory models, it is convenient to restrict our attention to computations without transactions, or with only committed

[3] This definition of $\Phi(v)$ is similar to Frigo's definition in [48, 52], but with a salient difference, namely, Frigo's observer function defines $\Phi(v)$ for all memory locations, not just for the location ℓ that v accesses. Moreover, if $W(v, \ell)$, Frigo defines $\Phi(v) = v$, whereas we define $\Phi(v) = u$ for some $u \neq v$.

transactions. Thus, we define the following subsets of $\mathcal{U}$:

$$\begin{aligned}\mathcal{U}_0 &= \{(\mathcal{C},\Phi)\in\mathcal{U} : \texttt{xactions}(\mathcal{C})=\emptyset\}\ ,\\ \mathcal{U}_{\text{com}} &= \{(\mathcal{C},\Phi)\in\mathcal{U} : \texttt{aborted}(\mathcal{C})=\emptyset\}\end{aligned}$$

In other words, $\mathcal{U}_0$ contains traces (whose computations) include no transactions, and $\mathcal{U}_{\text{com}}$ contains traces that include only committed transactions.

Open-Nested Transactions

One can extend this framework of transactional computations to model ***open TM systems*** — TM systems that allow both closed-nested and open-nested transactions, as described in [98, 102, 103].

Conceptually, to model open nesting, one can partition the set $\texttt{xactions}(\mathcal{C})$ into two sets, $\texttt{closedX}(\mathcal{C})$ and $\texttt{openX}(\mathcal{C})$. When a transaction $X \in \texttt{closedX}(\mathcal{C})$ commits, its readset and writeset are merged into $\texttt{xParent}(X)$, as before. For an open-nested transaction $Y \in \texttt{openX}(\mathcal{C})$, however, the commit of Y commits the values Y's readset and writeset to memory immediately (i.e., to the root of $\mathcal{C}$). Thus, with open nesting, Definition 5.3 changes to also exclude operations from open-nested transactions.

Definition 5.6. *Consider a computation C on an open TM system. For any transaction $X \in \texttt{xactions}(\mathcal{C})$, define the content of X as*

$$\texttt{content}(X) = V(X) - \bigcup_{Z\in\texttt{aborted}(X)-\{X\}} V(Z) - \bigcup_{Z\in\texttt{openX}(X)-\{X\}} V(Z)\ .$$

Although the hidden relation in Definition 5.5 is for a closed TM system, one can also use this definition for an open TM system by using the new definition of $\texttt{content}(X)$ in Definition 5.6. For example, in Figure 5-7, if $X_2 \in \texttt{openX}(\mathcal{C})$ is an open transaction and $X_2, X_3 \in \texttt{committed}(\mathcal{C})$, then we do not have $u_1 H z_2$, even if $X_1 \in \texttt{aborted}(\mathcal{C})$. As described in [8], one can use these definitions and generalize the memory models defined in Section 5.3 to TM with open-nested transactions.

5.3 Transactional Memory Models

Using this framework of transactional computations, we can define various memory models for TM. We begin by using the framework to study a simple memory model for programs without transactions, namely Lamport's classic memory model of sequential consistency [87]. We then extend this model to consider more complex memory models, including the models of serializability" [107] and a model we call "prefix-race freedom."

Sequential Consistency

To define Lamport's model of sequential consistency [87] in this transactional computation framework, we first mimic the definition in [48] to define a memory model for sequential consistency for computations without transactions. We then extend this definition to include transactions as well.

We now define a "last writer" observer function as in [48].

Definition 5.7. *Consider a trace $(\mathcal{C},\Phi) \in \mathcal{U}_0$ and a topological sort $\mathcal{S} \in \mathtt{tsorts}(\mathcal{G}(\mathcal{C}))$. For all $v \in \mathtt{memOps}(\mathcal{C})$ such that $R(v,\ell) \vee W(v,\ell)$, the* ***last writer*** *of v according to $\mathcal{S}$, denoted $\mathcal{L}_{\mathcal{S}}(v)$, is the unique $u \in \mathtt{memOps}(\mathcal{C}) \cup \{\mathtt{source}(\mathcal{C})\}$ that satisfies three conditions:*

1. *$W(u,\ell)$,*
2. *$u <_{\mathcal{S}} v$, and*
3. *$\{w \in V(\mathcal{C}) \ : \ (u <_{\mathcal{S}} w <_{\mathcal{S}}) \wedge W(w,\ell)\} = \emptyset$.*

In other words, if vertex v accesses (reads or writes) location ℓ, the last writer of v is the last vertex u before v in the order $\mathcal{S}$ that writes to location ℓ.

We can use the last-writer function to define sequential consistency for computations containing no transactions.

Definition 5.8. ***Sequential consistency*** *for computations without transactions is the memory model*

$$SC = \{(\mathcal{C},\Phi) \in \mathcal{U}_0 : \exists \mathcal{S} \in \mathtt{tsorts}(\mathcal{G}(\mathcal{C})) \text{ such that } \Phi = \mathcal{L}_{\mathcal{S}}\} \ .$$

By this definition, a trace $(\mathcal{C},\Phi) \in \mathcal{U}_0$ is sequentially consistent if there exists a topological sort $\mathcal{S}$ of $\mathcal{G}(\mathcal{C})$ such that the observer function Φ satisfies $\Phi(v) = \mathcal{L}_{\mathcal{S}}(v)$ for all memory operations $v \in \mathtt{memOps}(\mathcal{C})$. Definition 5.8 captures Lamport's notion [87] of sequential consistency: there exists a single order on all operations that explains the execution of program.

Figure 5-9 shows a sample computation DAG $\mathcal{G}(\mathcal{C})$ and two possible observer functions, Φ_1 and Φ_2. The trace $(\mathcal{C},\Phi_1)$ is sequentially consistent, but $(\mathcal{C},\Phi_2)$ is not because there is no topological sort that makes Φ_2 into a last-writer function. For the observer function Φ_1 with $\Phi_1(u_1) = \perp$, $\Phi_1(u_2) = u_1$, $\Phi_1(v_1) = \perp$, $\Phi_1(v_2) = \perp$, and $\Phi_1(u_3) = u_2$, the trace $(\mathcal{C},\Phi_1)$ is sequentially consistent, since Φ_1 is consistent with the order $\mathcal{S} = \langle u_1, u_2, v_1, v_2, u_3 \rangle$. On the other hand, the trace $(\mathcal{C},\Phi_2)$ with $\Phi_2(u_1) = \perp$, $\Phi_2(u_2) = u_1$, $\Phi_2(v_1) = \perp$, $\Phi_2(v_2) = \perp$, and $\Phi_2(u_3) = u_1$ is not sequentially consistent. Suppose for contradiction that there existed an order $\mathcal{S}$ such that $\Phi_2 = \mathcal{L}_{\mathcal{S}}$. Then, because $\Phi_2(u_3) = u_1$, we must have either $u_2 <_{\mathcal{S}} u_1$ or $u_3 <_{\mathcal{S}} u_2$. The first case is ruled out because $\Phi_2(u_2) = u_1$. Then, since $u_3 <_{\mathcal{S}} u_2$, we must have $v_2 << _{\mathcal{S}} v_1$. But then we have $\Phi_2(v_1) = \perp$, whereas $\mathcal{L}_{\mathcal{S}}(v_1) = v_2$, contradicting the fact that $\Phi_2 = \mathcal{L}_{\mathcal{S}}$.

We can extend this definition of sequential consistency to account for transactions. First, we consider a memory model that does not require atomicity of transactions, but only accounts for the fact that a transaction X in parallel with an aborted transaction Y should not "see" values written by Y. Moreover, this definition makes the assumption that an aborted computation is consistent up to the point that it aborts.

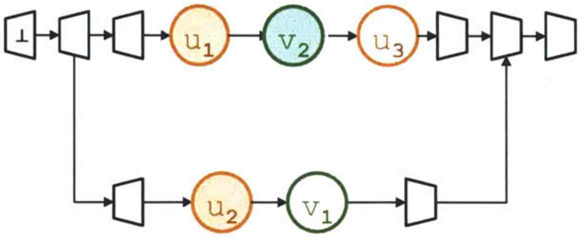


Figure 5-9: Example of sequential consistency for a computation $\mathcal{C}$ without transactions. Memory operations u_1 and u_2 both write to a memory location ℓ_1, while u_3 reads from ℓ_1. Operation v_1 reads from location ℓ_2, and v_2 writes to ℓ_2.

We first redefine the last-writer function to take aborted transactions into account. Intuitively, another transaction should not be able to "see" the values of an aborted transaction.

Definition 5.9. *Consider a trace $(\mathcal{C},\Phi) \in \mathcal{U}$ and a topological sort $\mathcal{S} \in \texttt{tsorts}(\mathcal{G}(\mathcal{C}))$. For all $v \in \texttt{memOps}(\mathcal{C})$ such that $R(v,\ell) \vee W(v,\ell)$, the* ***transactional last writer*** *of v according to $\mathcal{S}$, denoted $\mathcal{X}_{\mathcal{S}}(v)$, is the unique $u \in \texttt{memOps}(\mathcal{C}) \cup \{\texttt{source}(\mathcal{C})\}$ that satisfies four conditions:*

1. *$W(u,\ell)$,*
2. *$u <_{\mathcal{S}} v$,*
3. *$\neg(uHv)$, and*
4. *$\{w \in V(\mathcal{C}) \,:\, (u <_{\mathcal{S}} w <_{\mathcal{S}}) \wedge W(w,\ell) \wedge (\neg wHv)\} = \emptyset$.*

The first two conditions for the transactional last-writer function $\mathcal{X}_{\mathcal{S}}$ are the same as for the last-writer function $\mathcal{L}_{\mathcal{S}}$. The third and fourth conditions of Definition 5.9 parallel the third condition of Definition 5.7, except that now v ignores vertices u or w that write to ℓ but which are hidden from v.

Sequential consistency can now be defined for computations that include transactions. The definition is exactly like Definition 5.8, except that the last-writer function $\mathcal{L}_{\mathcal{S}}$ is replaced by the transactional last-writer function $\mathcal{X}_{\mathcal{S}}$.

Definition 5.10. ***Transactional sequential consistency*** *is the memory model*

$$TSC = \{(\mathcal{C},\Phi) \in \mathcal{U} : \exists \mathcal{S} \in \texttt{tsorts}(\mathcal{G}(\mathcal{C})) \text{ s.t. } \Phi = \mathcal{X}_{\mathcal{S}}\} \ .$$

Serializability

We can also use the transactional computation framework to define serializability, the standard correctness condition for transactional systems. The intuition behind this memory model is that we want to find a single linear order $\mathcal{S}$ on all operations that both "explains" all memory operations and guarantees that every transaction appears to execute atomically, i.e., all transactions appear as contiguous in $\mathcal{S}$.

Definition 5.11. *The **serializability** transactional memory model ST is the set of all traces $(C,\Phi) \in \mathcal{U}$ for which there exists a topological sort $\mathcal{S} \in \texttt{tsorts}(\mathcal{G}(C))$ that satisfies two conditions:*

1. $\Phi = \mathcal{X}_{\mathcal{S}}$, *and*
2. $\forall X \in \texttt{xactions}(C)$ *and* $\forall v \in V(C)$, *we have* $\texttt{source}(X) \leq_{\mathcal{S}} v \leq_{\mathcal{S}} \texttt{sink}(X)$ *implies* $v \in V(X)$.

Informally, an execution belongs to ST if there exists an ordering on all operations $\mathcal{S}$ such that the observer function Φ is the transactional last writer $\mathcal{X}_{\mathcal{S}}$, and for every transaction X, the vertices in $V(X)$ appear contiguous in $\mathcal{S}$.

Prefix-Race Freedom

Although programmers would like to reason about TM systems that guarantee a strong (more restrictive) memory model such as serializability, from the perspective of implementation, it is more efficient to design a TM system that executes operations in an order $\mathcal{S}'$ which itself is not serializable, but which is "equivalent" to an order $\mathcal{S}$ which is serializable. We can describe two seemingly weaker memory models — "race freedom" and "prefix-race freedom" which are designed to model more closely the properties of orderings $\mathcal{S}'$ generated by actual TM implementations. Race freedom weakens serializability by allowing transactions that do not "conflict" to interleave their memory operations in $\mathcal{S}$. Prefix-race freedom weakens race freedom by only prohibiting conflicts with the prefix of a transaction. Prefix-race freedom corresponds more closely to the guarantees provided by traditional TM systems. Sections 5.5 and 5.6 presents an operational model for TM which generates execution orders $\mathcal{S}'$ that can be shown to be prefix-race-free.

The definition of race freedom is motivated by the observation that actual TM implementations allow independent transactions to interleave their executions, as long as one transaction does not try to write to a memory location accessed by the other transaction. With only closed-nested transactions and ignoring operations from aborted transactions, as will be shown in Section 5.4, one can rearrange any interleaved execution order allowed by race freedom into an equivalent serializable order.

To define race freedom, we first consider what it means to have a transactional race between a memory operation and a transaction with respect to a topological sort of the computation DAG.

Definition 5.12. *Let C be a computation tree, and suppose that $\mathcal{S} \in \texttt{tsorts}(\mathcal{G}(C))$ is a topological sort of $\mathcal{G}(C)$. A **(transactional) race** with respect to $\mathcal{S}$ occurs between $v \in V(C)$ and $X \in \texttt{xactions}(C)$, denoted by the predicate $\text{RACE}_{\mathcal{S}}(v,X)$, if $v \notin V(X)$ and there exists a $w \in \texttt{content}(X)$ satisfying the following conditions:*

1. $\neg(vHw)$,
2. $\exists \ell \in \mathcal{M}$ *s.t.* $(R(v,\ell) \wedge W(w,\ell)) \vee (W(v,\ell) \wedge R(w,\ell)) \vee (W(v,\ell) \wedge W(w,\ell))$, *and*
3. $\texttt{source}(X) \leq_{\mathcal{S}} v \leq_{\mathcal{S}} \texttt{sink}(X)$.

The notion of a race is easier to understand when all transactions are committed, in which case no vertices are hidden from each other. Intuitively, a race occurs between transaction X and a vertex $v \notin V(X)$ appearing between $\texttt{source}(X)$ and $\texttt{sink}(X)$ in $\mathcal{S}$ if v conflicts with some vertex $w \in \texttt{content}(X)$ as defined in Definition 5.1.

We can use Definition 5.12 to define race freedom.

Definition 5.13. *The **race-free** transactional memory model RFT is the set of all traces $(\mathcal{C},\Phi) \in \mathcal{U}$ for which there exists a topological sort $\mathcal{S} \in \texttt{tsorts}(\mathcal{G}(\mathcal{C}))$ satisfying two conditions:*

1. $\Phi = X_{\mathcal{S}}$, *and*
2. $\forall v \in V(\mathcal{C})$ *and* $\forall X \in \texttt{xactions}(\mathcal{C})$, $\neg\text{RACE}_{\mathcal{S}}(v,X)$.

The first condition of race freedom is the same as for serializability, that the observer function is the transactional last writer. The second condition allows an operation v to appear between $\texttt{source}(X)$ and $\texttt{sink}(X)$ in $\mathcal{S}$, but only provided no race between v and X exists.

The notion of a prefix race is motivated by the operational semantics of TM systems. As two transactions X and X' execute, if X' discovers a memory-access conflict between a vertex $v \in V(X')$ and X, then the conflict must be with a vertex in X that has already executed, that is, with the prefix of X that executes before v. For prefix-race freedom, no such conflicts may occur.

Definition 5.14. *Let $\mathcal{C}$ be a computation tree, and let $\mathcal{S} \in \texttt{tsorts}(\mathcal{G}(\mathcal{C}))$ be a topological sort of $\mathcal{G}(\mathcal{C})$. A **(transactional) prefix race** with respect to $\mathcal{S}$ occurs between $v \in V(\mathcal{C})$ and $X \in \texttt{xactions}(\mathcal{C})$, denoted by the predicate* $\text{PRACE}_{\mathcal{S}}(v,X)$*, if $v \notin V(X)$ and there exists a $w \in \texttt{content}(X)$ satisfying the following conditions:*

1. $\neg(vHw)$
2. $\exists \ell \in \mathcal{M}$ *s.t.* $(R(v,\ell) \wedge W(w,\ell)) \vee (W(v,\ell) \wedge R(w,\ell)) \vee (W(v,\ell) \wedge W(w,\ell))$.
3. $\texttt{source}(X) \leq_{\mathcal{S}} w <_{\mathcal{S}} v \leq_{\mathcal{S}} \texttt{sink}(X)$.

Thus, this definition is identical to Definition 5.12, except that the potential conflicting vertex w must occur before v in $\mathcal{S}$. In an actual execution, a TM runtime cannot detect all the races that involve v at the time v happens, because v may be in a race because of an operation w that happens after v. A TM runtime can, however, detect all prefix races that involve v.

The notion of a prefix race gives rise to an corresponding memory model in which prefix races are absent.

Definition 5.15. *The **prefix-race free** transactional memory model PRFT is the set of all traces $(\mathcal{C},\Phi) \in \mathcal{U}$ for which there exists a topological sort $\mathcal{S} \in \texttt{tsorts}(\mathcal{G}(\mathcal{C}))$ satisfying two conditions:*

1. $\Phi = X_{\mathcal{S}}$, *and*
2. $\forall v \in V(\mathcal{C})$ *and* $\forall X \in \texttt{xactions}(\mathcal{C})$, $\neg\text{PRACE}_{\mathcal{S}}(v,X)$.

Thus, prefix-race freedom describes a weaker model than race freedom, where a vertex v is only guaranteed to not to conflict with the vertices of transaction X that appear before v in $\mathcal{S}$. If a "nontransactional" leaf node $v \in \texttt{memOps}(\mathcal{C})$ runs in parallel with a transaction X, all of Definitions 5.11, 5.13, and 5.15 check whether v interleaves within X's execution.

Thus, these models can be thought of as guaranteeing "strong atomicity" in the parlance of Blundell, Lewis, and Martin [32]. In Scott's model [111], $\text{RACE}_{\mathcal{S}}(v,X)$ and $\text{PRACE}_{\mathcal{S}}(v,X)$ can be viewed as particular "conflict functions."

Relationships among the Models

The following theorem states that the three memory models as presented are progressively weaker.

Theorem 5.1. *$ST \subseteq RFT \subseteq PRFT$.*

Proof. Follows directly from Definitions 5.11, 5.13, and 5.15. □

Section 5.4 shows that for a TM system which generates computations with only closed-nested and committed transactions, prefix-race freedom and serializability are equivalent. These memory models turn out to be distinct, however, for an open TM system, as described in [8]. Chapter 7 discusses in greater detail some of the issues associated with open nesting.

5.4 Equivalence of Memory Models

This section studies the memory models of serializability, prefix-race freedom, and race freedom. Specifically, this section shows that for computations containing only committed and closed-nested transactions, all three models are equivalent.

Dependency graphs

Before addressing the issue of comparing the memory models, we first present an alternative characterization of sequential consistency for the special case of computations with only committed transactions. The idea of a "dependency" graph is to add edges to the computation DAG to reflect the dependencies imposed by the observer function.

Definition 5.16. *The set of **dependency** edges of a trace $(\mathcal{C},\Phi) \in \mathcal{U}_{\text{com}}$ is*

$$\Psi_d(\mathcal{C},\Phi) = \{(u,v) \in V(\mathcal{C}) \times V(\mathcal{C}) \ : \ u = \Phi(v)\} \ ,$$

*and the set of **antidependency** edges is*

$$\Psi_a(\mathcal{C},\Phi) = \{(u,v) \in V(\mathcal{C}) \times V(\mathcal{C}) \ : \ (\Phi(u) = \Phi(v)) \wedge W(v,\ell)\}$$

*The **dependency graph** of $(\mathcal{C},\Phi)$ is the graph $\mathcal{DG}(\mathcal{C},\Phi) = (V,E)$, where $V = V(\mathcal{C})$ and $E = E(\mathcal{C}) \cup \Psi_d(\mathcal{C},\Phi) \cup \Psi_a(\mathcal{C},\Phi)$.*

The sets Ψ_d and Ψ_a capture the usual notions of dependency and antidependency edges from the study of compilers [83]. A dependency edge (u,v) indicates that v observed the value written by u. An antidependency edge (u,v) means that if both u and v observe the same write to a location ℓ, and if v performs a write, then u must "come before" v.

The following lemma shows that in the universe of all traces with only committed transactions, a trace $(\mathcal{C},\Phi)$ is sequentially consistent if and only if the dependency graph $\mathcal{DG}(\mathcal{C},\Phi)$ is acyclic.[4]

Lemma 5.2. *Suppose that $(\mathcal{C},\Phi) \in \mathcal{U}_{\mathrm{com}}$. Then we have $(\mathcal{C},\Phi) \in SC$ if and only if the dependency graph $\mathcal{DG}(\mathcal{C},\Phi)$ is acyclic.* □

Proof. First, suppose that $(\mathcal{C},\Phi) \in SC$. Then there exists a topological sort $\mathcal{S}$ of $\mathcal{G}(\mathcal{C})$ such that $\Phi = \mathcal{L}_{\mathcal{S}}$. From the definition of Ψ_d and Ψ_a, since Φ is a last-writer function according to $\mathcal{S}$, any dependency or antidependency edge (u,v) that is added to $\mathcal{DG}(\mathcal{C},\Phi)$ satisfies $u <_{\mathcal{S}} v$. Thus, $\mathcal{S}$ is a valid topological sort of $\mathcal{DG}(\mathcal{C},\Phi)$, and $\mathcal{DG}(\mathcal{C},\Phi)$ is acyclic.

Next, suppose that $\mathcal{DG}(\mathcal{C},\Phi)$ is acyclic. Then there exists a topological sort $\mathcal{S}$ of $\mathcal{DG}(\mathcal{C},\Phi)$. Since $\mathcal{G}(\mathcal{C})$ has the same vertices but fewer edges than $\mathcal{DG}(\mathcal{C},\Phi)$, $\mathcal{S}$ is also a topological sort of $\mathcal{G}(\mathcal{C})$. Thus, all that remains to show that $(\mathcal{C},\Phi) \in SC$ is to demonstrate that $\Phi = \mathcal{L}_{\mathcal{S}}$.

Suppose for contradiction that $\Phi \neq \mathcal{L}_{\mathcal{S}}$. Then let $u \in \texttt{memOps}(\mathcal{C})$ be the first memory operation in the order $\mathcal{S}$ for which $\Phi(u) \neq \mathcal{L}_{\mathcal{S}}(u)$, and let ℓ be the memory location accessed by u. We know that $W(\Phi(u),\ell)$ and that there must exist a memory operation w such that $\Phi(u) <_{\mathcal{S}} w <_{\mathcal{S}} u$ and $W(w,\ell)$. Let v be the first node in the order $\mathcal{S}$ that satisfies $W(v,\ell)$ and $\Phi(u) <_{\mathcal{S}} v \leq_{\mathcal{S}} w$, i.e., v is the first write to ℓ that happens after $\Phi(u)$. Since $w <_{\mathcal{S}} u$ and u is the first operation in $\mathcal{S}$ for which Φ contradicts the last writer property, it follows that $\Phi(v) = \mathcal{L}_{\mathcal{S}}(v) = u$. Thus, we have $\Phi(v) = \Phi(u)$, and $\mathcal{DG}(\mathcal{C},\Phi)$ must have an antidependency edge (u,v). This edge creates a contradiction, since it implies that the order $\mathcal{S}$ is not a valid topological sort of $\mathcal{DG}(\mathcal{C},\Phi)$. Therefore, we must have $\Phi = \mathcal{L}_{\mathcal{S}}$. □

Figure 5-10 shows the dependency graphs for the example traces from Figure 5-9. The trace $(\mathcal{C},\Phi_1)$ is sequentially consistent, but the trace $(\mathcal{C},\Phi_2)$ is not. Equivalently by Lemma 5.2, the dependency graph $\mathcal{DG}(\mathcal{C},\Phi_1)$ is acyclic, but the graph $\mathcal{DG}(\mathcal{C},\Phi_2)$ is not.

We can now prove the equivalence of serializability, race freedom, and prefix-race freedom when we consider only computations with committed and closed-nested transactions.

Theorem 5.3. $ST \cap \mathcal{U}_{\mathrm{com}} = RFT \cap \mathcal{U}_{\mathrm{com}} = PRFT \cap \mathcal{U}_{\mathrm{com}}$.

Proof. Since Theorem 5.1 shows that $ST \subseteq RFT \subseteq PRFT$, it suffices to prove that

$$PRFT \cap \mathcal{U}_{\mathrm{com}} \subseteq ST \cap \mathcal{U}_{\mathrm{com}}.$$

We start by defining some terminology. For all $u \in V(\mathcal{C})$, define the ***holder count*** of u as $\lambda(u) = |\texttt{holders}(u)|$. For $u,v \in V(\mathcal{C})$, define the ***alternation count*** of u and v as

$$alt(u,v) = \lambda(u) + \lambda(v) - 2\lambda(\texttt{LCA}(u,v)) .$$

Thus, $alt(u,v)$ counts the number of transactions $X \in \texttt{xactions}(\mathcal{C})$ that contain either u or v, but not both. For any topological sort $\mathcal{S}$ of $\mathcal{G}(\mathcal{C})$, define the ***alternation count*** of $\mathcal{S}$,

[4] One must extend the definition of an antidependency edge to prove an analogous result when the computation $\mathcal{C}$ has aborted transactions. Lemma 5.2 requires the assumption that every write to a location also performs a read.

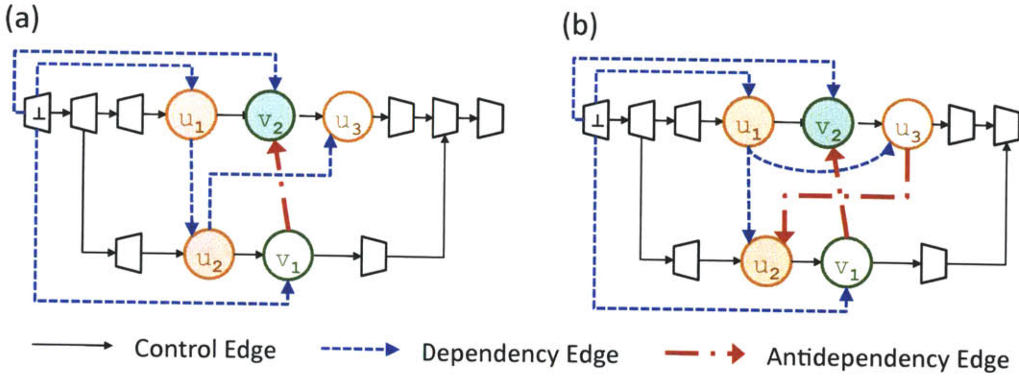


Figure 5-10: Dependency graphs for the traces from Figure 5-9. (a) Since the graph $\mathcal{DG}(C,\Phi_1)$ is acyclic, we have $(C,\Phi_1) \in SC$. (b) Since $\mathcal{DG}(C,\Phi_2)$ has a cycle, we have $(C,\Phi_2) \notin SC$.

denoted $alt(\mathcal{S})$, as the sum of all $alt(u,v)$ for consecutive u and v in $\mathcal{S}$. Intuitively, $alt(\mathcal{S})$ counts the number of times we "switch" between transactions as we run through $\mathcal{S}$.

We prove by contradiction that for any trace $(C,\Phi) \in \mathcal{U}_{\text{com}}$, we have that $(C,\Phi) \in PRFT$ implies $(C,\Phi) \in ST$. Suppose that there exists a trace $(C,\Phi) \in \mathcal{U}_{\text{com}}$ that is prefix-race-free but not serializable. Consider a topological sort $\mathcal{S} \in \texttt{tsorts}(\mathcal{DG}(C,\Phi))$ that is both prefix-race-free, and has a minimum alternation count $alt(\mathcal{S})$ over all prefix-race-free sorts in $\texttt{tsorts}(\mathcal{DG}(C,\Phi))$. By Lemma 5.2, $\mathcal{S}$ satisfies the condition $\Phi = \mathcal{X}_{\mathcal{S}}$ (the first condition for all three transactional models).

Since $(C,\Phi) \notin ST$, some transaction X exists that is not contiguous in $\mathcal{S}$ (and therefore violates the second condition in Definition 5.11). Let v_1 be the first vertex in $\mathcal{S}$ such that there exists a transaction X with $v_1 \notin V(X)$ and $\texttt{source}(X) <_{\mathcal{S}} v <_{\mathcal{S}} \texttt{sink}(X)$. We can find three consecutive intervals in the sort $\mathcal{S}$, namely $A_1 = [u_1,u_2]$, $A_2 = [v_1,v_2]$, and $A_3 = [w_1,w_2]$, satisfying the following properties:

- $u_1 = \texttt{source}(X)$,
- $A_1 \subseteq V(X)$,
- $A_2 \cap V(X) = \emptyset$, and
- $A_3 \subseteq V(X)$.

In other words, A_1 and A_3 contain only nodes from $V(X)$, and A_2 contains only nodes outside $V(X)$. Thus, A_2 is an interval in $\mathcal{S}$ interleaved between contiguous fragments of X. Figure 5-11(a) shows these three intervals in the topological sort $\mathcal{S}$. Let t be the node that immediately precedes $\texttt{source}(X)$ in $\mathcal{S}$.

From $\mathcal{S}$, we construct the new order $\mathcal{S}'$ shown in Figure 5-11(b) in which the intervals A_1 and A_2 are interchanged. We shall show that

1. $\mathcal{S}' \in \texttt{tsorts}(\mathcal{DG}(C,\Phi))$ (and therefore $\Phi = \mathcal{X}_{\mathcal{S}'}$),
2. $\mathcal{S}'$ is still a prefix-race-free topological sort of $\mathcal{DG}(C,\Phi)$, and
3. $alt(\mathcal{S}') < alt(\mathcal{S})$.

The last condition leads to the contradiction that $\mathcal{S}$ is not a prefix-race-free topological sort with minimum alternation count.

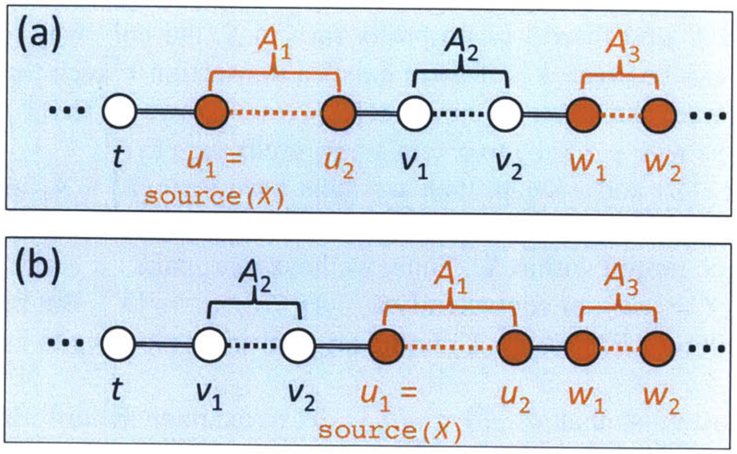


Figure 5-11: Two topological sorts of a computation graph $\mathcal{G}(\mathcal{C})$ for a hypothetical trace $(\mathcal{C}, \Phi)$ which is prefix-race-free, but not serializable. Transaction X is not contiguous in the topological sort $\mathcal{S}$ in (a). One can convert $\mathcal{S}$ into the topological sort $\mathcal{S}'$ in (b). Doing so reduces the alternation count.

More specifically, we prove each fact as follows:

1. To establish that $\mathcal{S}' \in \texttt{tsorts}(\mathcal{DG}(\mathcal{C},\Phi))$, we show that no edge (y,z) where $y \in A_1$ and $z \in A_2$ belongs to the graph $\mathcal{DG}(\mathcal{C},\Phi)$. Suppose for contradiction that (y,z) is an edge in $\mathcal{DG}(\mathcal{C},\Phi)$. Consider the possibilities for the edge (y,z):
   (a) If we have $(y,z) \in \Psi_d(\mathcal{C},\Phi) \cup \Psi_a(\mathcal{C},\Phi)$, then y and z access the same memory location ℓ and one of those accesses is a write. We have $y \in V(X) = \texttt{content}(X)$, since we are only considering traces with committed transactions. But then we would have $\text{PRACE}_{\mathcal{S}}(z,X)$, since $y \in \texttt{content}(X)$, $z \notin V(X)$, and $\texttt{source}(X) <_{\mathcal{S}} y <_{\mathcal{S}} z <_{\mathcal{S}} \texttt{sink}(X)$. This prefix-race contradicts the assumption that $(\mathcal{C},\Phi) \in \mathit{PRFT}$.
   (b) Alternatively, if we have $(y,z) \in E(\mathcal{C})$, then $\texttt{LCA}(y,z)$ must be an S-node with y to the left of z. Since $z \notin V(X)$, we have $\texttt{LCA}(y,z) = \texttt{LCA}(X,z)$. Therefore, we must have $\texttt{sink}(X) \prec_{\mathcal{G}(\mathcal{C})} z$. Thus, $\mathcal{S}$ was not a valid sort of $\mathcal{DG}(\mathcal{C},\Phi)$.

2. Next, we establish that $\mathcal{S}'$ is prefix-race-free by showing that swapping A_1 and A_2 cannot introduce any prefix races that weren't already in $\mathcal{S}$.

   Suppose that $\mathcal{S}'$ contains a prefix-race $\text{PRACE}_{\mathcal{S}}(v,Z)$, i.e., there exists a transaction $Z \in \texttt{xactions}(\mathcal{C})$ and a vertex $v \in V(\mathcal{C}) - V(Z)$ satisfying all three conditions of Definition 5.14 for $\mathcal{S}'$. Let $w \in \texttt{content}(Z)$ be the candidate vertex that satisfies the three conditions. In particular, the third condition gives us $\texttt{source}(Z) <_{\mathcal{S}'} w <_{\mathcal{S}'} v <_{\mathcal{S}'} \texttt{sink}(Z)$. We consider two cases, each of which leads to a contradiction.
   (a) In the first case, suppose that $v <_{\mathcal{S}} w$. Since v and w swap in the two orders, we must have $v \in A_1$ and $w \in A_2$, which implies that $v \in V(X)$ and $w \notin V(X)$. We also have $\texttt{source}(X) <_{\mathcal{S}} v <_{\mathcal{S}} w <_{\mathcal{S}} \texttt{sink}(X)$. But then in $\mathcal{S}$, we would have $\text{PRACE}_{\mathcal{S}}(v,X)$, which is a contradiction since there were no prefix races in the original ordering.

(b) In the second case, suppose that $w <_{\mathcal{S}} v$. Since w comes before v in both sorts $\mathcal{S}$ and $\mathcal{S}'$ and there was no prefix race in $\mathcal{S}$, the only way a new prefix race in $\mathcal{S}'$ can be created is if there exists a transaction Y such that $\texttt{source}(Y) <_{\mathcal{S}} w <_{\mathcal{S}} \texttt{sink}(Y) <_{\mathcal{S}} v$, but $\texttt{source}(Y) <_{\mathcal{S}'} w <_{\mathcal{S}'} v <_{\mathcal{S}'} \texttt{sink}(Y)$. In other words, v swaps with $\texttt{sink}(Y)$ to create a new prefix race in $\mathcal{S}'$.
If $\texttt{sink}(Y)$ and v swap, then we must have $\texttt{sink}(Y) \in A_1$ and $v \in A_2$. Then since $A_1 \subseteq \texttt{content}(X)$, it follows that $\texttt{sink}(Y) \in \texttt{content}(X)$, and thus Y must be nested within X. Thus, we have $\texttt{source}(X) <_{\mathcal{S}} \texttt{source}(Y) <_{\mathcal{S}} w <_{\mathcal{S}} \texttt{sink}(Y) \leq_{\mathcal{S}} u_2$, or equivalently $w \in A_1 \subseteq \texttt{content}(X)$. But then w must create a prefix race $\text{PRACE}_{\mathcal{S}}(v,X)$ in the original sort $\mathcal{S}$, leading to a contradiction.

3. Finally, to establish that $alt(\mathcal{S}') < alt(\mathcal{S})$, let us examine the difference $\delta = alt(\mathcal{S}) - alt(\mathcal{S}')$ in the alternation counts of $\mathcal{S}$ and $\mathcal{S}'$. The only terms that contribute to δ are at the boundaries of A_1 and A_2. We have that

$$\begin{aligned}
\delta &= alt(t,u_1) + alt(u_2,v_1) + alt(v_2,w_1) - alt(t,v_1) - alt(v_2,u_1) - alt(u_2,w_1) \\
&= -2\lambda(\texttt{LCA}(t,u_1)) - 2\lambda(\texttt{LCA}(u_2,v_1)) - 2\lambda(\texttt{LCA}(v_2,w_1)) \\
&\quad + 2\lambda(\texttt{LCA}(t,v_1)) + 2\lambda(\texttt{LCA}(v_2,u_1)) + 2\lambda(\texttt{LCA}(u_2,w_1)) \,.
\end{aligned}$$

By construction, we know that $\{u_1,u_2,w_1,w_2\} \subseteq V(X)$, whereas none of t, v_1, and v_2 have X as an ancestor. For any $y \in V(X)$ and $z \notin V(X)$, we know $\texttt{LCA}(y,z) = \texttt{LCA}(X,z)$, which implies that $\texttt{LCA}(v_2,u_1) = \texttt{LCA}(v_2,w_1) = \texttt{LCA}(v_2,X)$, $\texttt{LCA}(t,u_1) = \texttt{LCA}(t,X)$, and $\texttt{LCA}(u_2,v_1) = \texttt{LCA}(X,v_1)$. Simplifying the expression for δ, we get

$$\delta = -2\lambda(\texttt{LCA}(t,X)) - 2\lambda(\texttt{LCA}(X,v_1)) + 2\lambda(\texttt{LCA}(t,v_1)) + 2\lambda(\texttt{LCA}(u_2,w_1)) \,.$$

To show that $\delta > 0$, let $Y = \texttt{LCA}(t,v_1)$, and consider the two cases for Y, which are illustrated in Figure 5-12.

(a) Suppose that Y is not an ancestor of X. Then, as shown in Figure 5-12(a), we have

$$\begin{aligned}
\delta &= -h - h + (g+h) + (c+d+h) \\
&= c+d+g \,.
\end{aligned}$$

We know $c+d > 0$ since it contains the transaction X, and hence $\delta > 0$.

(b) Suppose that Y is an ancestor of X, and suppose without loss of generality that $\texttt{LCA}(t,X)$ is a descendant of Y. (The other case, where t and v_1 are swapped, is symmetric.) In this case, as shown in Figure 5-12(b), we have

$$\begin{aligned}
\delta &= -f - 0 + 0 + (c+d+f) \\
&= c+d \\
&> 0 \,.
\end{aligned}$$

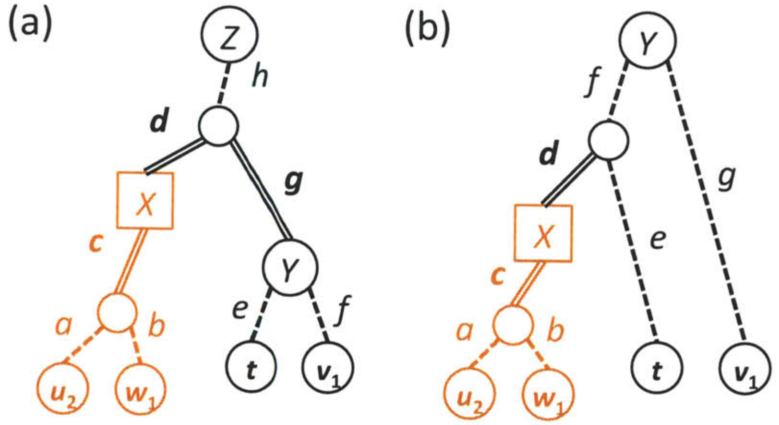


Figure 5-12: Two cases for alternation count in proof of Theorem 5.3. Let $Y = \texttt{LCA}(t, v_1)$. (a) Y is not an ancestor of X, and we have $\delta = c + d + g$. (b) Y is an ancestor of X, and we have $\delta = c + d$.

□

5.5 The TCO Model

This section describes the ***transactional-computation operational model*** or (***TCO model*** for short), an operational semantics for a TM system with nested parallelism. The TCO model extends the framework described in Section 5.2 to construct computation trees dynamically as a program executes.

Conceptually, the TCO model is a nondeterministic state machine with two components: a ***program*** and a ***runtime system***. The runtime system dynamically constructs and traverses a computation tree $\mathcal{C}$ as it executes instructions generated by the program. The TCO runtime maintains a set of ***ready*** nodes, denoted by $\texttt{ready}^{(t)}(\mathcal{C}) \subseteq \texttt{nodes}^{(t)}(\mathcal{C})$, and at every ***step*** t, the TCO model nondeterministically chooses one of these ready nodes $X \in \texttt{ready}^{(t)}(\mathcal{C})$ to issue the next instruction. The program then issues an instruction on X's behalf. For shorthand, we sometimes say that X issues an instruction.

The TCO model extends the definitions and notation for a computation tree $\mathcal{C}$ to allow the tree $\mathcal{C}$ to change as steps execute. Generalizing the definition of computation-tree nodes $\texttt{nodes}(\mathcal{C})$, define the set $\texttt{nodes}^{(t)}(\mathcal{C})$ as the dynamic set of tree nodes in $\mathcal{C}$ after taking step t. Conceptually, as a program executes, nodes are added but never deleted from a computation tree $\mathcal{C}$. Similarly, for each of the subsets of $\texttt{nodes}(\mathcal{C})$ defined earlier (e.g., $\texttt{sNodes}(\mathcal{C})$, $\texttt{xactions}(\mathcal{C})$, etc.), define a step-dependent set (i.e., $\texttt{sNodes}^{(t)}(\mathcal{C})$, $\texttt{xactions}^{(t)}(\mathcal{C})$, etc.) representing a subset of computation-tree nodes after taking a particular step t. Usually, these sets monotonically increase as the step count increases, i.e. for all t, $\texttt{nodes}^{(t)}(\mathcal{C}) \subseteq \texttt{nodes}^{(t+1)}(\mathcal{C})$.

The TCO model also maintains a ***status field*** for each $X \in \texttt{nodes}^{(t)}(\mathcal{C})$ at each step t, denoted by $\texttt{status}^{(t)}(X)$, which stores runtime information about X. Nodes that are not transactions have a status which is either RUNNING or DONE. A RUNNING node X is a node that is either currently executing or an ancestor of a currently executing node. A node X is $\mathtt{DONE}$ if the program has finished executing X. For a transaction node $X \in \mathtt{xactions}(C)$, instead of having a status of $\mathtt{RUNNING}$, we say that a transaction that is currently in progress has $\mathtt{status}(X)$ that is either $\mathtt{PENDING}$, or $\mathtt{PENDING\_ABORT}$, depending on whether that transaction should eventually be committed or aborted. Similarly, instead of X having a status of $\mathtt{DONE}$, we have that $\mathtt{status}(X)$ is either $\mathtt{COMMITTED}$ or $\mathtt{ABORTED}$, depending on whether X completes successfully or not.

From these status fields, we can define several sets:[5]

$$\begin{aligned}
\mathtt{RUNNING}^{(t)}(C) &= \left\{X \in \mathtt{nodes}^{(t)}(C) \ : \ \mathtt{status}^{(t)}(X) = \mathtt{RUNNING}\right\} \\
\mathtt{DONE}^{(t)}(C) &= \left\{X \in \mathtt{nodes}^{(t)}(C) \ : \ \mathtt{status}^{(t)}(X) = \mathtt{DONE}\right\} \\
\mathtt{PENDING}^{(t)}(C) &= \left\{X \in \mathtt{xactions}^{(t)}(C) \ : \ \mathtt{status}^{(t)}(X) = \mathtt{PENDING}\right\} \\
\mathtt{PENDING\_ABORT}^{(t)}(C) &= \left\{X \in \mathtt{xactions}^{(t)}(C) \ : \ \mathtt{status}^{(t)}(X) = \mathtt{PENDING\_ABORT}\right\} \\
\mathtt{COMMITTED}^{(t)}(C) &= \left\{X \in \mathtt{nodes}^{(t)}(C) \ : \ \mathtt{status}^{(t)}(X) = \mathtt{COMMITTED}\right\} \\
\mathtt{ABORTED}^{(t)}(C) &= \left\{X \in \mathtt{nodes}^{(t)}(C) \ : \ \mathtt{status}^{(t)}(X) = \mathtt{ABORTED}\right\} \\
\mathtt{vTree}^{(t)}(C) &= \mathtt{RUNNING}^{(t)}(C) \cup \mathtt{PENDING}^{(t)}(C) \cup \mathtt{PENDING\_ABORT}^{(t)}(C)
\end{aligned}$$

On a step t, the set $\mathtt{vTree}^{(t)}(C)$ corresponds to the ***active tree***, the portion of C which is active.

We can organize the instructions of the TCO model into three main groups. The ***control-flow instructions*** — spawn (spawn of a function), sync (sync in a function), and sReturn (return of a spawned function) — describe fork-join parallelism. The ***memory operations*** — read and write— model accesses to memory. Finally, the ***transaction-control*** instructions — xbegin (begin a transaction), xend (commit the current transaction), xabort (abort the current transaction), and sigabort (signal a transaction abort) — model the beginning and end of transactions.

The TCO model describes only a sequential semantics, that is, it assumes at every step t, a program issues a single instruction. The parallelism in this model arises from the fact that on a particular step, several nodes can be ready, and the runtime nondeterministically chooses which node to issue an instruction. Thus, the TCO model can simulate the execution of the computation on any number of processors. In practice, an actual runtime system would need some synchronization to ensure that the work performed by individual instructions in the TCO model appear to happen atomically.

Transaction Readsets and Writesets

To model conflict detection and transaction commits in a TM system, the TCO model maintains readsets and writesets for transactions. More precisely, on a step t, for all $X \in \mathtt{xactions}^{(t)}(C) \cap \mathtt{vTree}^{(t)}(C)$ (i.e., for every active transaction X), let $\mathtt{R}^{(t)}(X)$ denote the

[5] On the last step t, we have $\mathtt{COMMITTED}^{(t)}(C) = \mathtt{committed}(C)$ and $\mathtt{ABORTED}^{(t)}(C) = \mathtt{aborted}(C)$.

readset of X. The readset $\mathtt{R}^{(t)}(X)$ is a set of pairs (ℓ, v), where $\ell \in \mathcal{M}$ is a memory location and $v \in \mathtt{memOps}(\mathcal{C})$ is a memory operation that reads from ℓ, i.e., $R(v, \ell)$. Similarly, let $\mathtt{W}^{(t)}(X)$ denote the ***writeset*** of X, i.e., a set of pairs (ℓ, v) such that $W(v, \ell)$.

The TCO model maintains two invariants on transaction readsets and writesets. First, $\mathtt{W}^{(t)}(X) \subseteq \mathtt{R}^{(t)}(X)$ for every transaction $X \in \mathtt{xactions}^{(t)}(\mathcal{C})$, i.e., a write also counts as a read. Second, $\mathtt{R}^{(t)}(X)$ and $\mathtt{W}^{(t)}(X)$ each contain at most one pair (ℓ, v) for any location ℓ. We use the shorthand $\ell \in \mathtt{R}^{(t)}(X)$ to mean that there exists a node u such that $(\ell, u) \in \mathtt{R}^{(t)}(X)$, and similarly for $\mathtt{W}^{(t)}(X)$.

We also overload the union operator: at some step t, an operation

$$\mathtt{R}(X) \leftarrow \mathtt{R}(X) \cup \{(\ell, u)\}$$

means we construct $\mathtt{R}^{(t+1)}(X)$ by

$$\mathtt{R}^{(t+1)}(X) \leftarrow \{(\ell, u)\} \cup \left(\mathtt{R}^{(t)}(X) - \left\{(\ell, u') \in \mathtt{R}^{(t)}(X)\right\}\right).$$

In other words, we add (ℓ, u) to $\mathtt{R}(X)$, replacing any $(\ell, u') \in \mathtt{R}^{(t)}(X)$ that might have existed previously.

The TCO model initializes readsets and writesets at the beginning of transactions, merges them on the commit of transactions, and (conceptually) discards them on the abort of transactions. When the TCO model begins a transaction X, X begins with an empty readset and writeset, i.e., $\mathtt{R}(X) = \mathtt{W}(X) = \emptyset$. If transaction X commits, since we are considering closed-nested transactions X, $\mathtt{R}(X)$ and $\mathtt{W}(X)$ are merged into the $\mathtt{R}(\mathtt{xParent}(X))$ and $\mathtt{W}(\mathtt{xParent}(X))$, respectively. If X aborts, however, $\mathtt{R}(X)$ and $\mathtt{W}(X)$ are implicitly emptied when X is finished, since readsets and writesets are only defined for active transactions.

For convenience, we assume that $\mathtt{root}(\mathcal{C}) \in \mathtt{xactions}(\mathcal{C})$, i.e., the root of the computation tree is a transaction. Conceptually, this transaction is a PENDING transaction that never conflicts with any other transaction, and thus conceptually "commits" when program execution ends. We also represent main memory as the readset/writeset of the root $\mathtt{root}(\mathcal{C})$. At step $t = 0$, we assume $\mathtt{R}^{(0)}(\mathtt{root}(\mathcal{C}))$ and $\mathtt{W}^{(0)}(\mathtt{root}(\mathcal{C}))$ initially contain a pair $(\ell, \perp)$ for all locations $\ell \in \mathcal{M}$, that is, it has an initial value for all of memory.

Finally, to precisely define transaction conflicts, it is useful to consider on any given step t, the set of active transactions that have a particular memory location ℓ in their readset or writeset. Define the ***readers*** of ℓ as

$$\mathtt{readers}^{(t)}(\ell) = \left\{X \in \mathtt{xactions}^{(t)}(\mathcal{C}) \ : \ \exists v \in \mathtt{memOps}(\mathcal{C}) \text{ s.t. } (\ell, v) \in \mathtt{R}^{(t)}(X)\right\} .$$

Similarly, define the ***writers*** of ℓ as

$$\mathtt{writers}^{(t)}(\ell) = \left\{X \in \mathtt{xactions}^{(t)}(\mathcal{C}) \ : \ \exists v \in \mathtt{memOps}(\mathcal{C}) \text{ s.t. } (\ell, v) \in \mathtt{W}^{(t)}(X)\right\} .$$

The TCO model uses transaction readsets and writesets to check for conflicts between transactions. In this model, a ready node Z can try to issue a `read` or `write` instruction to attempt to perform a memory operation v, which may create v as a child of Z in the computation tree $\mathcal{C}$. Before successfully completing v and creating a node in $\mathcal{C}$, however, the TM runtime checks for conflicts according to the rules given in the following definition.

Definition 5.17. *Suppose that on a step t, the TCO model issues a* `read` *or* `write` *instruction that attempts to create a memory operation v. We say that v generates a* ***memory conflict*** *if there exists a location $\ell \in \mathcal{M}$ and an active transaction $X \in \texttt{vTree}^{(t)}(\mathcal{C})$ such that $X \notin \texttt{ances}(v)$, and either*

1. *$R(v,\ell)$ and $X \in \texttt{writers}^{(t)}(\ell)$, or*
2. *$W(v,\ell)$ and $X \in \texttt{readers}^{(t)}(\ell)$.*

A memory operation v that generates a conflict triggers the abort of one or more transactions until the conflict with v is resolved. To resolve a conflict, the TCO model can nondeterministically choose between either aborting $\texttt{xParent}(v)$ (the transaction that contains v), or the transaction $X \notin \texttt{ances}(v)$ from Definition 5.17 that is causing a conflict (and any descendants of X).

A memory operation v that does not generate a conflict updates the readset and/or writeset of its enclosing transaction, $\texttt{xParent}(v)$. To set $\Phi(v)$, the observer function for v, the TCO model takes the value (ℓ,u) from the readset of the closest ancestor transaction of v that contains ℓ. More formally, we choose (ℓ,u) from the readset $\texttt{R}(Y)$, where $Y = \texttt{leaf}(\texttt{readers}^{(t)}(\ell) \cap \texttt{ances}(v))$.

The TCO Model

The TCO model dynamically constructs a computation tree $\mathcal{C}$ by nondeterministically choosing some ready S-node $Z \in \texttt{ready}^{(t)}(\mathcal{C})$ on each step t, and having Z execute an instruction on that step.[6]

We can organize the instructions of the TCO model into three main groups: (1) control flow instructions (`spawn`, `sync`, `sReturn`) to describe control flow, (2) memory operations (`read`, `write`), (3) transaction control (`xbegin`, `xend`, `sigabort`, and `xabort`).

The `spawn` and `sync` instructions in the TCO model are analogous to `spawn` and `sync` statements in a fork-join parallel language. For simplicity, we assume there are no called functions in the TCO model, i.e., all called functions are inlined.

1. **spawn**: A spawn of a function F creates a new P-node Y as a child of X with Y having two task nodes as children, namely a left child Z_1 and a right child Z_2. The left child Z_1 also creates a single function node child $F \in \texttt{functions}(\mathcal{C})$ corresponding

[6] The TCO model is similar to the *CCT* model described in Section A.4, except that it incorporates transactions. First, the TCO model replaces the primitive operation (`primOp`) instruction with one of two instructions — a `read` or `write` — corresponding to memory operations. Next, instead of generic `call`/`cReturn` instructions for the call/return of a function F, the TCO model has `xbegin` and `xend` instructions which correspond to the begin and end, respectively, of a transaction. Finally, the TCO model eliminates the notion of a task node with status of `QUEUED`, as well as deques or producer nodes for a worker. The *CCT* model maintains these quantities because it assumes a specific runtime scheduler with P worker threads and a Cilk-like work-stealing algorithm. In contrast, instead of changing producer nodes and having `QUEUED` nodes, the TCO model changes $\texttt{ready}^{(t)}(\mathcal{C})$, the set of ready nodes. While the *CCT* model has $\texttt{producerNodes}^{(t)}(\mathcal{C}) \cup \texttt{QUEUED}^{(t)}(\mathcal{C})$ being the set of leaves of $\texttt{vTree}^{(t)}(\mathcal{C})$, the TCO model has $\texttt{ready}^{(t)}(\mathcal{C})$ as the leaves of $\texttt{vTree}^{(t)}(\mathcal{C})$.

to the spawned function. All the nodes (Y,Z_1, Z_2, and F) are created with status RUNNING, and F and Z_2 are added to ready($\mathcal{C}$), i.e.,

$$\texttt{ready}^{(t+1)}(\mathcal{C}) = (\texttt{ready}^{(t)}(\mathcal{C}) - X) \cup \{F, Z_2\} .$$

2. **sync**: A task node $X \in$ tasks($\mathcal{C}$) executes a sync instruction to finish a task X by setting status(X) to DONE and then removing X from ready($\mathcal{C}$). Let Q be the ***sync block*** that contains X, i.e., the set containing all tasks which wait on a particular sync statement, plus their parent P-nodes.[7] Let root(Q) denote the root of Q. Then, the sync behaves differently depending on whether X's sync block is now complete.

   (a) If status(Z) $=$ DONE for all tasks $Z \in Q$, then the sync instruction sets the status of all remaining nodes $Y \in Q$ to DONE, and adds parent(root(Q)) back into ready($\mathcal{C}$).

   (b) Otherwise, some task $Z \in Q$ is still executing, and the instruction does not change ready($\mathcal{C}$).

3. **sReturn**: If X is a spawned function, then X can execute a return instruction, which sets status(X) to DONE. The return from a spawn removes X from ready($\mathcal{C}$), conceptually adds parent(X) (which is a task node) to ready($\mathcal{C}$). Finally, the completion of the sReturn instruction immediately triggers a sync for the task parent(X).

The read and write instructions in the TCO model perform memory operations.

4. **read**: A ready node Z can issue this instruction to attempt a read of a memory location ℓ. If this read does not generate a conflict (as described in Definition 5.17), then it creates a memory operation v with $R(v, \ell)$ and makes the following changes:

   (a) adds v to the computation tree, i.e., adds v to memOps($\mathcal{C}$) with parent(v) $= Z$,

   (b) sets $\Phi(v) = u$, where u is the unique operation such that

   $$(\ell, u) \in \texttt{R}(\texttt{leaf}(\texttt{readers}^{(t)}(\ell) \cap \texttt{ances}(v))) ,$$

   (c) updates R(xParent(v)), i.e., R(xParent(v)) $\leftarrow$ R(xParent(v)) $\cup \{(\ell, v)\}$.

5. **write**: A ready node Z can issue this instruction to attempt a write of a memory location ℓ. If this write does not generate a conflict, then it creates a memory operation v with $W(v, \ell)$ and makes the following changes:

   (a) adds v to the computation tree, i.e., adds v to memOps($\mathcal{C}$) with parent(v) $= Z$,

   (b) sets $\Phi(v) = u$, where u is the unique operation such that

   $$(\ell, u) \in \texttt{R}(\texttt{leaf}(\texttt{readers}^{(t)}(\ell) \cap \texttt{ances}(v))) ,$$

[7] For a more formal definition of a sync block, see Definition A.2 in Section A.4.

(c) updates $\mathtt{R}(\mathtt{xParent}(v))$ and $\mathtt{W}(\mathtt{xParent}(v))$, i.e.,

$$\begin{aligned}\mathtt{R}(\mathtt{xParent}(v)) &\leftarrow \mathtt{R}(\mathtt{xParent}(v)) \cup \{(\ell, v)\} \\ \mathtt{W}(\mathtt{xParent}(v)) &\leftarrow \mathtt{W}(\mathtt{xParent}(v)) \cup \{(\ell, v)\} .\end{aligned}$$

The final group of instructions handles the beginning and end of transactions.

6. **xbegin**: A ready node Z executes this instruction to begin a new transaction X. In the computation tree, we replace Z with X in $\mathtt{ready}(\mathcal{C})$, that is,

$$\mathtt{ready}^{(t+1)}(\mathcal{C}) = (\mathtt{ready}^{(t)}(\mathcal{C}) - \{Z\}) \cup \{X\} .$$

   The new transaction X begins with an empty readset and writeset, i.e.,

$$\mathtt{R}^{(t+1)}(X) = \emptyset, \quad \mathtt{W}^{(t+1)}(X) = \emptyset.$$

7. **xend**: A node $X \in \mathtt{ready}(\mathcal{C}) \cap \mathtt{xactions}(\mathcal{C})$ with $\mathtt{status}(X) = \mathtt{PENDING}$ can issue this instruction to commit the transaction X. A commit of X merges the readset and writeset of X into that of its transactional parent, i.e.,

$$\begin{aligned}\mathtt{R}(\mathtt{xParent}(X)) &\leftarrow \mathtt{R}(\mathtt{xParent}(X)) \cup \mathtt{R}(X) \\ \mathtt{W}(\mathtt{xParent}(X)) &\leftarrow \mathtt{W}(\mathtt{xParent}(X)) \cup \mathtt{W}(X)\end{aligned}$$

8. **sigabort**: A read or write that causes a conflict will trigger a sigabort of one or more transactions X. This instruction changes $\mathtt{status}(X)$ from PENDING to PENDING_ABORT. If $X \in \mathtt{ready}(\mathcal{C})$, then after the sigabort of X, X can be immediately issue an xabort.

9. **xabort**: A node $X \in \mathtt{ready}(\mathcal{C}) \cap \mathtt{xactions}(\mathcal{C})$ (a ready node which is a transaction) with $\mathtt{status}(X) = \mathtt{PENDING\_ABORT}$ can issue this instruction, which changes $\mathtt{status}(X)$ to ABORTED, removes X from $\mathtt{ready}(\mathcal{C})$, and then adds $\mathtt{parent}(X)$ to $\mathtt{ready}(\mathcal{C})$.

For a TM system with ordinary, closed-nested transactions, a transaction conflict on a transaction X conceptually issues sigabort instructions for all transactions in $\mathtt{xDesc}(X)$, which in turn triggers xabort, sync, and sReturn instructions in $\mathtt{desc}(X)$ to finish up the subtree for X. Assuming that this abort process could happen atomically, it would be possible to eliminate the sigabort instruction, instead modeling a transaction abort as a series of xabort calls from the leaves of $\mathtt{desc}(X)$ back up to X itself. We choose to have separate sigabort and xabort instructions in the TCO model, however, because the sigabort instruction is useful for modeling more complex TM systems in which the abort of a transaction Y nested inside X is not instantaneous. For example, if we have TM with open-nested transactions, as we consider in Chapter 7, the abort of a transaction X may require the TM system to execute a "compensating transaction" to compensate for the effects of a transaction Y open-nested in X which has already committed.

5.6 Serializability of the TCO Model

In this section, we show that the TCO model presented in Section 5.5 generates computation traces $(\mathcal{C},\Phi)$ which are prefix-race-free, as defined by Definition 5.15. Thus, by Theorem 5.3, we know that the TCO model guarantees the serializability of transactions when the effects of aborted transactions can be safely ignored.

Properties of the TCO Model

This section first presents some structural invariants and definitions for the TCO model that will be useful later for showing that the TCO model generates prefix-race-free traces.

First, because the TCO model relies on Definition 5.17 for conflict detection, the next theorem shows that it preserves key invariants on readers and writers of a location ℓ.

Theorem 5.4. *On any step t, for all memory locations $\ell \in \mathcal{M}$, the TCO model maintains the following invariants on the sets $\mathtt{readers}(\ell)$ and $\mathtt{writers}(\ell)$:*

1. *For all $\ell \in \mathcal{M}$, $\left|\mathtt{leafSet}\left(\mathtt{writers}^{(t)}(\ell)\right)\right| = 1$, i.e., $\mathtt{leaf}(\mathtt{writers}^{(t)}(\ell))$ exists.*
2. *For any $X \in \mathtt{readers}^{(t)}(\ell)$, either*
   (a) $X \in \mathtt{ances}(\mathtt{leaf}(\mathtt{writers}^{(t)}(\ell)))$, or
   (b) $X \in \mathtt{desc}(\mathtt{leaf}(\mathtt{writers}^{(t)}(\ell)))$.

Proof. The proof is by induction on the instructions of the TCO model.

In the base case, for all locations $\ell \in \mathcal{M}$, we begin with $\mathtt{readers}^{(0)}(\ell) = \{\mathtt{root}(\mathcal{C})\}$, $\mathtt{writers}^{(0)}(\ell) = \{\mathtt{root}(\mathcal{C})\}$, and no other nodes in $\mathcal{C}$ except $\mathtt{root}(\mathcal{C})$. Thus, Invariants 1 and 2 are satisfied.

In the inductive step, suppose at the beginning of step t that Invariants 1 and 2 are satisfied. A `read` or `write` instruction on step t cannot break the invariants without causing a conflict according to Definition 5.17. Therefore, successful `read` and `write` operations preserve the invariant. An unsuccessful `read` or `write` operation can only trigger the `sigabort` of transactions, which does not affect either invariant.

An `xend` that commits a transaction X can only add the transaction $\mathtt{xParent}(X)$ to $\mathtt{readers}(\ell)$ or $\mathtt{writers}(\ell)$. Since $\mathtt{xParent}(X)$ is an ancestor of X, it cannot break either of the two invariants.

The remaining instructions preserve Invariants 1 and 2 trivially. A `spawn`, `sync`, or `sReturn` on step t preserves the invariants because they do not change the set active transactions or any transaction readsets or writesets. An `xbegin` preserves the invariants because it creates new transactions X with empty readsets and writesets. The `xabort` instruction preserves the invariants because it can only remove transactions from $\mathtt{readers}^{(t)}(\ell)$ or $\mathtt{writers}^{(t)}(\ell)$. □

The next definition generalizes the notion of the content set ($\mathtt{content}(X)$ from Section 5.2) to construct a step-dependent classification of all the nodes $v \in V(X)$.

Definition 5.18. *On any step t, for any $X \in \texttt{xactions}^{(t)}(C)$ and a memory operation $u \in \texttt{memOps}^{(t)}(C)$, define the sets $\texttt{cContent}^{(t)}(X)$, $\texttt{aContent}^{(t)}(X)$, and $\texttt{vContent}^{(t)}(X)$ according the $\texttt{ContentType}(t,u,X)$ procedure:*

```
ContentType($t,u,X$)        // For any $u \in \texttt{memOps}^{(t)}(t)$
$Z = \texttt{xParent}(u)$
while ($Z \neq X$)
    if $Z \in \texttt{vTree}^{(t)}(C)$, return $u \in \texttt{vContent}^{(t)}(X)$
    if $Z \in \texttt{ABORTED}^{(t)}(C)$, return $u \in \texttt{aContent}^{(t)}(X)$
    $Z \leftarrow \texttt{xParent}(Z)$
return $u \in \texttt{cContent}^{(t)}(X)$
```

Conceptually, Definition 5.18 partitions the node $u \in V(X)$ into one of three sets — the ***active content*** $\texttt{vContent}(X)$, the ***aborted content*** $\texttt{aContent}(X)$, or the ***closed content*** $\texttt{cContent}(X)$. When a transaction X completes on step t, the closed content is equal to the memory operations in the static content set defined earlier, i.e., $\texttt{cContent}^{(t)}(X) = \texttt{content}(X) \cap \texttt{memOps}^{(t)}(C)$.

Prefix-Race Freedom

Using the structural invariants in Theorem 5.4, as well as some other invariants on transaction readsets and writesets given in Appendix C, we can show that the TCO model generates traces (C,Φ) which are prefix-race-free.

First, the following theorem shows that once a memory operation u that reads from (writes to) a location ℓ is added to $\texttt{cContent}^{(t)}(Y)$ for some transaction Y, the location ℓ remains in the readset (writeset) of some active transaction Z which is a descendant of Y up until the time when Y ends.

Theorem 5.5. *Suppose that the TCO model generates a trace (C,Φ) with an execution order $\mathcal{S}$. For any transaction Y, let t_f be the step on which $\texttt{xend}$ or $\texttt{xabort}$ of Y occurs. Consider a memory operation $u \in \texttt{cContent}^{(t_f)}(Y)$ which accesses memory location ℓ on step t_u. On any step t such that $t_u \leq t < t_f$, there exists some $Z \in \texttt{xDesc}(Y) \cap \texttt{vTree}^{(t)}(C)$ (i.e., Z is an active transactional descendant of Y) satisfying the following conditions:*

1. *If $R(u,\ell)$, then $\ell \in \texttt{R}^{(t)}(Z)$.*
2. *If $W(u,\ell)$, then $\ell \in \texttt{W}^{(t)}(Z)$.*

Proof. Let $X_1, X_2, \ldots X_k$ be the chain of transactions from $\texttt{xParent}(u)$ up to but not including Y, i.e., $X_1 = \texttt{xParent}(u)$, $X_j = \texttt{xParent}(X_{j-1})$, and $\texttt{xParent}(X_k) = Y$. Since we assume that $u \in \texttt{cContent}^{(t_f)}(Y)$ and since Y completes at step t_f for every j in the range $1 \leq j < k$, there exists a unique step t_j (satisfying $t_u \leq t_j < t_f$) when an $\texttt{xend}$ changes $\texttt{status}(X_j)$ from $\texttt{PENDING}$ to $\texttt{COMMITTED}$. Otherwise, if any of the X_j aborted, we would have $u \in \texttt{aContent}^{(t_f)}(Y)$.

First, suppose that $R(u,\ell)$. On step t_u, when the memory operation u happens, (ℓ,u) is added to $\texttt{R}(X_1)$. In general, on step t_j, the $\texttt{xend}$ will propagate ℓ from $\texttt{R}(X_j)$ to $\texttt{R}(X_{j+1})$. Therefore, for any step t in the interval $[t_{j-1},t_j)$, we know $\ell \in \texttt{R}^{(t)}(X_j)$, i.e., for Theorem 5.5,

$Z = X_j$. Similarly, for any step t in the interval $[t_k, t_f)$, we have $\ell \in \mathtt{R}^{(t)}(Y)$, i.e., we choose $Z = Y$.

The case where $W(u,\ell)$ is completely analogous to the case of $R(u,\ell)$, except we have both $\ell \in \mathtt{R}^{(t)}(Z)$ and $\ell \in \mathtt{W}^{(t)}(Z)$. □

If the TCO model generates a trace $(\mathcal{C},\Phi)$ according to an execution order $\mathcal{S}$, the next theorem shows that the observer function Φ is a transactional-last-writer as defined in Definition 5.9.

Theorem 5.6. *Suppose that the TCO model generates a trace $(\mathcal{C},\Phi)$ and an execution order $\mathcal{S}$. Then, we have $\Phi = \mathcal{X}_{\mathcal{S}}$.*

Proof. This result can be proved by induction on the instructions of the TCO model using some invariants on the properties of readsets and writesets. See Appendix C, page 255 for details. □

Finally, we can put together Theorems 5.5 and 5.6 to prove that the TCO model generates traces which are prefix-race-free.

Theorem 5.7. *Suppose that the TCO model generates a trace $(\mathcal{C},\Phi)$ with an execution order $\mathcal{S}$. Then $\mathcal{S}$ is a prefix-race-free sort of $(\mathcal{C},\Phi)$.*

Proof. For the first condition of Definition 5.15, we know by Theorem 5.6 that the TCO model generates an order $\mathcal{S}$ for which $\Phi = \mathcal{X}_{\mathcal{S}}$.

To check the second condition, assume for contradiction that we have an order $\mathcal{S}$ generated by the TCO model, but there exists a prefix race between a transaction X and a memory operation $v \notin \mathtt{memOps}(X)$. Let w be the memory operation from Definition 5.14, i.e., $w \in \mathtt{cContent}(X)$, $w <_{\mathcal{S}} v <_{\mathcal{S}} \mathtt{sink}(X)$, $\neg(vHw)$, and w and v access the same location ℓ, with one of the accesses being a write. Let t_w and t_v be the steps when operations w and v occurred, respectively, and let t_f be the step when $\mathtt{sink}(X)$ occurs (either xend or xabort of X). We argue that on step t_v, the memory operation v should not have succeeded because it generated a conflict.

There are three cases for v and w. First suppose that $W(v,\ell)$ and $R(w,\ell)$. Since $t_w < t_v < t_f$, by Theorem 5.5, on step t_v, $\ell \in \mathtt{W}^{(t_v)}(Z)$ for some active transaction $Z \in \mathtt{desc}(X)$. Since $v \notin \mathtt{memOps}(X)$, we know $X \notin \mathtt{ances}(v)$. Thus, since Z is a descendant of X, we have $Z \notin \mathtt{ances}(v)$. Since $Z \notin \mathtt{ances}(v)$, by Definition 5.17, on step t_v, v generates a conflict with Z. The other two cases, where $R(v,\ell) \wedge W(w,\ell)$ or $W(v,\ell) \wedge W(w,\ell)$, are analogous. □

Chapter 6

Nested Parallelism in TM

This chapter investigates the challenge of designing a transactional memory (TM) system for a dynamic-threading platform which supports transactions with nested parallelism. Chapter 5 formally described semantics for such a TM system, but it did not discuss how these semantics might be implemented. Implementing transactions with nested parallelism efficiently can be tricky. Conflict detection in a dynamic-threading platform is complicated, and when transactions have large nesting depth, straightforward implementations can incur significant performance penalties for conflict detection and committing transactions. This chapter presents CWSTM,[1] the design of a software TM system that supports transactions with nested parallelism and nested transactions. CWSTM provides a theoretical performance bound on the overhead of conflict detection which is independent of the maximum nesting depth of transactions. CWSTM demonstrates that a dynamic-threading platform can support composable synchronization using transactions and still provide provable guarantees on performance.

Most work on TM focuses on supporting nested transactions that execute serially. This restriction arises naturally because most TM systems are designed for programs using static threads, and the overhead of creating or destroying a static thread naturally discourages programmers from having nested parallelism inside a transaction. Furthermore, the special case of serial transactions greatly simplifies conflict detection for TM. To detect a conflict, the runtime can easily determine whether two transactions X and Y are executing in parallel by simply comparing the ids of the threads executing X and Y. If the thread ids for X and Y match, then because transactions are serial, one must be nested inside the other. Otherwise, the thread ids are different, and X and Y are executing in parallel. A TM runtime also only needs to track relatively few thread ids, since programs with static threads typically try to avoid creating many more than P threads, where P is the number of available processors.

If one writes a program using a dynamic-threading language such as Cilk, however, then transactions with nested parallelism seem to be easy to express linguistically but difficult to implement efficiently. Because of Cilk's work-stealing scheduler, multiple worker threads may participate in executing a transaction X that has nested `spawn` statements. Thus, for conflict detection, comparing ids of worker threads is insufficient to determine whether two memory accesses by different transactions are executing in parallel or not. Instead,

[1] CWSTM represents joint work [3] with Kunal Agrawal and Jeremy T. Fineman.

one would like to label individual strands of serial execution in a program. Unfortunately, the number of parallel strands in a dthreaded program can in general be quite large. In a `parallel_for` loop, for example, the number of strands is proportional to the number of iterations in the loop, which is often related to the input size n of a problem, rather than P, the number of available processors. Thus, a natural question arises: how can we add transactions to a dynamic-threading language such as Cilk?

Contributions

In this chapter, I describe CWSTM, the first design of a TM system that supports transactions with nested parallelism and nested parallel transactions of unbounded nesting depth in a dynamic-threading platform. CWSTM demonstrates that one can provide a theoretical performance bound on the overhead of conflict detection in TM which is independent of the maximum nesting depth of transactions.

More specifically, CWSTM achieves this performance bound by using the XConflict data structure, a new data structure for Cilk-like dynamic-threading platforms which can answer concurrent conflict queries in $O(1)$ time and can be maintained efficiently. Using XConflict, for the restricted case when no transactions abort and there are no concurrent readers, CWSTM executes a transactional computation with work T_1 and span T_∞ on P processors in time $O(T_1/P + PT_\infty)$. This bound for TM supporting transactions with nested parallelism requires rather optimistic assumptions. To my knowledge, however, this result represents the first theoretical performance bound on such a TM system which is independent of the maximum nesting depth of transactions.

Outline

The remainder of this chapter is organized as follows. Section 6.1 explains the semantics of a generic TM system that supports transactions with nested parallel transactions using the transactional-computation framework from Chapter 5. Section 6.2 describes a naive conflict-detection algorithm that correctly implements these semantics, but has poor worst-case performance when the nesting depth of transactions is large. Section 6.3 describes the high-level design of CWSTM and its use of XConflict for conflict detection. Section 6.4 gives an overview of the XConflict algorithm. Section 6.5 provide details on data structures used by XConflict. Section 6.6 shows that XConflict, and hence CWSTM, is efficient for programs that experience no conflicts or contention. Finally, Section 6.7 concludes this chapter by discussing related work on nested parallelism in TM.

6.1 Semantics of Nested Parallel Transactions

This section reviews some of the key concepts from Chapter 5 that are useful for describing CWSTM. More specifically, this section uses the computation-tree framework and TCO model from Chapter 5 to describes the semantics of a generic TM system that supports transactions with nested parallelism and nested parallel transactions.

Conflict Detection

As described in Chapter 5, in the TCO model, every transaction X maintains a readset $\mathtt{R}(X)$ and a writeset $\mathtt{W}(X)$, which contains pairs (ℓ, v), where ℓ is a memory location (or object), and v is a memory operation that either reads from or writes to ℓ.[2] On any step, t, the sets $\mathtt{readers}^{(t)}(\ell)$ and $\mathtt{writers}^{(t)}(\ell)$ represent the set of active transactions that currently have ℓ in their readset or writeset, respectively.

This chapter considers TM systems that performs ***eager conflict detection***, where the TM system must test for conflict before performing each `read` or `write` instruction. An access is unsuccessful if it generates a transactional conflict. TM systems with serial, closed-nested transactions report conflicts when two active transactions on different threads are accessing the same object ℓ, and one of those accesses is a write. Thus, only a single thread at a time can have ℓ in its writeset. When transactions can have nested parallelism, one can generalize this definition of conflict in a straightforward manner, as given in Definition 5.17 from Section 5.5.

Equivalently, we can think of the TM system as maintaining the invariants described in Theorem 5.4. Intuitively, Theorem 5.4 states that for any memory location ℓ, at any time t, the set $\mathtt{writers}(\ell)$ must fall along a single ***spine*** — a chain through the computation tree C that has a single transaction $\mathtt{leaf}(\mathtt{writers}(\ell))$ which is a descendant of all transactions in $\mathtt{writers}(\ell)$. Theorem 5.4 also states that for all transactions $Y \in \mathtt{readers}(\ell)$, Y must either be an ancestor or descendant of $\mathtt{leaf}(\mathtt{writers}(\ell))$.

Theorem 5.4 suggests that one can check for conflicts for a memory operation u that accesses a location ℓ by looking at only one writer and only a small number of readers. Since all the transactions writing to ℓ fall along a single spine by Theorem 5.4, Invariant 1, the transaction $\mathtt{leaf}(\mathtt{writers}(\ell))$ belongs to $\mathtt{writers}(\ell)$ and is a descendant of all transactions in $\mathtt{writers}(\ell)$. Define the set $\mathtt{lastReaders}(\ell)$ as

$$\mathtt{lastReaders}(\ell) = \mathtt{readers}(\ell) \cap \mathtt{desc}(\mathtt{leaf}(\mathtt{writers}(\ell))) .$$

By Invariant 2 of Theorem 5.4, for all transactions $Y \in \mathtt{readers}(\ell)$, we either have $Y \in \mathtt{ances}(\mathtt{leaf}(\mathtt{writers}(\ell)))$ or $Y \in \mathtt{lastReaders}(\ell)$. Thus, we can check for conflicts with a memory operation u as follows:

1. If u tries to read from ℓ, then there is no conflict if and only if $\mathtt{leaf}(\mathtt{writers}(\ell))$ is an ancestor of u.
2. If u tries to write to ℓ, there is no conflict if and only if for all $Z \in \mathtt{lastReaders}(\ell)$, Z is an ancestor of u.

Transaction Commit and Abort

Recall that in the TCO model, each active transaction X maintains a status field $\mathtt{status}(X)$ to indicate whether it can eventually commit or abort. Normally, a transaction X begins with an empty readset and writeset, and $\mathtt{status}(X) = \mathtt{PENDING}$. If a transaction X completes

[2] In this chapter, I use the terms "memory location" and "object" interchangeably. For a TM system that works with objects, for simplicity, I assume that objects do not overlap, and that accessing any part of an object is equivalent to accessing the entire object.

without any conflicts, then the TM system can commit X by changing `status`(X) from `PENDING` to `COMMITTED`, and merging X's readset and writeset into the corresponding sets of `xParent`(X).

If a memory operation v would cause a conflict between $X =$ `xParent`(v) and another transaction Y, then v triggers an abort of either X or Y (or both). Say X is aborted. An abort of a transaction X changes `status`(X) from `PENDING` to `PENDING_ABORT`, and also changes the status of any `PENDING` (nested) transaction Y in the subtree of X to `PENDING_ABORT`. In general, a `PENDING_ABORT` transaction X that is also ready can only complete by changing its status to `ABORTED`. Conceptually, whenever a transaction X is `ABORTED`, the TM system discards the readset and writeset of X. Since X is no longer active after this action occurs, this abort also conceptually removes X from `readers`(ℓ) and `writers`(ℓ) for all objects ℓ. If v causes a conflict and the runtime chooses to abort $Y \neq$ `xParent`(v), then the conflict is not fully resolved until `status`(Y) has changed to `ABORTED`.

Code Example

To understand the semantics of TM with nested parallel transactions concretely, we consider an example program shown in Figure 6-1 and investigate how a TM system implementing conflict detection according to Definition 5.17 constrains the possible program outputs. First, this section describes the **parallel** construct used by Figure 6-1. Then it illustrates the semantics of conflict detection for this code using the TCO model.

Instead of using Cilk syntax, with the `spawn` and `sync` keywords, Figure 6-1 uses a **parallel** construct similar to Dijkstra's "cobegin," which allows the two following code blocks (each contained in $\{.\}$) to run in parallel. In other words, **parallel**$\{A_1\}\{A_2\}$ is conceptually equivalent to `spawn` A_1, a call to A_2, and then a `sync`. Figure 6-1 presents pseudocode for the same transactional computation as shown in Figure 5-5 and the DAG in Figure 5-6. The **parallel** construct is slightly less general than Cilk code, since it allows only two strands of execution to join at a `sync` instead of having multiple parallel functions in the same sync block. It has the advantages of using more compact notation and generating slightly simpler computation trees. Thus, in the remainder of this chapter, we only consider parallel computations generated using the **parallel** construct for CWSTM. Figure 6-2 shows a computation tree C which represents a complete execution of the computation in Figure 6-1.

The scoping of atomicity in constrains the possible outputs of the computation in Figure 6-1 as compared to the original code in Figure 5-3. The increment in line 4 and the code block in lines 6–9 must appear as though one executes entirely before the other. If the **atomic** statements in lines 4 and 5 were removed, then these two blocks could interleave arbitrarily, even though the entire procedure is protected by an **atomic** statement in line 1. Basically, the atomicity applies only when comparing two blocks of code belonging to different transactions (protected by different atomic statements), not parallel blocks within the same transaction (protected by the same atomic statement).

Conflict as stated in Definition 5.17 naturally enforces what is sometimes referred to in TM literature as strong atomicity [32]. Intuitively, strong atomicity states that a TM system still detects a conflict when a memory operation v accessing a location ℓ outside any transaction runs in parallel with a transaction X that has ℓ in its readset or writeset. Strong atomicity implies that although line 8 itself is not atomic, it cannot perform its

```
XParallelIncrement()
atomic {                                                  // Transaction $X_1$
    $x \leftarrow 0$
    parallel
        { atomic {$x \leftarrow x+1$}              }      // $X_2$
        { atomic {                                        // $X_3$
              $x \leftarrow x+10$
              parallel
                { $x \leftarrow x+100$          }
                { atomic {$x \leftarrow x+1000$} }        // $X_4$
          }                                        }
}
print $x$
```

Figure 6-1: The computation from Figure 5-5, expressed using the **parallel** construct instead of Cilk keywords (spawn and sync). Since atomic blocks are not placed around *all* increments, this program still permits multiple outputs—valid outputs are 111 and 1111. The (symmetric) 1011 is excluded due to strong atomicity.

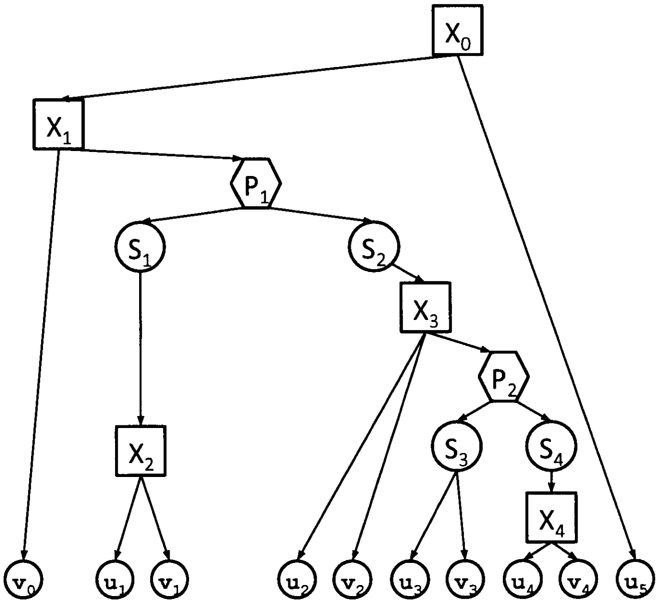


Figure 6-2: A computation tree representing a complete execution of the code from Figure 6-1. Each update of x is decomposed into a read u_i and a write v_i.

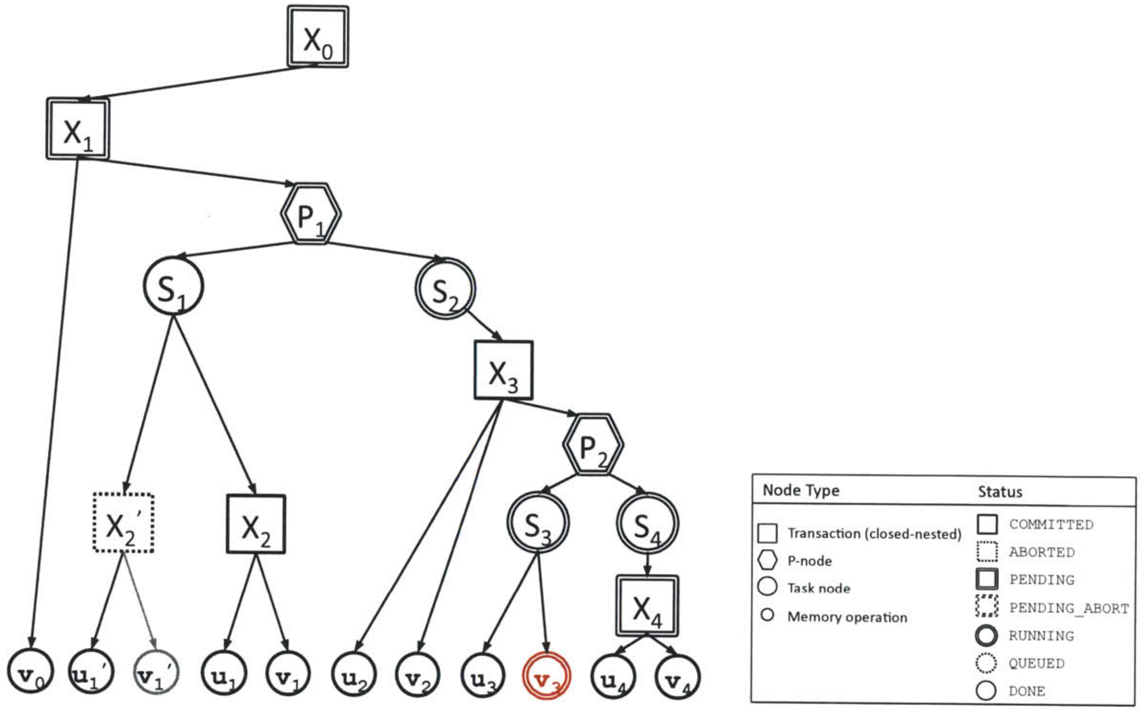


Figure 6-3: A computation tree representing an ongoing execution of the computation from Figure 6-1. The transaction X_4 is active, with $(x, v_4) \in \mathtt{W}(X_4)$ and $(x, v_4) \in \mathtt{R}(X_4)$. Thus, an attempt to execute memory operation v_3 causes a conflict with X_4.

`write` between line 9's `read` and `write`.

To understand strong atomicity more precisely, consider the computation tree shown in Figure 6-3. After u_4 performs a `read` of x, it adds x to the readset of X_4. After u_4 occurs but before v_4 occurs or X_4 commits, if v_3 tries to write to x, it causes a conflict with X_4. We can, however, have line 8 read x (u_3), line 9 read and write x and commit (transaction X_4), and then line 8 write x (v_3). This interleaving can occur because when u_3 happens, it adds x to the readset of X_1, and u_4 and v_4 can subsequently happen because they are both descendants of X_1 in the computation tree. This behavior means that for the execution of X_3, the increment of 1000 can be "lost" (by being overwritten) but the increment of 100 cannot. Another way of describing strong atomicity is that each memory operation is viewed as its own transaction that does not abort.

For the execution shown in Figure 6-3, the first transaction instance which attempts to execute line 4, X_2' is `ABORTED`. One way this abort X_2' could occur is if x was already in the readset or writeset of X_3 (e.g., because of u_2) when v_1' tries to write to x. In general, the computation tree framework and the TCO model captures the instances of transactions that abort as well as those that commit.

In summary, the only possible outputs for the final value of x in Figure 6-1 are 111 and 1111. The only update in Figure 6-1 that can get lost is the one performed by X_4, because it is the only update that occurs in parallel with an update to x which is not synchronized with respect to X_4.

6.2 A Simple TM with Nested Parallel Transactions

The CWSTM semantics described in Section 6.1 suggest a design for a TM system that supports transactions with nested parallelism. In particular, Theorem 5.4 suggests that for conflict detection, a TM system can maintain an active writing transaction `leaf(writers(`ℓ`))` and some active reading transactions `lastReaders(`ℓ`)` for each object (or memory location) ℓ. This section focuses on a straightforward data structure, called an "access stack," used to maintain these values. One can show, however, that an access stack yields a TM with poor worst-case performance, even assuming the rest of the TM system incurs no overhead. Later, Section 6.3 describes the CWSTM design, which uses a lazy variant of the access stack and has better worst-case performance.

The ***access stack*** for an object ℓ is a stack where every element is either a transaction that has written to ℓ or a list of transactions that have read from ℓ. The order of transactions on the stack is consistent with the ancestry of transactions in the computation tree. The writing transaction `leaf(writers(`ℓ`))` is either on top (first item to pop) of the stack or is the next element on the stack. If the writer is not on the top of the stack, then `lastReaders(`ℓ`)` is. No two consecutive elements are sets of readers.

To maintain access stacks, we can perform the following operations on an access to an object ℓ, locking the stack for ℓ to guarantee atomicity. Consider (a memory operation whose transactional parent is) a transaction X that successfully reads ℓ. If the top of the stack contains a set of readers, then X is added to that set, assuming it is not already there. If the top of the stack is a writer other than X, then $\{X\}$ is added to the top of the stack. Similarly, if X successfully writes ℓ, then X is pushed onto the top of the stack if it not already there.

Whenever a transaction X commits, for each ℓ in X's readset, X is removed from the top of ℓ's access stack and replaced with `xParent`(X) (in a fashion that ensures there are no duplicated transactions). This action mimics the commit semantics from Section 6.1: when a transaction X commits, the objects in its readset and writeset are moved to `xParent`(X)'s readset and writeset, respectively. If instead X aborts, then X is popped from each relevant object's access stack. To facilitate rollback on aborts, every access-stack entry corresponding to a write stores the old value before the write.[3]

Maintaining the access stack has poor worst-case performance because the work required on the commit of transaction X is proportional to the size X's readset. If the original program (without transactions) had work T_1, then this implementation might require work $\Omega(dT_1)$, where d is the maximum nesting depth of transactions. In particular, consider the function `f` shown in Figure 6-4. A call of `f(d)` generates a serial chain of nested transactions, each incrementing a different place in the array `x`. When the transaction at nesting depth j commits, it updates $d-j$ access stacks for a total of $\Theta(d^2)$ access-stack updates. The work of the original program (without transactions), however, is only $\Theta(d)$.

In general, this asymptotic blowup can occur if a TM system with nested transactions must perform work proportional to the size of a transaction's readset or writeset on every commit. For example, a TM system that validates every transaction due to lazy conflict detection for reads exhibits this problem. Similarly, a TM system that copies data on commit

[3] This value can either be stored in the stack itself or in a log per transaction.

```
void f(int i) {
  if (i >= 1) {   atomic { x[i]++; f(i-1); }    }
}
```

Figure 6-4: A computation which generates deeply nested transactions. The call f(d) generates nested transactions d levels deep. If the transactions in f are elided, this call performs $\Theta(d)$ work. Using a TM system that needs to update multiple access stacks on every transaction commit, however, $\Omega(d^2)$ work might be required to commit transactions, even if no transaction conflicts or aborts are possible.

due to lazy object updates also has this issue.

6.3 Overview of CWSTM Design

This section describes the CWSTM design for a transactional-memory system with nested parallel transactions. It first describes how CWSTM updates the status of computation-tree nodes on commits and aborts. It then gives an overview of the conflict-detection mechanism, which includes a "lazy access stack," improving on the shortcoming of the access stack from Section 6.2. Finally, this section describes properties of the Cilk-like work-stealing scheduler used by CWSTM. The XConflict data structure requires such a scheduler for its performance and correctness. The details of the XConflict data structure are deferred until later, in Sections 6.4 and 6.5.

CWSTM explicitly builds the internal nodes of the computation tree (i.e., leaf nodes for memory operations are omitted). Each node maintains a field which explicitly represents the node's status (PENDING_ABORT, PENDING, COMMITTED, or ABORTED) and updates these fields as described in the TCO model (from Chapter 5, Section 5.5). For example, when a transaction X commits, CWSTM atomically changes status(X) from PENDING to COMMITTED.

Since a transaction may signal an abort of a transaction running on a (possibly different) processor whose descendants have not yet completed, aborting transactions is more involved. When an active transaction X aborts itself (possibly because of a conflict) it simply atomically updates status(X) to ABORTED. This type of update corresponds to an xabort instruction from the TCO model.

Alternatively, if a worker p wishes to abort X even though p is not currently executing X, then p performs an sigabort instruction. First, p atomically changes status(X) from PENDING to PENDING_ABORT. Then p walks X's active subtree, changing status(Y) to PENDING_ABORT atomically for each active $Y \in$ desc(X). Notice that p does not change any status to ABORTED—only the worker p' which finishes executing the transaction Y is allowed to perform this update. More precisely, a worker p' only "discovers" that the status of a transaction Y is PENDING_ABORT when Y becomes ready (i.e., and thus has no active descendants). When it does, then p' performs an xabort for Y.

During an abort of a transaction X, CWSTM may also change the status field of some of X's COMMITTED descendants Y to ABORTED. These changes enable CWSTM to more quickly determine that Y has an ABORTED ancestor X, and thus a memory operation should not conflict with Y. Section 6.5 describes these updates in greater detail.

In CWSTM, the rollback of objects on abort occurs lazily, and thus is decoupled from an `xabort` operation. Once the status of a transaction X changes to ABORTED, other transactions that try to access an object modified by X help with cleanup for that object.

Conflict Detection and the Lazy Access Stack

We now discuss conflict detection. Recall that to commit a transaction X, the TM implementation described in Section 6.2 performs work proportional to the size of the readset of X, since it explicitly maintains the list of active transactions accessing a given location ℓ. CWSTM is able to avoid this overhead by utilizing Definition 6.1, an alternate characterization of transaction conflicts.

Definition 6.1. *Consider a memory operation v that accesses a memory location ℓ on step t_v, and a (possibly inactive) transaction $X \in \mathtt{xactions}^{(t_v)}(\mathcal{C})$. Then we say that v* ***conflicts with** X if three conditions are satisfied:*

1. *There exists a $u \in \mathtt{cContent}^{(t_v)}(X)$ such that*

$$\{(R(u,\ell) \vee W(u,\ell)) \wedge W(v,\ell)\} \vee \{W(u,\ell) \wedge (R(v,\ell) \vee W(v,\ell))\} \ ,$$

2. *For all $D \in \mathtt{xAnces}(X)$, $\mathtt{status}(D) \neq$ ABORTED, and*
3. *$\mathtt{leaf}(\mathtt{xAnces}(X) \cap \mathtt{vTree}^{(t_v)}(\mathcal{C})) \notin \mathtt{ances}(v)$.*

Intuitively, Definition 6.1 reports a conflict whenever Definition 5.17 reports a conflict. The key property of Definition 6.1 is that it enables conflict detection for a memory operation v by checking against transactions X that have already completed, whereas the original definition can only check for conflicts against active transactions Y.

Lemma 6.1. *Consider a memory operation v that occurs on a step t_v. For any transaction $X \in \mathtt{xactions}^{(t_v)}(\mathcal{C})$, let $Y = \mathtt{leaf}(\mathtt{xAnces}(X) \cap \mathtt{vTree}^{(t_v)}(\mathcal{C}))$. Then we have the following:*

1. *If v conflicts with X according to Definition 6.1, then v conflicts with Y according to Definition 5.17, and*

2. *If v does not conflict with X according to Definition 6.1, and $X \in \mathtt{xactions}^{(t_v)}(\mathcal{C}) \cap \mathtt{vTree}^{(t_v)}(\mathcal{C})$, then v does not conflict with Y according to Definition 5.17.*

Proof. This characterization of conflict can be derived from Definition 5.17 and using the structural invariants of the TCO model described in Section 5.6. Consider each statement in Lemma 6.1.

1. For the first statement, consider two cases for X, depending on whether X is active.

(a) Suppose that $X \in \texttt{xactions}^{(t_v)}(\mathcal{C}) \cap \texttt{vTree}^{(t_v)}(\mathcal{C})$, i.e., X is an active transaction. Since X is active, in Lemma 6.1, we have $Y = X$. Suppose that v conflicts with X according to Definition 6.1. Then there exists a $u \in \texttt{cContent}^{(t_v)}(X)$ that is causing a conflict according to Definition 6.1.

First, assume that $(R(u,\ell) \vee W(u,\ell)) \wedge W(v,\ell)$. By Theorem 5.5 from Section 5.6, there exists some transaction $Z \in \texttt{xDesc}(X) \cap \texttt{vTree}^{(t_v)}(\mathcal{C})$ with $\ell \in \texttt{R}^{(t_v)}(Z)$, or equivalently, with $Z \in \texttt{readers}^{(t_v)}(\ell)$. Also, we know $Y = X \in \texttt{ances}(Z)$. Thus, we have $Z \notin \texttt{ances}(v)$, since Z is a descendant of Y, and by Condition 3 of Definition 6.1, we have $Y \notin \texttt{ances}(v)$. Since $Z \notin \texttt{ances}(v)$, $Z \in \texttt{readers}^{(t_v)}(\ell)$, and $W(v,\ell)$, it follows that v generates a conflict with Z according to Definition 5.17.

The other case, when $W(u,\ell) \wedge (R(v,\ell) \vee W(v,\ell))$, is analogous. By the same logic, there exists a transaction Z such that $Z \notin \texttt{ances}(v)$, $Z \in \texttt{writers}^{(t_v)}(\ell)$, and $R(v,\ell)$. Thus, v generates a conflict with Z according to Definition 5.17.

(b) Suppose that $X \notin \texttt{xactions}^{(t_v)}(\mathcal{C}) \cap \texttt{vTree}^{(t_v)}(\mathcal{C})$, i.e., X is no longer active, and v conflicts with X according to Definition 6.1. We can reduce this case to the previous one.

More precisely, by the first two conditions of Definition 6.1, there exists a $u \in \texttt{cContent}^{(t_v)}(X)$, and X has no ABORTED ancestors D. Since X is inactive and all transactions D on the path from X up to but not including Y must be COMMITTED, we must have $u \in \texttt{cContent}^{(t_v)}(D)$. Thus, $u \in \texttt{cContent}^{(t_v)}(Y)$. In this case, we have $\texttt{leaf}(\texttt{xAnces}(X) \cap \texttt{vTree}^{(t_v)}(\mathcal{C})) = \texttt{leaf}(\texttt{xAnces}(Y)) \cap \texttt{vTree}^{(t_v)}(\mathcal{C}) \notin \texttt{ances}(v)$. Thus, a conflict of v with X also implies a conflict of v with Y according to Definition 6.1. By the previous case, since Y is an active transaction, a conflict for Y according to Definition 6.1 implies a conflict according to Definition 5.17.

2. For the second statement of Lemma 6.1, consider the contrapositive. We can show that if v conflicts with Y according to Definition 5.17, then, v conflicts with $Y = X$ according to Definition 6.1.

   If v creates a conflict with $X = Y$ according to Definition 5.17, then $Y \notin \texttt{ances}(v)$ and either $R(v,\ell) \wedge (Y \in \texttt{writers}^{(t_v)}(\ell))$ or $W(v,\ell) \wedge (Y \in \texttt{readers}^{(t_v)}(\ell))$. The second and third conditions of Definition 6.1 are satisfied since $X = Y$. To check the first condition, consider the two options of whether Y is a writer or a reader of ℓ. If $Y \in \texttt{writers}^{(t_v)}(\ell)$, then there exists a memory operation u such that $(\ell,u) \in \texttt{W}^{(t_v)}(Y)$. One can then show (by Theorem C.2 in Appendix C) that $u \in \texttt{cContent}^{(t_v)}(\ell)$ and $W(u,\ell)$, thereby satisfying the first condition of Definition 6.1. Similarly, if $Y \in \texttt{readers}^{(t_v)}(\ell)$, one can show that there exists a $u \in \texttt{cContent}^{(t_v)}(\ell)$ with $R(u,\ell)$.

□

Intuitively, Lemma 6.1 is true because for any inactive transaction X that has no aborted ancestors, if X wrote to a memory location ℓ, then Y, the nearest active transactional ancestor of X (as in Definition 6.1), logically belongs to $\texttt{writers}(\ell)$, since the commit of all the nested transactions on the path from X to Y propagates ℓ into the writeset of Y.

```
XCONFLICT-ORACLE($X,u$)
    // For any node $X$ and active memory operation $u$
1   if $\exists D \in$ ances($X$) such that status($D$) = ABORTED
2       return "no conflict: $X$ aborted"

3   $Y \leftarrow$ closest active transactional ancestor of $X$
4   if $Y \in$ ances($u$)
5       return "no conflict: $X$ committed to $u$'s ancestor"
6   else pick a transaction $B$ in (ances($Y$) − ances(LCA($Y,u$)))
7       return "conflict with $B$"
```

Figure 6-5: Pseudocode for a conflict-detection query suggested by Lemma 6.1. The details of subroutines (e.g., line 3) are omitted. In fact, many of these subroutines do not have $O(1)$-time implementations.

Thus, Lemma 6.1 suggests a conflict-detection algorithm that does not require maintaining leaf(writers(ℓ)) and updating an access stack eagerly, i.e., on every memory access. Let X be the last transaction that has successfully written to ℓ. When u accesses ℓ, test for conflict by finding X's nearest active transactional ancestor Y and determining whether Y is an ancestor of u. Figure 6-5 gives pseudocode for this test. CWSTM does not actually implement this query as given—instead, it uses an equivalent, but more efficient query, which is described in Section 6.4.

CWSTM facilitates the necessary conflict queries by using a ***lazy access stack*** to maintain the most recent successful write (and reads). The structure of the lazy access stack is somewhat different from the simple access stack given in Section 6.2. An object ℓ's lazy access stack stores (possibly inactive) transactions that have written to ℓ and sets of transactions that have read from ℓ with the stack entries ordered chronologically by access. The top of the stack holds either the last writer (leaf(writers(ℓ))) or the set of last readers (lastReaders(ℓ)). CWSTM maintains the invariant that if a transaction X on the stack has aborted, then all transactions located above X on the stack (later chronologically) also have aborted ancestors, and thus they represent values that should be rolled back. Unlike a normal access stack, the lazy access stack is not updated on a transaction commit, i.e., the stack need not be updated to conceptually merge a transaction's readset or writeset into its parent's readset or writeset. On memory operations, new transactions are added to the access stack in the same way as described in Section 6.2.

Figure 6-6 gives pseudocode for an instrumentation of each memory access, assuming for simplicity that all memory accesses behave as `write` instructions.[4] Incorporating readers into the access stacks is more complicated, but conceptually similar. If a memory access u does not belong to an aborting transaction (checked in line 2), then it is allowed to proceed. Lines 4–5 test for conflict with the last writer to ℓ.

If the last writer has aborted or has an aborted ancestor, then lines 6–9 fix the access

[4] It is possible to reduce locking on the access stack, but Figure 6-6 does not describe these optimizations.

```
ACCESS $(u, \ell)$
Z ← xParent$(u)$
if status$(Z)$ = PENDING_ABORT return XABORT

// Otherwise $Z$ is active
accessStack$(\ell)$.LOCK()

// Set $X$ to be the last writer of $\ell$.
$X$ ← accessStack$(\ell)$.TOP()
result ← XCONFLICT-ORACLE $(X, u)$

if result is "no conflict: $X$ aborted"
    accessStack$(\ell)$.UNLOCK()
    CLEANUP$(\ell)$                          // Rollback some values
    return RETRY                           // The access should be retried

if result indicates a conflict with transaction $B$
    if choose to abort self
        accessStack$(\ell)$.UNLOCK()
        return XABORT
    else accessStack$(\ell)$.UNLOCK()
        signal an abort of $B$
        return RETRY

// Otherwise, there is no conflict: $X$ is an ancestor of $Z$
if $Z \neq X$                                  // $Z$'s first access to $\ell$
    // Log the access
    LOGVALUE$(Z, \ell)$
    accessStack$(\ell)$.PUSH(Z)

// Actually perform the write operation
Perform the write
accessStack$(\ell)$.UNLOCK()
return SUCCESS
```

Figure 6-6: Pseudocode for a memory operation u which accesses an object ℓ, assuming that all accesses are writes. ACCESS (u, ℓ) returns XABORT if Z should abort, RETRY if the access should be retried, or SUCCESS if the memory operation succeeded.

```
CLEANUP($\ell$)
accessStack($\ell$).LOCK()
$X \leftarrow$ accessStack($\ell$).TOP()
if $\exists D \in$ ances($X$) such that status($D$) = ABORTED
    RESTOREVALUE($X,\ell$) // Restore $\ell$ from $X$'s log
    accessStack($\ell$).POP()
accessStack($\ell$).UNLOCK()
```

Figure 6-7: Code for removing an aborted transaction from the top of an access stack for location ℓ, assuming all accesses are writes. To completely update an access stack, this method should be called repeatedly until the top transaction has no ABORTED ancestor.

stack by calling CLEANUP. This auxiliary procedure, given in Figure 6-7, rolls back the value of the topmost aborted transaction on ℓ's access stack. Since CLEANUP cannot introduce new conflicts, the access is retried in line 9.

If, on the other hand, there is a conflict between u and an active transaction (lines 10–16), then either xParent(u) must abort or a conflicting transaction (B) must abort.

Finally, if there are no conflicts, then the access is successful. Lines 18–20 update the access stack as necessary, before the actual access occurs.

While u is running the ACCESS method, concurrent transactions (that access ℓ) can continue to commit or abort. The commit or abort of such a transaction can eliminate a conflict with u, but it never creates a new conflict with u. Thus, concurrent changes may introduce spurious aborts, but they do not affect correctness.

Scheduling for CWSTM

CWSTM relies on the properties of a "Cilk-like" work-stealing scheduler for efficiency and correctness. Conceptually, the CWSTM runtime dynamically generates a computation tree $\mathcal{C}$ as it executes.[5] As in Cilk, CWSTM executes a computation $\mathcal{C}$ on P worker threads, each maintaining a deque to store tasks that can be stolen. Conceptually, on each step, each worker thread maintains a ***producer node*** X, a node in the computation tree $\mathcal{C}$ that can execute instructions. A computation begins with root($\mathcal{C}$), the root of the computation tree, as the producer node for a single worker, with the deques of all other workers empty. When a worker p_1 executes a spawn instruction, it creates a P-node with two S-nodes as children, pushes the right S-node onto the bottom of its deque, and begins working on the left S-node. When p_1 runs out of S-nodes on its deque, it can steal an S-node Y from the top of the deque of a ***victim*** worker, set Y as its producer node, and begin executing instructions and the subtree of $\mathcal{C}$ rooted at Y.

[5] The execution model for CWSTM represents a combination of the TCO model from Section 5.5 and the *CCT* model from Section A.4. CWSTM transactions conceptually behave as they do in the TCO model, but instead of assuming a generic scheduler that nondeterministically chooses any ready node to execute the next instruction, CWSTM assumes the specific scheduler described by the *CCT* model.

More precisely, we say that a scheduler is ***Cilk-like*** if it satisfies two properties. First, each worker thread executes subtrees of the computation tree in a left-to-right fashion, i.e., it starts executing the left child of a P-node before the right. Second, workers use deques for load balancing. In particular, each worker stores tasks on a deque, pushes and pops tasks from the bottom of its deque in the common case, and tries to steal tasks from the top of a victim's deque only when its own deque is empty. In terms of a computation tree, this second condition implies that whenever a worker p_1 steals from a victim worker p_2, then p_1 steals the right subtree from the highest P-node in p_2's subtree that has work available.

6.4 Conflict Detection in CWSTM

This section describes the high-level ***XConflict*** algorithm for conflict detection in CWSTM. As the computation tree dynamically unfolds during an execution, XConflict dynamically partitions the computation tree into "traces," where each trace consists of memory operations (and internal nodes) that can only be executed by one worker at a time. The XConflict algorithm uses several data structures that organize either traces, or nodes and transactions contained in a single trace. This section describes traces and gives a high-level algorithm for conflict detection. The description of the data structures used by XConflict is deferred until Section 6.5.

By dividing the computation tree into traces, the XConflict algorithm reduces the cost of locking on shared data structures. In XConflict, updates and queries on a data structure whose elements belong to a single trace can be performed without locks because these updates are performed by a single worker. Also, data structures whose elements are traces also support queries in constant time without locks. These data structures are, however, shared among all processors, and therefore require a global lock on updates. Since the XConflict algorithm creates traces only on steals, however, the number of traces is with high probability at most $O(PT_\infty)$ — the number of steals performed by a Cilk-like randomized work-stealing scheduler when executing a computation using P worker threads/processors. Therefore, one can similarly bound the number of updates for XConflict's data structures.

The technique of splitting the computation into traces and having two types of data structures — "global" data structures whose elements are traces and "local" data structures whose elements belong to a single trace— appears in the SP-hybrid algorithm for series-parallel maintenance, which was described by Bender et al. [25], and was later in improved by Fineman [47]. Compared to SP-hybrid, the traces for CWSTM differ slightly, and the data structures are a little more complicated, but the analysis technique is similar.

Trace Definition and Properties

XConflict assigns computation-tree nodes to ***traces*** in the essentially the same fashion as the SP-hybrid data structure described in [25, 47]. This section briefly describes the structure of traces here. Since the computation tree for CWSTM has a slightly different canonical form from the canonical Cilk parse tree use for SP-hybrid, XConflict simplifies the trace structure slightly by merging some traces together.

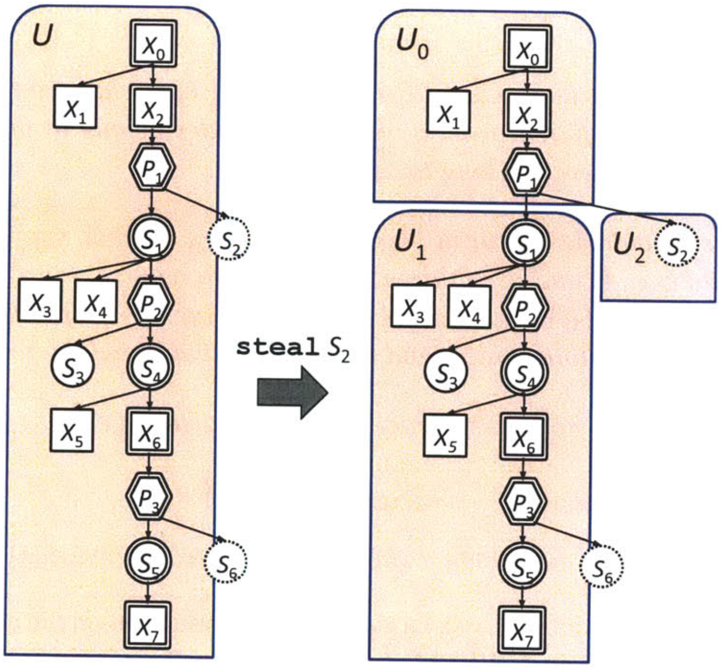


Figure 6-8: Traces of a computation tree (a) before and (b) after a steal instruction. Before the steal, only one worker is executing the subtree, but S_2 and S_6 are ready. After the steal, the subtree rooted at the highest ready S-node (S_2) is executed by the thief. The subtree rooted at S_1, on the other hand, is still being executed by the victim worker.

Formally, each trace U is a disjoint subset of nodes of the computation tree. Let $\mathcal{J}$ denote the set of all traces. $\mathcal{J}$ partitions the nodes of the computation tree $\mathcal{C}$. For any trace $U \in \mathcal{J}$ and any computation-tree node $B \in \texttt{nodes}(\mathcal{C})$, if B belongs to trace U, we say that $B \in U$, or $\mathsf{trace}(B) = U$.

In CWSTM, only one worker at a time can be executing nodes from a given trace U. Initially, the entire computation belongs to a single trace, and normally a worker executes nodes from a trace in a depth-first manner. As the program executes, however, traces dynamically split into multiple traces when steals occur.

More specifically, when a steal occurs and a processor steals the right subtree of a P-node $\mathsf{P} \in U$, the trace U splits into three traces U_0, U_1, and U_2 (i.e., $\mathcal{J} = \mathcal{J} \cup \{U_0, U_1, U_2\} - \{U\}$). Each of the left and right subtrees of P become traces U_1 and U_2, respectively. The trace U_0 consists of those nodes remaining after P's subtrees are removed from U. Although the worker performing the steal begins work on *only the right subtree* of P, both subtrees become new traces. Figure 6-8 gives an example of traces resulting from a steal. The left and right children of the *highest uncompleted* P-node P_1 are task nodes which are the roots of two new traces, U_1 and U_2.

Traces in CWSTM satisfy the following properties.

Property 1. *Every trace $U \in \mathcal{J}$ has a well-defined* **head** *task S-node $\mathsf{S} = \texttt{head}[U] \in U$ such that for all nodes $B \in U$, we have $\mathsf{S} \in \texttt{ances}(B)$.*

Property 2. *The computation-tree nodes of a trace $U \in \mathcal{J}$ form a tree rooted at* $\mathsf{S} = \mathit{head}[U]$.

Property 3. *Trace boundaries occur at P-nodes. Either both children of the P-node and the node itself belong to different traces, or all three nodes belong to the same trace. All children of an S-node, however, belong to the same trace.*

Property 4. *Trace boundaries occur at "highest" P-nodes. That is, suppose that a P-node* P *has a stolen child (i.e.,* P *and its children belong to different traces). Consider any node* $\mathsf{P}' \in \mathit{ances}(\mathsf{P}) \cap \mathit{pNodes}(\mathcal{C})$ *(i.e.,* P' *is an ancestor P-node of* P*). If* P *is in the left subtree of* P'*, then* P' *must have a stolen child (and therefore,* P *and* P' *belong to different traces).*

Property 5. *On any step t, consider a trace U which is* ***active****, i.e.,* $U \cap \mathit{vTree}^{(t)}(\mathcal{C}) \neq \emptyset$*. Then, we have*

$$\mathit{head}[U] = \mathit{root}(U \cap \mathit{vTree}^{(t)}(\mathcal{C})) \ ,$$

that is, all active nodes in U fall along a single chain rooted at $\mathit{head}[U]$.

These properties can be proved by a straightforward induction on the actions of the TCO model and on the rules for how CWSTM generates traces. Property 4 follows from the use of Cilk-like work-stealing, in which workers steal from the top of deques. Property 5 holds because $\mathtt{vTree}(\mathcal{C})$, the tree of active nodes in the computation tree, only branches at the children of P-nodes where steals occur, and these steals cause traces to split. Properties 1 and 5 imply that a trace U is active if and only if $\mathtt{head}[U]$ is active in $\mathcal{C}$.

It is useful to define some notation on traces. For a trace $U \in \mathcal{J}$, we use $\mathtt{xparent}[U]$ as a shorthand for $\mathtt{xParent}(\mathtt{head}[U])$. It is also useful to define the ***task parent*** of nodes and traces. For any node $B \in \mathtt{nodes}(\mathcal{C})$, define the task parent $\mathtt{tParent}(B)$ as

$$\mathtt{tParent}(B) = \begin{cases} \mathtt{parent}(B) & \text{if } B \in \mathtt{tasks}(\mathcal{C}) \cup \{\mathtt{root}(\mathcal{C})\}, \\ \mathtt{tParent}(B) & \text{otherwise .} \end{cases}$$

Similarly, define the task parent of a trace U as $\mathtt{tparent}[U] = \mathtt{tParent}(\mathtt{head}[U])$.

From the partition $\mathcal{J}$ of nodes in the computation tree $\mathcal{C}$, we can define a ***trace tree*** $\mathcal{J}(\mathcal{C})$ as follows. For any traces $U, U' \in \mathcal{J}$, there is an edge $(U, U') \in \mathcal{J}(\mathcal{C})$ if and only if $\mathtt{parent}(\mathtt{head}[U']) \in U$.[6] The properties of traces and the fact that traces partition $\mathcal{C}$ into disjoint subtrees together imply that $\mathcal{J}(\mathcal{C})$ is also a tree.

These properties imply that if an active trace U' is a descendant of a trace U, then $\mathtt{head}[U']$ is a descendant of *all* active nodes in U, as the following lemma shows.

Lemma 6.2. *Consider active traces $U, U' \in \mathcal{J}$, with $U \neq U'$. Let $D \in U'$ be an active node, and suppose that* $D \in \mathit{desc}(\mathit{head}[U])$ *(i.e., U' is a descendant trace of U). Then for any active node $B \in U$, we have* $B \in \mathit{ances}(D)$.

Proof. Intuitively, since all active nodes in trace U fall along a single chain, and since the descendant trace U' is active, $\mathtt{head}[U']$ must be the descendant of some node in the active

[6] The function $\mathtt{parent}()$ refers to the parent in the computation tree $\mathcal{C}$, not in the trace tree $\mathcal{J}(\mathcal{C})$.

chain of U. In fact, $\texttt{head}[U']$ must be a descendant of the bottom of the active chain of U, for otherwise, we would have a branch in the set of active nodes of U.

More formally, first we can show that $\texttt{head}[U] \in \texttt{pAnces}(\texttt{head}[U'])$. Since $D \in U'$, we know $D \in \texttt{desc}(\texttt{head}[U'])$ by Property 1. By our original assumption, we have also $D \in \texttt{desc}(\texttt{head}[U])$ and thus, one of $\texttt{head}[U']$ and $\texttt{head}[U]$ must be a proper ancestor of the other. Suppose for contradiction that $\texttt{head}[U'] \in \texttt{pAnces}(\texttt{head}[U])$. Then along the path from D up to the root of the tree, we would have $\texttt{head}[U] \in U$ in between two nodes from U' (D and $\texttt{head}[U']$), which would contradict Property 2 for U'.

Thus, since $\texttt{head}[U] \in \texttt{pAnces}(\texttt{head}[U'])$ and $\texttt{head}[U']$ itself is active, we can show that all nodes $B \in (U \cap \texttt{vTree}(\mathcal{C}))$ must satisfy $B \in \texttt{ances}(\texttt{head}[U']) \subseteq \texttt{ances}(D)$. By Property 5, only a single head-to-leaf path of U can be active: let $Y = \texttt{leaf}(U \cap \texttt{vTree}(\mathcal{C}))$, i.e., Y is the bottom node in this path. Since $B \in U$ is active, we also know that $B \in \texttt{ances}(Y)$. Thus, if we can show that $Y \in \texttt{ances}(\texttt{head}[U'])$, we have that $B \in \texttt{ances}(D)$.

Suppose for contradiction that $Y \notin \texttt{ances}(\texttt{head}[U'])$. Let $X = \texttt{LCA}(\texttt{head}[U'], Y)$. Then X is also an active node in U with $X \neq Y$, because both $\texttt{head}[U']$ and Y are descendants of $\texttt{head}[U]$. The node X must be an active P-node, since $\texttt{vTree}(\mathcal{C})$ can only branch at P-nodes. But then by Property 3, the children of X should belong to different traces than X, contradicting the fact that both X and Y belong to U. Thus we have $Y \in \texttt{ances}(\texttt{head}[U'])$, yielding $B \in \texttt{ances}(D)$, as desired. □

From Lemma 6.2, we can show that to perform an ancestor query between a memory operation u and an active node Y in the computation tree, it suffices to perform an ancestor query of their respective traces.

Theorem 6.3. *On any step t that is executing a memory operation v, consider any active node $Y \in \texttt{vTree}^{(t)}(\mathcal{C})$. Then $Y \in \texttt{ances}(v)$ if and only if $\mathsf{trace}(Y)$ is an ancestor of $\mathsf{trace}(v)$ in $\mathcal{J}(\mathcal{C})$.*

Proof. Consider both directions.

- Suppose that $Y \in \texttt{ances}(v)$. In $\mathcal{C}$, the path starting from v and ending at Y corresponds to a path of traces in $\mathcal{J}(\mathcal{C})$, starting at $\mathsf{trace}(v)$ and ending at $\mathsf{trace}(Y)$. Since $\mathcal{J}(\mathcal{C})$ is a valid tree, $\mathsf{trace}(Y)$ must be an ancestor of $\mathsf{trace}(v)$.

- Let $U = \mathsf{trace}(Y)$ and $U' = \mathsf{trace}(v)$, and suppose that U is an ancestor of U'. Consider the two cases for U and U'.

  First, suppose that $U = U'$. Since v is a currently executing memory operation, we know $D = \texttt{parent}(v) \in U$ is active and a leaf in $\texttt{vTree}^{(t)}(\mathcal{C})$. By Property 5, all active nodes in U must fall along a single chain. Thus, D must be the bottom node in this active chain. Since Y is also active, we must have $Y \in \texttt{ances}(D) \subset \texttt{ances}(v)$.

  Otherwise, consider $U \neq U'$. Since U is an ancestor of U', we have $\texttt{head}[U'] \in \texttt{desc}(\texttt{head}[U])$. Since $D = \texttt{parent}(v) \in U'$, we know that $D \in \texttt{desc}(\texttt{head}[U])$. Also, by our original assumption, $Y \in U$ is also active. Therefore, by Lemma 6.2, we have $Y \in \texttt{ances}(D)$, which implies $Y \in \texttt{ances}(v)$.

□

The XConflict Algorithm

Recall that CWSTM instruments memory accesses and tests for transaction conflicts on each memory access by querying the XConflict data structure. This section sketches the high-level algorithm for XConflict, the data structure that CWSTM uses to check for transaction conflicts.

Section 6.3 described a conceptual algorithm (in Figure 6-5) for conflict detection according to Definition 5.17 that can be translated into a simple, but potentially inefficient implementation. In particular, one can perform the steps in Figure 6-5 by walking up and down the computation tree. To check for a conflict between X and a memory operation u, one can first walk up the tree to check that X has no `ABORTED` ancestors, as well as find Y, the nearest active transactional ancestor of X. Then by walking up the tree from u, we can determine whether Y is an ancestor of u. Definition 5.17 states that a conflict between X and u occurs only if X has no `ABORTED` ancestors and Y is not an ancestor of u. Unfortunately, this query may be inefficient for computations with deep nesting of parallelism or transactions, since it requires walking up and down the computation tree.

Instead, XConflict performs a more asymptotically efficient query that takes advantage of traces. As before, let Y be the nearest active transactional ancestor of X. Intuitively, XConflict determines whether Y is an ancestor of u without explicitly finding Y. More specifically, XConflict manages to find U_Y, the trace that contains Y. As described later in Theorem 6.4, it turns out that testing whether U_Y is an ancestor of u is sufficient to determine whether Y is an ancestor of u. XConflict does not lock on any queries, because many of the subroutines only need to perform simple ABA tests [66] to see if anything changed between the start and end of the query.

The pseudocode in Figure 6-9 gives a high-level description of the XConflict algorithm. First, the code in lines 1–4 handle the simple cases of the query. The code in lines 1–2 covers the case where X and an active node u belong to the same trace. In this case, a conflict between X and u is impossible. The code in lines 3–4 handles the case where X has an `ABORTED` ancestor. By Definition 6.1, Property 2, there is no conflict between X and any memory operation u.

Otherwise, the remainder of the code handles the more complex case, where X has no `ABORTED` ancestor and X and u belong to different traces. In line 5, XConflict finds B, the nearest transactional ancestor of X that belongs to an active trace. Figure 6-11 illustrates three cases, representing the possible locations of B in the computation tree. Let $U_B = \mathsf{trace}(B)$. Although U_B is active, B may be active or inactive. For cases (a) or (b), we find B with a simple lookup of `xParent`(X). Case (c) involves first finding U, the highest completed ancestor trace of $\mathsf{trace}(X)$ and then performing a simple lookup of `xparent`$[U]$. Section 6.5 describes how to find the highest completed ancestor trace. If B happens to be the root of the computation tree (as checked in line 6), then X is part of the outermost transaction, and there is no conflict.

Otherwise, execution proceeds to the final cases in Figure 6-9. In line 9, XConflict finds Z, the highest active transaction in U_B, the trace containing B. There are two subcases to consider, depending on the relationship between Z and B.

- If Z exists and is an ancestor of B, as shown in the left of Figure 6-12, then XConflict is in the case given by lines 11–13. If U_B is an ancestor of u, we conclude that X has

```
XCONFLICT($X,u$)
    // For any computation-tree node $X$ and any active memory-operation $u$

    // Test for simple base cases
 1  if trace($X$) = trace($u$)
 2      return "no conflict"
 3  if some ancestor transaction of $X$ is aborted
 4      return "no conflict: $X$ aborted"

 5  Let $B$ be the nearest transactional ancestor of $X$
        belonging to an active trace.
 6  if $B$ = root($C$)                              // committed at top level
 7      return "no conflict: $X$ committed to root"
 8  $U_B \leftarrow$ trace($B$)

 9  Let $Z$ be the highest active transaction in $U_B$

10  if $Z \neq$ null and $Z$ is an ancestor of $B$
11      if $U_B$ is an ancestor of $u$
12          return "no conflict: $X$ committed to $u$'s ancestor"
13      else return "conflict with $Z$"
14  else $Y \leftarrow$ xparent[$U_B$]
15      if $Y$ = null or trace($Y$) is an ancestor of $u$
16          return "no conflict: $X$ committed to $u$'s ancestor"
17      else return "conflict with $Y$"
```

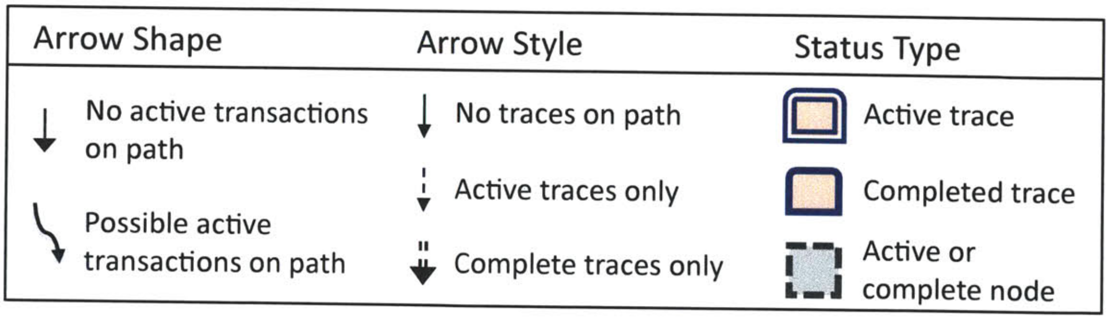
Figure 6-9: Pseudocode for the XConflict algorithm.

committed to an ancestor of u. Figure 6-12 (a) and (b) show the possible scenarios where U_B is an ancestor of u: either B is an ancestor u, or B has committed to some transaction Y that is an ancestor of u.

- Suppose that Z is not an ancestor of B (or that Z does not exist), as shown in the left of Figure 6-13. Then XConflict follows the case given in lines 15–17. Let Y be the transactional parent of U_B. Since B has no active transactional ancestor in U_B, it follows that B has committed to Y. Thus, if trace(Y) is an ancestor of u, we conclude that X has committed to an ancestor of u, as shown in Figure 6-13.

We now prove the correctness of the XConflict algorithm more formally.

Theorem 6.4. *Let X be any node in the computation tree, and let u be a currently executing memory access. Then* XCONFLICT(X,u) *(in Figure 6-9) reports a conflict if and only if* XCONFLICT-ORACLE(X,u) *(from Figure 6-5) reports a conflict.*



Arrow Shape	Arrow Style	Status Type
↓ No active transactions on path	↓ No traces on path	▣ Active trace
	⇣ Active traces only	□ Completed trace
↓ Possible active transactions on path	⇊ Complete traces only	⬚ Active or complete node

Figure 6-10: The definition of arrows used to represent paths in Figures 6-11, 6-12 and 6-13.

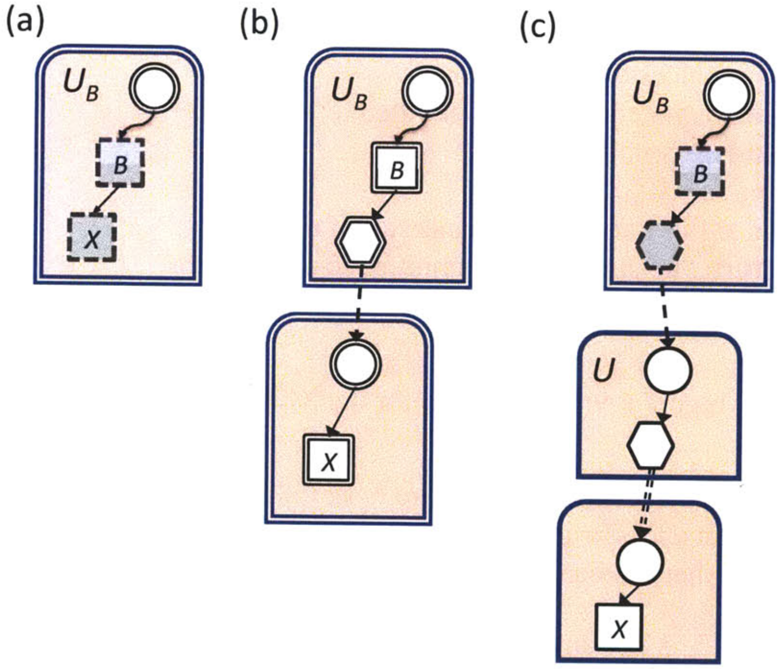


Figure 6-11: The three possible scenarios in which B is the nearest transactional ancestor of X that belongs to an active trace. Arrows represent paths between nodes (i.e., many nodes are omitted): see Figure 6-10 for definitions. (a) X and xParent(X) both belong to the same active trace. (b) X belongs to an active trace and xParent(X) belongs to an ancestor trace of trace(X). (c) X belongs to a complete trace, U is the highest completed ancestor trace of X, and B is the xparent[U].

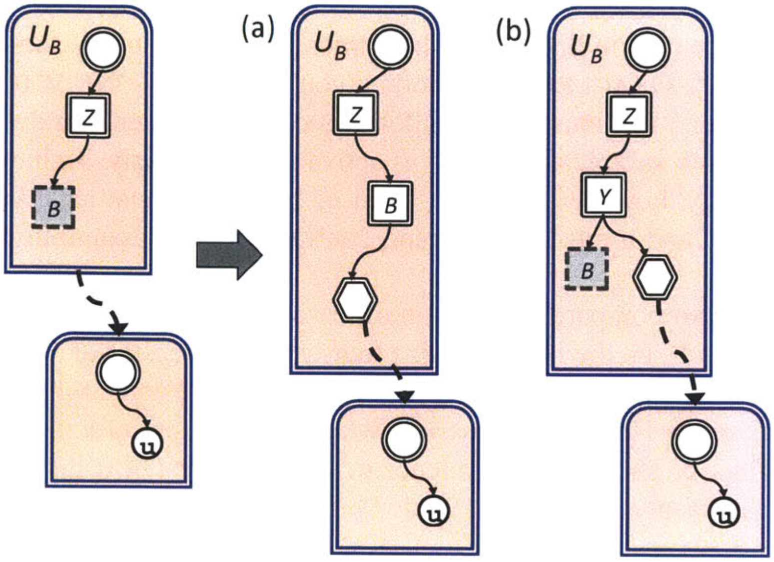


Figure 6-12: The possible scenarios in which the highest active transaction Z in U_B is an ancestor of B, and U_B is an ancestor of u (i.e., line 11 of Figure 6-9 returns TRUE). Arrows represent paths between nodes (i.e., many nodes are omitted): see Figure 6-10 for definitions. The block arrow shows implication from the left side to either (a) or (b).

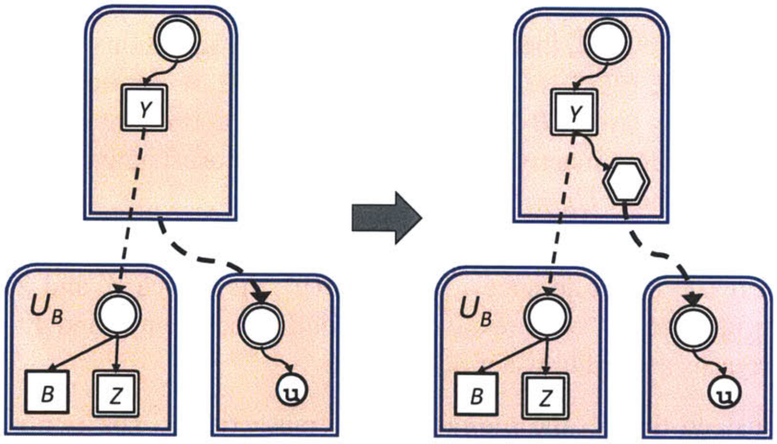


Figure 6-13: The scenario in which the highest active transaction Z in U_B is not an ancestor of B, and $Y = \texttt{xparent}[U_B]$ is an ancestor of u (i.e., line 15 of Figure 6-9 returns TRUE). The block arrow shows implication from the left side to the situation on the right. The case when no active transaction in U_B exists (i.e., $Z = \texttt{null}$) looks similar, except with Z removed from the diagram.

Proof. If X has an ABORTED ancestor, then both XCONFLICT and XCONFLICT-ORACLE both report no conflict. Thus, we focus on the remaining case, where all the ancestors of X are either COMMITTED or PENDING.

First, consider the case in lines 1–2, where $\text{trace}(X) = \text{trace}(u)$. Assume for contradiction that XCONFLICT-ORACLE(X,u) reports a conflict, but XCONFLICT(X,u) does not. For a conflict to occur, there must be a parallel relationship between X and u (i.e., LCA(X,u) is a P-node), with both subtrees of LCA(X,u) active simultaneously. To have both subtrees active simultaneously, however, the right child of LCA(X,u) must have been stolen, and thus the traces for X and u must be different, contradicting the assumption that the traces were the same.

Otherwise, we can compare the remaining cases of Figure 6-9 to the oracle method from Figure 6-5. Let Y' be the nearest active transactional ancestor of X, which is utilized by the query XCONFLICT-ORACLE(X,u). Let $U_{Y'}$ be the trace containing Y'. Because Y' is active, $U_{Y'}$ is active. By Theorem 6.3, we know $U_{Y'}$ is an ancestor of u if and only if Y' is an ancestor of u. Thus, to show the equivalence of XCONFLICT-ORACLE and XCONFLICT, we show that in all cases, XCONFLICT finds $U_{Y'}$ and performs an ancestor query between $U_{Y'}$ and $\text{trace}(u)$.

In line 5, XConflict first finds B, the nearest transactional ancestor of X belonging to an active trace. If $B = \texttt{root}(C)$, then lines 6–7 and XCONFLICT-ORACLE report the same answer, since X is a top-level transaction.

Otherwise, let U_B be the trace containing B, and let Z be the highest active transaction in U_B, if one exists. Consider the two cases for Z:

- Suppose that Z exists and is an ancestor of B. If Z exists, then Y', the closest active transactional ancestor of X, must satisfy $Y' \in U_B$. We also have $Y' \in \texttt{desc}(Z)$, because all active nodes in U_B fall along a chain and there is at least one transaction Z along that chain.

  As shown in Figure 6-12, there are two possible configurations for U_B, depending on whether B is active (PENDING) or inactive (COMMITTED). For both configurations, we can show that $U_B = U_{Y'}$. If B is active, then by construction, we must have $B = Y'$, and thus $U_B = U_{Y'}$. If B is inactive, then Y' is also the closest active ancestor transaction of B, and we still have $U_B = U_{Y'}$.

  Since $U_B = U_{Y'}$ for both configurations, line 11 performs the correct ancestor query. In line 11, if $U_{Y'}$ is not an ancestor of $\text{trace}(u)$, then since Y' and Z both belong to $U_{Y'}$, aborting either Y' or Z resolves the conflict between u and Y'. Thus, line 13 correctly returns a valid transaction to abort.

- Suppose instead that Z, the *highest* active transaction in U_B, is not an ancestor of B, or Z does not exist. Then no active transaction in U_B is an ancestor of B. Let $Y = \texttt{xParent}(U_B)$. Since U_B is active, Y must be active. Thus, we know $Y = Y'$, i.e., Y is the nearest active transactional ancestor of X. Consequently, XConflict performs the correct test in lines 15–17. Figure 6-13 illustrates this case when there is no conflict.

In this proof thus far, we have assumed that the XConflict data structure does not change concurrently while a query XCONFLICT executes. One can show, however, that the query does not miss reporting any conflicts when concurrent updates are allowed. The only changes to the data structure that could affect correctness are the splitting of traces, and the commit or abort of transactions. Neither of these actions can introduce a new conflict where one did not exist before, however. Since the query itself is read-only, one can retry the query to detect whether any traces or transactions change.[7] □

To execute the steps in Figure 6-9 efficiently, the XConflict algorithm maintains a data structure which supports several operations.

Definition 6.2. *The **XConflict data structure** supports following operations:*

1. *Find* `xParent`(X), *the transactional parent of a given node X in the computation tree.*
2. *Compute* trace(X) *for any node X in the computation tree (e.g., lines 1, 8, and 15 in Figure 6-9).*
3. *Find the highest active transaction within a trace (line 9).*
4. *Find an aborted ancestor trace (line 3).*
5. *Perform an ancestor query within a trace (line 10).*
6. *Perform an ancestor queries between traces (lines 11 and 15).*

Operation 1 is straightforward to maintain by having each node in the computation tree maintain a pointer to its transactional parent. The other operations, however, require some more sophisticated data structures (discussed in Section 6.5) to guarantee that they run efficiently, i.e., asymptotically in $O(1)$ time.

6.5 CWSTM Data Structures

This section describes the XConflict data structure, which enables a TM runtime using a Cilk-like scheduler to support the conflict query presented in Figure 6-9. In particular, it describes the four main components of XConflict which are used to support the operations in Definition 6.2. First, to support Operation 2, XConflict maintains trace objects. Next, to find the highest active transaction within a trace (Operation 3), XConflict augments each trace object with a transaction stack. To facilitate Operation 4, XConflict groups traces together into "supertraces." Finally, XConflict maintains state within a trace to answer local ancestor queries (Operation 5), and global data structures to answer global ancestor queries (Operation 6).

Trace Maintenance

The XConflict data structure maintains traces to enable efficient trace membership queries, i.e., to perform operation 2, which computes trace(X) for any computation tree node $X \in$ `nodes`(C), in $O(1)$ time in the worst case. We give only a high-level overview of the

[7] Technically, since the commit or abort of a transaction during a query can only eliminate a conflict, it is safe to let transactions change status during the query.

scheme for trace maintenance in this section. Readers interested in additional details should see [25,47], since these traces are similar to the local-tier of the SP-hybrid data structure described.

To support trace membership queries, XConflict organizes computation-tree nodes belonging to a single trace as follows. Within a single trace U, each node $D \in U$ is associated with $\texttt{tParent}(D)$, its closest task-node ancestor. The task nodes in U are grouped into sets, called "trace bags." Each bag b maintains a pointer to a trace, denoted by $\mathsf{traceField}[b]$. Since a trace may contain many trace bags, most of the complexity of traces lies in maintaining $\mathsf{traceField}[b]$ efficiently.

Bags are merged dynamically in a way similar to SP-bags [45] in the local tier of SP-hybrid [25,47] using a disjoint-sets data structure [38, Chapter 21]. Since each trace can be executed by only a single worker at a time, we do not need to lock the data structure on update (UNION) operations. The difference in our setting is that XConflict uses only one kind of bag (instead of two in SP-bags).

When steals occur, a global ***steal lock*** is acquired, and then a trace is split into multiple traces, as in the global tier of SP-hybrid [25,47]. The difference for XConflict is that a trace splits into three traces (instead of five in SP-hybrid).[8] It turns out that trace splits can be done in $O(1)$ worst-case time by simply moving a constant number of bags. When the trace constant-time split completes, the steal lock is released.

To compute $\mathsf{trace}(X)$, i.e., to query which trace a node X belongs to, XConflict performs the operation

$$\mathsf{trace}(X) = \mathsf{traceField}[\text{FIND-BAG}(\texttt{tParent}(X))] .$$

These queries (in particular, FIND-BAG) take $O(1)$ worst-case time as in SP-hybrid [25,47]. Merging bags uses a UNION operation and takes $O(1)$ amortized time, but an optimization [47] gives a technique that improves UNIONs to worst-case $O(1)$ time whenever the amortization might adversely increase the program's critical path.

Highest Active Transaction

XConflict also augments the trace objects with additional data to support finding the highest active transaction within a trace (operation 3) in $O(1)$ time.

For each task node S, XConflict maintains a field *nextx*[S] that stores a pointer to the nearest active descendant transaction of S. Maintaining this field for *all* S-nodes is expensive, and so instead, XConflict maintains it only for some S-nodes as follows. Let $\mathrm{S} \in U$ be an active task S-node such that either $\mathrm{S} = \texttt{head}[U]$ or S is the left child of a P-node and $\texttt{xParent}(\mathrm{S}) = \texttt{parent}(\texttt{parent}(\mathrm{S}))$. Then *nextx*[S] is defined to be the nearest, active descendant transaction of S in U. Otherwise, *nextx*[S] $= \texttt{null}$.

Finding the highest active transaction simply entails a call to *nextx*[$\texttt{head}[U]$], which takes $O(1)$ time. The complication is maintaining the *nextx* values, especially subject to dynamic trace splits.

[8] Traces in CWSTM also differ from those in SP-hybrid in that a single trace U can be executed by multiple workers. In CWSTM, one worker p can begin executing a function F in a trace U, and then a different worker p' can resume the execution of F after a sync as part of the same trace U. For SP-hybrid, the continuation of the sync would begin a different trace.

To maintain *nextx*, XConflict keeps a stack of S-nodes in U for which *nextx* is defined. Initially, push head$[U]$ onto the stack. For each of the following scenarios, let S be the S-node on the top of the stack. Whenever encountering a transactional S-node X, check *nextx*[S]. If *nextx*[S] $=$ null, then set *nextx*[S] $= X$. Otherwise, do nothing. Whenever completing a transaction X, check *nextx*[S]. If *nextx*[S] $= X$, then set *nextx*[S] $=$ null. Otherwise, do nothing. Whenever encountering a task S-node S$'$. If *nextx*[S] $=$ null, do nothing. Otherwise, push S$'$ onto the stack. Whenever completing a task S-node S$'$, pop S$'$ from the stack if it is on top of the stack.

Finally, XConflict maintains these *nextx* values even subject to trace splits. Consider a split of trace U into three traces U_1, U_2, and U_3, rooted at S, S$_1$, and S$_2$, respectively. Since CWSTM steals from the highest P-node in the computation tree, S$_1$ must be the highest, active, task-node descendant of S that is the left child of a P-node. Thus, either S$_1$ is the second S-node on U's stack, or S$_1$ is not on U's stack.

If S$_1$ is on U's stack, then *nextx*[S] is defined to be an ancestor of S, and XConflict leaves it as such. Moreover, since S$_1$ is on the stack, *nextx*[S$_1$] is defined appropriately. Simply split the stack into two just below S to adjust the data structure to the new traces. Suppose instead that S$_1$ is not on U's stack. Then the *nextx*[S] may be a descendant of S$_1$ (or it is undefined). Set *nextx*[S$_1$] $=$ *nextx*[S] and *nextx*[S] $=$ null. Then split the stack below S, and prepend S$_1$ at the top of its stack. The necessary stack splitting takes $O(1)$ worst-case time. This stack splitting occurs while holding the steal lock and traces are being split.

Supertraces

This section describes XConflict's data structure to find the highest completed ancestor trace of a given trace, which is used as a subprocedure for Operation 4. To facilitate this type of query, XConflict groups traces together into "supertraces." Grouping traces into supertraces also facilitates faster aborts: when aborting a transaction in a trace U, we need only mark some of the supertrace children of U as ABORTED, not the entire subtree in $\mathcal{C}$. This section also describes how XConflict performs the abort of a transaction.

To explain the properties of supertraces, we first require some definitions on $\mathcal{J}(\mathcal{C})$, the tree of traces. Recall that a trace U is active if status(head$[U]$) is active (PENDING or PENDING_ABORT); otherwise, U is completed. For a completed trace U, we say that U is ABORTED if head$[U]$ has been marked by CWSTM as ABORTED; otherwise, we say that U is COMMITTED. (Note that head$[U]$ is a task node, not a transaction.) CWSTM can mark some task nodes as ABORTED to speed up the abort of an enclosing transaction. Finally, we define a "representative trace" of a completed trace U.

Definition 6.3. *On a step t, for any complete trace $U \in \mathcal{J}(\mathcal{C})$, define the* ***representative trace*** *of U, denoted by* strRep(U), *as closest ancestor trace of U which is either ABORTED, or whose parent trace is active.*[9]

At any point during program execution, the completed traces $U \in \mathcal{J}(\mathcal{C})$ are partitioned into ***supertraces***. For each complete trace U, let strace$[U]$ denote the supertrace for U. Each supertrace object $K =$ strace$[U]$ is conceptually a set of traces, i.e., $K \subseteq \mathcal{J}(\mathcal{C})$. Also,

[9] Ancestors include U itself, and so strRep$(U) = U$ if U itself is ABORTED or has an active parent trace.

each supertrace maintains a ***high trace***, denoted by $\mathsf{high}[K]$, which stores the highest completed ancestor trace within K.

First, we investigate the ***supertrace algorithm***, i.e., the algorithm XConflict uses to maintain supertraces. Supertraces are implemented using a disjoint-sets data structure [38, Chapter 21]. In particular, XConflict uses the data structure of Gabow and Tarjan that supports MAKE-SET, FIND (implementing $\mathsf{strace}[U]$), and UNION operations, all in $O(1)$ amortized time when unions are restricted to a tree structure (as they are in our case). The supertrace algorithm operates as follows:

- When a trace U is created, create an empty supertrace for U (i.e., $\mathsf{strace}[U] = \emptyset$).
- When the trace U completes (i.e., at a `sync` instruction, which finishes $\texttt{head}[U]$), do the following:
  1. Acquire the (global) steal lock.
  2. Add U to its own supertrace. In other words, create a supertrace $K \leftarrow \{U\}$, and set $\mathsf{strace}[U] \leftarrow K$.
  3. Merge the supertrace set of U with the supertrace set of the traces U' which are children of U, if U' is part of a COMMITTED supertrace.[10] More formally, for each child U' of U, let $K' = \mathsf{strace}[U']$.
     (a) If $\mathsf{high}[\mathsf{strace}[U']]$ is COMMITTED, merge the two supertraces, i.e., execute UNION$(\mathsf{strace}[U], K')$.
     (b) Otherwise, $\mathsf{high}[\mathsf{strace}[U']]$ is ABORTED, and $\mathsf{strace}[U']$ is left unchanged.
  4. Set the high trace of $\mathsf{strace}[U]$ to U, i.e., $\mathsf{high}[\mathsf{strace}[U]] \leftarrow U$.
  5. Release the steal lock.

Although a trace U may complete on a step t, the supertrace of U will not be merged into its parent supertrace until some later step $t' > t$ when the parent supertrace completes.

XConflict is able to exploit the structure of supertraces to optimize on transaction aborts. A naive algorithm to abort a transaction X must walk the entire computation subtree rooted at X, changing all of X's COMMITTED descendants to ABORTED. This walk can be potentially expensive, since large portions of the subtree of X may be completed.

XConflict, however, only walks the active portions of the subtree rooted at X in $U = \mathsf{trace}(X)$, not $\mathcal{C}$. When it hits a boundary between an active trace U and a completed trace U' (i.e., a node $B \in U$, $D \in U'$, with $\texttt{parent}(D) = B$), it sets D to be ABORTED, and does not walk the descendants of D.

The following theorem shows that XConflict preserves desired invariants for supertraces.

Theorem 6.5. *XConflict's supertrace algorithm maintains the following invariants on supertraces. On any given step, consider the tree of traces $\mathcal{J}(\mathcal{C})$. Consider any two (possibly equal) complete traces $U, V \in \mathcal{J}(\mathcal{C})$ with $V = \texttt{strRep}(U)$. Then, we have $\mathsf{strace}[U] = \mathsf{strace}[V]$, and one of two cases holds:*

[10] Maintaining a list of all child traces is not difficult. We keep a linked list for each node in the trace tree and add to it whenever a trace splits.

1. *If V is* COMMITTED*, then* high[strace[U]] $= V$.

2. *If V is* ABORTED*, then* high[strace[U]] *is also* ABORTED.

Proof. We can prove this result by induction, on the actions taken to maintain the computation tree, traces, and supertraces.

In the base case, we begin with a single trace U_0 which is active. Since strace[U] $= \emptyset$ for any active trace U, the theorem holds vacuously.

For the inductive step, we consider the actions which can change supertraces— the `sync` and `xabort` instructions.

- Consider a `sync` which completes a trace U. First, suppose that U is a leaf if $\mathcal{J}(C)$. Then the supertrace algorithm creates a supertrace $K = \{U\}$, set strace[U] $= K$, and set high[K] $= U$. Then both conditions of the theorem hold for U trivially (whether head[U] is COMMITTED or ABORTED).

  Otherwise, suppose that U is not a leaf in $\mathcal{J}(C)$ but has children traces. For any trace W in the subtree of $\mathcal{J}(C)$ rooted at U, consider what happens to W and strace[W] before and after the update of supertraces performed by the supertrace algorithm. If $W = U$, then the theorem holds as in the base case. If $W \neq U'$, then W must belong to the subtree of some trace U', where U' is a child trace of U. Since U is already completed, we know that both U' and W must also be completed, and thus the theorem holds inductively for both U' and W. Let $K_W =$ strace[W], let $V =$ strRep(W), and consider the various cases:

  1. high[K_W] is COMMITTED.

     First, we can show that $V =$ strRep(W) $= U'$ for the trace U' which is a child of U. Suppose for contradiction that $V \neq U'$ for all children U'. By the definition of $V =$ strRep(W), we know V must be the closest ancestor trace along the path from W to U' in $\mathcal{J}(C)$ such that either V is ABORTED, or the parent of V is an active trace. We can derive a contradiction for each of these two cases:

     - Suppose that V is ABORTED. By Case 2 of the inductive hypothesis on W and V we know that strace[W] $=$ strace[V] and high[strace[W]] is ABORTED. But these facts contradict our original assumption that high[K_W] is COMMITTED.
     - Suppose that the parent of V is an active trace. Since U is the only active trace in its subtree, we must have $V = U'$ for some child U' of U, contradicting our original assumption that $V \neq U'$.

     Now that we know $V = U'$, consider what happens after the supertrace update operation. XConflict sets strace[W] $=$ strace[V] $=$ strace[U], and we also have high[strace[U]] $= U$. Because U' is COMMITTED, and the parent trace of U is active, we also have that strRep(W) $= U$ after the update. Thus, the conditions of Case 1 are satisfied for W.

  2. high[K_W] is ABORTED.

     In this case, the operations on supertraces leave $K_W =$ strace[W] unchanged, because this trace is not involved in any of the UNION operations.

We can also show that the completion of U cannot change strRep(W). Consider $V =$ strRep(W) before the update. We must have V being ABORTED. Otherwise, if V were COMMITTED, then by Case 1 of the inductive hypothesis, we would have high[strace[W]] $= V$ being COMMITTED, contradicting the assumption that high[K_W] is ABORTED. Since V is ABORTED, after the completion of U, we still have V as the closest ancestor trace of W which is either ABORTED or whose parent is active, i.e., we still have $V =$ strRep(W).

Thus, since strace[W] and strRep(W) both remain unchanged, the conditions of Case 2 remain satisfied for W.

- Consider an xabort of a transaction X. This xabort may need to abort transactions Y nested inside X which belong to an active trace $U =$ trace(Y). The xabort marks any trace V which is a child of U as ABORTED.

  Marking V as ABORTED cannot break the invariants for V or any of its descendant traces, however. To be more precise, because V has an active parent, for any complete trace W which is a descendant of V, strRep(W) remains the same. Also, for any W with strRep(W) $= V$, changing V to ABORTED only shifts us from Case 1 to Case 2 of Theorem 6.5.

□

All update operations on supertraces take place while holding the steal lock. Unlike the updates for trace or stack splitting, however, these updates on supertraces do not coincide with steals. As Section 6.6 argues, however, one can still bound the number of supertrace-update operations. In particular, the number of updates is asymptotically identical to the number of steals. This amortization is similar to the "global tier" of SP-hybrid [25].

Ancestor Queries

This section describes how XConflict performs ancestor queries. First, it describes how XConflict performs "local" ancestor queries (operation 5), i.e., queries between nodes belonging to the same trace. Then, it discusses how XConflict performs "global" ancestor queries (operation 6) — queries between nodes belonging to different traces. Both of these queries can be performed in $O(1)$ worst-case time.

CWSTM executes a trace using a single worker at a time, with each trace being executed in depth-first (e.g.., left-to-right) order. We thus view a trace execution as a depth-first execution of a computation (sub)tree (or a depth-first tree walk). To perform ancestor queries on a depth-first walk of a tree, we can associate with each tree node u the ***discovery time*** $d[u]$, indicating when u is first visited (i.e., before visiting any of u's children), and the***finish time*** $f[u]$, indicating when u is last visited (i.e., when all of u's descendants have finished). (This same labeling appears in depth-first search in [38, Section 22.3].) These timestamps are sufficient to perform ancestor queries in constant time.

In the context of XConflict, we simply need to associate a "time" counter with each trace. Whenever a trace splits, this counter's value is copied to the new traces.

To handle global ancestor queries, XConflict requires a more complicated data structure. Since the computation tree does not execute in a depth-first manner, the same discovery/finish time approach does not work for ancestor queries between traces. Instead, XConflict keeps two total orders on the traces dynamically using order-maintenance data structures [24,42]. These two orders provide enough information to determine the ancestor-descendant relationship between two nodes in the tree of traces. These total orders are updated while holding the global steal lock. Since our global ancestor-query data structure resembles the global series-parallel-maintenance data structure in SP-hybrid, I refer the reader to [25] for the details of the data structure. As in SP-hybrid, each query has a worst-case cost of $O(1)$, and trace splits have an amortized cost of $O(1)$.

Correctness of the global ancestor queries relies on Property 4 of traces— the property that a thief processor steals a subtree from the highest available P-node owned by the victim.

6.6 Theoretical Performance Bounds

The section bounds the running time of an CWSTM program in the absence of conflicts. The bound covers the cost to maintain the XConflict data structure and the time to check for conflicts *assuming that all accesses are writes*. Checking for conflicts with multiple readers or transaction aborts both add more work to the computation, and these slowdowns are not included in the analysis. First, we can argue that each query to the XConflict data structure requires $O(1)$ time in the worst case. Then we can bound the time required to maintain the XConflict data structure during a program's execution.

The next theorem bounds the time for each XConflict query.

Theorem 6.6. *Each query* XCONFLICT(X,u) *requires* $O(1)$ *time.*

Proof. We shall prove this result by showing that each line of code in Figure 6-9 requires $O(1)$ time to execute.

In line 1, looking up the trace for a node in the computation tree (Operation 2) requires finding the representative trace bag in a disjoint-sets data structure. Using the disjoint-sets data structure of Gabow and Tarjan, this operation requires $O(1)$ time.

In line 3, there are two cases, depending on whether X has an aborted ancestor.

1. If $\mathsf{trace}(X)$ is active, XConflict simply checks whether `status`$(X) =$ ABORTED.[11]
2. If $\mathsf{trace}(X)$ is completed, then we find the supertrace of $\mathsf{trace}(X)$ and use the properties of supertraces in Theorem 6.5. More precisely, XConflict checks whether $V = \mathsf{high}[\mathsf{strace}[\mathsf{trace}(X)]]$ is ABORTED. If not, then we know any potential ABORTED ancestor of X must be within an active trace U which is a proper ancestor trace of V. Thus, XConflict checks for an aborted ancestor of `xParent`(V), as in the first case.

In line 5, to find the B, the nearest transactional ancestor of X that belongs to an active trace, XConflict first finds the node X' that is the nearest transactional ancestor of X (either $X' = X$ if X is a transaction, or $X' =$ `xParent`(X) otherwise). If $\mathsf{trace}(X')$ is active, then $B = X'$. Otherwise, $\mathsf{trace}(X')$ is complete, and XConflict finds the closest ancestor trace of

[11] Recall that the `xabort` instruction of a transaction Y in an active trace U is responsible for walking Y's subtree within U and changing the status field of any $X \in$ `xDesc`$(Y) \cap U$ to ABORTED.

trace(X') which has an active parent trace, i.e., it looks up $V = \text{high}[\text{strace}[\text{trace}(X')]]$.[12] Then we can find $B = \texttt{xParent}(\texttt{head}[V])$.

Lines 6–8 require constant time, since they involve only comparison of computation-tree nodes and a trace lookup.

In line 9, finding the highest active transaction within a trace U_B (Operation 3) requires looking at the top of the S-node stack within U_B and can be done in $O(1)$ time.

In line 10, since B and Z are both within U_B, XConflict performs a local ancestor query (Operation 5). This local query requires a comparison of node timestamps, which occurs in constant time.

Finally, in lines 11 and 15, XConflict performs a global ancestor query by comparing two traces in two order-maintenance data structures. These data structures, as described in [24,42] and used for SP-hybrid in [25], support queries in $O(1)$ time.

□

The key insight in this analysis is to amortize the cost of updates that hold the steal lock against the number of steals, similar to the proof of performance of SP-hybrid in [47]. One important feature of XConflict's "global" data structures is that they have $O(|\mathcal{J}(C)|)$ total update cost, where $|\mathcal{J}(C)|$ represents the total number of traces in the computation. The analysis makes the pessimistic assumption that while the steal lock is held, only the worker holding the steal lock makes any progress.

The following theorem states the running time of an CWSTM program under nice conditions, giving bounds for both Cilk's normal randomized work-stealing scheduler and for a round-robin work-stealing scheduler (as in [47]).

Theorem 6.7. *Consider an CWSTM program with work T_1 and span (critical-path length) T_∞ in which all memory accesses are writes. Suppose that the program, augmented with XConflict, is executed on P processors and has no transaction conflicts.*

1. *When using a randomized work-stealing scheduler, the program runs in $O(T_1/P + P(T_\infty + \lg(1/\varepsilon)))$ time with probability at least $1 - \varepsilon$.*
2. *When using a round-robin work-stealing scheduler, the program runs in $O(T_1/P + PT_\infty)$ worst-case time.*

Intuitively, Theorem 6.7 bounds the overhead of XConflict algorithm itself. These bounds nearly match those of a Cilk program without XConflict's conflict detection. The only difference is that the T_∞ term is multiplied by a factor of P. In most cases, since we expect $PT_\infty \ll T_1/P$, these bound represents only constant-factor overheads beyond optimal.

These XConflict bounds translate to bounds on completion time of an CWSTM program under optimistic conditions. For illustration, consider a program where all concurrent paths access disjoint sets of memory as writes. The overhead of maintaining the XConflict data structures is $O(T_1/P + PT_\infty)$. Each memory access queries the XConflict data structure at most once. Since each query requires only $O(1)$ time, the entire program runs in $O(T_1/P + PT_\infty)$ time.

[12] The trace V should have an active parent trace, rather than being ABORTED, since XConflict determined in line 3 that X does not have an ABORTED ancestor.

The CWSTM design described in this section does not provide any reasonable performance guarantees when multiple readers are allowed. There are two reasons for this problem. First, concurrent reads to an object may contend on the access stack to that object. Second, even in the case where concurrent read operations never wait to acquire an access stack lock, it appears that write operation may need to check for conflicts against potentially many readers in a reader list (some of which may have already committed). Therefore, a write operation is no longer a constant time operation, and it seems the work of the computation might increase proportionally to the number of parallel readers to an object. It is an interesting open question whether one can extend the CWSTM design to achieve stronger bounds when concurrent reads are permitted.

6.7 Related Work

I conclude this chapter by describing other work related to nested parallelism in TM.

Transactions with nested parallelism have been extensively studied in the context of database systems [56]. Wing et al. [125] proposed a design of a system that supports transactions with nested parallelism in the context ML, a persistent programming language. To my knowledge, CWSTM [3] represents the first design of a system supporting transactions with nested parallelism in the context of transactional memory.

Since the original work on CWSTM [3] was published, other designs for TM that support transactions with nested parallelism have been proposed in the literature. Barreto et al. [23] describe an STM design for transactions with nested parallelism which is similar in design to CWSTM, but which uses a simplified algorithm for conflict detection. This design uses bitvectors to represent sets of transactions, which enables a TM system to answer conflict queries using only a few bit operations. The algorithm has the limitation, however, that only a constant number of transaction ids can be active at once (e.g., 64) before the ids must be reclaimed. Although the design of [23] is likely to be more efficient in practice, it uses a scheme for reclaiming transaction ids that seems tricky to analyze theoretically. It would be interesting to explore whether one can provide worst-case theoretical bounds for this design.

Others have also proposed and implemented prototype designs of TM that support transactions with nested parallelism. Volos et al. in [122] describe NePalTM, a design and implementation of a TM system that integrates atomic blocks with OpenMP. Baek et al. [19] describe NesTM, a STM design that supports transactions with nested parallelism. The NesTM design focuses on having low overhead for a single level of nesting of transactions. In their system the commit of nested transactions requires work proportional to the size of the transaction readset and writeset. In another paper [20], the authors also discuss hardware support for transactions with nested parallelism. These designs may be more practical to implement, but it is an open question how well they work on programs with large nesting depths.

Chapter 7

Ownership-Aware TM

This chapter explores the challenge of designing transactional memory (TM) that provides strong semantic guarantees for "open-nested" transactions. Chapters 5 and 6 described semantics and runtime support for TM with nested parallelism and "closed-nested" transactions. TM with closed-nesting semantics has the potential problem, however, that it does not provide any mechanism for programmers to eliminate conflicts between transactions that they deem to be "unnecessary." Thus, researchers have proposed the idea of open-nested transactions [98, 103], a special type of nested transaction that allows programmers to eliminate some kinds of transaction conflicts. Unfortunately, as I discuss in this chapter, open-nested transactions significantly complicate the semantics of TM because their use breaks the traditional guarantees of serializability. Without this guarantee, it can be difficult for programmers to reason about programs that use open-nested transactions.

In this chapter, I describe ***ownership-aware TM***,[1] a design for TM that uses information about which memory locations a software module owns to make open-nested transactions safer and more intuitive to use. Ownership-aware TM supports open-nesting of transactions, but still provides clean memory-level semantics and provable guarantees of "abstract serializability" to programmers. Ownership-aware TM demonstrates that a parallel-programming platform can support composable synchronization using open-nested transactions and still provide provable guarantees of safety and correctness to programmers.

This section reviews the concept of open-nested transactions through a series of code examples, using these examples to motivate the utility of ownership-aware TM. First, I present a code example which shows that existing TM mechanisms for open nesting can improve the performance of code using TM. Unfortunately, as I show using additional examples, an unconstrained use of these mechanisms can also lead to anomalous program behavior. I then give a brief overview of ownership-aware TM, a TM design which allows open nesting but is able to eliminate some of these anomalies. Finally, I conclude this section by giving an outline of the rest of this chapter.

Motivation for Open-Nesting in TM

A simple code example illustrates the benefits of open-nested transactions.

[1] Ownership-aware TM represents joint work [5] with Kunal Agrawal and I-Ting Angelina Lee.

```
void Insert2(int k, int v) {
  B[k] = v;
  num_inserts++;
}
```

Figure 7-1: A modification of the `Insert` method from Figure 5-1. The `Insert2` method performs the same computation as `Insert`, but also updates a global counter `num_inserts` that counts the number of inserts performed.

In a closed TM system, consider a modification to the code from Figure 5-1, where every call to `Insert` is replaced with a call to `Insert2` from Figure 7-1. In the `Insert2` method, each insert also updates a global counter counting the number of inserts. With this modification, Schedule 3 from Figure 5-1 is no longer serializable, even when variables `a` through `f` are all distinct, because the inserts happen in the order X_2, Y_2, X_4, Y_4, and thus neither X or Y sees consecutive values of the counter `num_inserts`. Because transactions X and Y both update the same counter, they will likely conflict when run concurrently, ***even if*** the counter updates are a tiny fraction of the execution time of X and Y.

An application developer may find this kind of transaction conflict between X and Y undesirable for several reasons. First, since neither X or Y read the value of this counter, the increments to `num_inserts` conceptually commute with each other, and Schedule 3 still appears "abstractly serializable" at the level of program semantics, even though it is not technically serializable when viewed at the level of memory reads and writes.

Second, a small change to the `Insert` method which does not affect the correctness of the insert now has a significant impact on the performance of X and Y, because it controls whether X and Y conflict or not. Closed-nested transactions in TM have the benefit of being composable: the programmer who writes X and Y can still write correct code without knowing the implementation details of a library `Insert` method. Closed-nesting semantics also have the downside, however, that a library writer coding the `Insert` method cannot isolate a library user from the performance implications of code changes such as the one in Figure 7-1.

To improve the performance of TM in such examples, researchers have proposed mechanisms for an ***open-nested commit*** of a transaction [98, 103]. Conceptually, a transaction A which is "open-nested" inside X can commit state to global memory even before the outer transaction X completes. For example, consider an open TM system that executes the code shown in Figure 7-2. When A_1 completes, its changes to `B` and `num_inserts` are committed immediately, even though X has not completed yet and may later abort. In this example, all insert calls occur inside open-nested transactions. The TM system ignores any conflicts between transactions X and Y due to any memory accesses inside the open-nested transactions A_1, A_2, B_1, and B_2. Open nesting provides a loophole in the strict guarantee of transaction serializability by allowing an outer transaction to "ignore" the memory operations of its open-nested subtransactions.

As Moss [102] argues, the use of open-nested transactions requires reasoning about TM programs at multiple levels of abstraction. Using the ***open-nesting methodology***, whenever a transaction X has an open-nested transaction Z, X should not care about the memory operations inside Z when checking for conflicts. Instead, X may need to acquire an ***abstract***

```
// Transaction X
atomic {
  int r1, r2;
  r1 = A[a];      // X1
  open_atomic {   // A1
    Insert2(a, r1);
  }
  r2 = A[b];      // X2
  open_atomic {   // A2
    Insert2(b, r2);
  }
  A[c] = r1+r2;   // X3
}
```

```
// Transaction Y
atomic {
  int r1, r2;
  r1 = A[d];       // Y1
  open_atomic {    // B1
    Insert2(d, r1);
  }
  r2 = A[e];       // Y2
  open_atomic {    // B2
    Insert2(e, r2);
  }
  A[f] = r1+r2;    // Y3
}
```

Figure 7-2: Two concurrent transactions X and Y, each with open-nested transactions (indicated by the blocks labeled with the open_atomic keyword). The transaction X performs each call to Insert2 inside open-nested transactions, A_1 and A_2. Similarly, Y has open-nested transactions B_1 and B_2.

lock based on the high-level operation that Z represents so that the TM system can check for transaction conflicts for X using this abstract lock. Also, if X aborts, then, the TM system may need to execute a ***compensating action*** to undo the effect of an open-nested subtransaction Z which has already committed. Moss [102] illustrates use of open nesting with an application that uses a B-tree. Ni et al. [105] describe a software TM system that supports the open-nesting methodology.

Anomalous Behavior Using Open-Nested Commits

Unfortunately, a significant gap exists between the high-level programming methodology of open nesting of [102, 105] and the memory-level mechanism of an open-nested commit described in [98, 103]. Ideally, a programmer would like to reason about only the high-level methodology. Since a TM system only provides guarantees about transactions only at the level of reads and writes to memory, however, a programmer also needs to understand the memory-level semantics of open-nested commits to correctly apply the methodology. By using open-nested commits, a TM system can permit some nonserializable schedules which a programmer considers desirable, but understanding exactly which schedules are allowed requires careful reasoning about memory-level semantics.

Unfortunately, once a TM system with open nesting admits some desirable nonserializable schedules, the proverbial cat is out of the bag. As far as the memory semantics are concerned, it seems difficult to prohibit additional program behaviors that might arguably be undesirable. For example, Figure 7-3 shows a program execution allowed by the open-nested commit mechanisms described in [98, 103]. In this example, it is possible for all transactions X, Z, Y_1, and Y_2 to commit, even though X does not appear to execute atomically. Transaction X reads inconsistent data, since Y_2 writes to A[b] between X's reads of A[a] and A[b]. Thus, the "snapshot" of memory seen by X when it begins is different from its snapshot part way through its computation.

```
   // Transaction X
 1 atomic {
 2   int r1, r2;
 3   r1 = A[a];
     // Transaction Z
 4   open_atomic {
 5      A[i]++;
 6   }
 7   r2 = A[b];
 8   A[c] = (r1+r2);
 9 }
```

```
   // Transaction Y1
10 atomic {
11   A[i]++;
12 }

   // Transaction Y2
13 atomic {
14   A[b] = A[i];
15   A[b]++;
16 }
```

Figure 7-3: A nonserializable program execution permitted by TM that uses an open-nested commit mechanism. Suppose X with an open-nested transaction Z runs concurrently with Y_1 and Y_2. In an execution order given by the lines 1–6, 10–16, and then 7–9, transaction X can read an "inconsistent" value for `A[b]`, because Y_1 and Y_2 can appear to interleave after the open-nested transaction Z completes, but before X commits.

```
 1 bool Contains(int i) {
 2   bool empty;
 3   open_atomic {
 4     if (tbl.size>0) {
 5        empty = false;
 6     }
 7   }
 8   if (!empty) {
 9     return (tbl.A[i]>0);
10   }
11   return false;
12 }
```

```
13 void Insert(int k, int v)
14 {
15   atomic {
16     tbl.A[k] = v;
17     tbl.size++;
18   }
19 }
```

Figure 7-4: A flawed implementation of a table data structure using an open-nested transaction. In this implementation, `tbl.A[i]` being 0 indicates that element i is not in the table.

```
   // Transaction X
 1 atomic {
 2   if (!Contains(5)) {
 3     Insert(5, 15);
 4   }
 5 }
```

```
   // Transaction Y
 6 atomic {
 7   if (!Contains(5)) {
 8     Insert(5, 10);
 9   }
10 }
```

Figure 7-5: Two transactions X and Y which no longer appear atomic because of open-nested transactions. Because both X and Y use the flawed table data structure implemented in Figure 7-4, and the `Contains` method uses an open-nested transaction incorrectly, transactions X and Y no longer appear to execute atomically. In particular, the following execution order is possible: lines 1–2, lines 6–10, and then lines 3–5.

The code in Figure 7-4 illustrates how the open-nested commit mechanism can admit subtle program behaviors that affect the composability of transactions. This code describes an implementation of a simple table which supports the `Contains(x)` and `Insert(x,y)` methods, but which contains a subtle flaw. Since the `size` field is the primary source of transaction conflicts between table operations, the `Contains` method "optimizes" its search method by checking `size` within an open-nested transaction.

Using TM with open nesting, in any sequence of `Contains` or `Insert` operations, each individual operation still appears atomic. Thus, in transaction X in Figure 7-5, we might expect that if the `Contains` operation returns false, then the key can be safely inserted into the hash table without adding duplicates.

Unfortunately, one cannot correctly call both `Contains` and `Insert` inside a transaction X and still have X appear to be atomic. A TM that uses open-nested commits allows the entire transaction Y_1 to execute between line 2 and line 3 of transaction X, since the open-nested commit does not add the read of `tbl.size` (in line 4 of Figure 7-4) to the readset of X. Thus, in this example, that composability of transactions is not preserved. When using open nesting, simply ensuring the atomicity of individual transactions is not sufficient to guarantee composability.

Of course, the programs in Figures 7-3 and 7-4 are clearly contrived examples. In particular, in Figure 7-4, transactions cannot be partitioned into clear abstraction levels, with each level accessing disjoint memory locations, as Moss [102] suggests may be necessary in the open-nesting methodology. These examples demonstrate, however, that for open-nested transactions, the distinction between the abstract program model and the low-level memory model is much more significant than for closed or flat nesting.

These examples also suggest that to avoid anomalous program behavior that can be tricky to reason about, that one should constrain the behavior of open-nested transactions, possibly by limiting the memory locations that transactions are allowed to access. The programs in Figures 7-3 and 7-4 are "obviously" pathological, and one might argue that a "reasonable" program should not exhibit such behavior. Unfortunately, it is not as obvious what exactly what constitutes a reasonable program. Furthermore, because the TM runtime is unaware of the different levels of memory, it is not clear that a TM runtime can easily guarantee or even check whether a program is reasonable.

One potential reason for the apparent complexity of open nesting is that the mechanism and the methodology make different assumptions about memory. Consider a transaction Y that is open nested inside transaction X. The open-nesting methodology requires that X ignore the "lower-level" memory conflicts generated by Y, while the open-nested commit mechanism ignores *all* the memory operations inside Y. Say Y accesses two memory locations ℓ_1 and ℓ_2, and suppose that X does not care about changes made to ℓ_2, but does care about ℓ_1. The TM system cannot distinguish between these two accesses and commits both in an open-nested manner, leading to anomalous behavior.

On the other hand, researchers have presented specific program examples [35, 102, 105] that safely use an open-nested commit mechanism. Moss [102] illustrates the use of open nesting with an application that uses a B-tree. Ni et al. [105] describe a software TM system that supports the open-nesting methodology. Carlstrom et al. [35] describe the Transactional Collection Classes, a set of transactional data structures that use an open-nested commit mechanism. These examples work primarily because the inner (open-nested) transactions never write to any data that is accessed by the outer transactions. Unfortunately, these examples offer relatively little in the way of formal programming guidelines which one can follow to have *provable* guarantees of safety when using open-nested commits for other applications. Moreover, since these examples require only two levels of nesting, it is not obvious how one can correctly use open-nested commits in a program with more than two levels of abstraction.

Contributions

This chapter describes ownership-aware transactional memory, which associates memory locations and nested transactions with particular "transactional modules." By providing a TM runtime with information about which module "owns" a particular memory location, the runtime can take into account the "abstraction level" of modules and memory when detecting conflicts.

Ownership-aware TM is designed to bridge the gap between memory-level mechanisms for open nesting and the high-level methodology by explicitly integrating the notions of ***transactional modules*** (Xmodules) and ownership into a TM system. The structure imposed by ownership allows a language and runtime to enforce properties needed to provide provable guarantees of "safety" to the programmer. More specifically, this chapter describes the primary features of ownership-aware TM, namely:

1. A concrete set of guidelines for sharing of data and interactions between Xmodules.
2. A description of how Xmodules and ownership can be specified in a Java-like language using ownership types.
3. The OAT model, an operational model for ownership-aware TM, which uses a new ownership-aware commit mechanism.
4. A proof that if a program follows the proposed guidelines for Xmodules, then the OAT model guarantees ***serializability by modules***, which is a generalization of "serializability by levels" used in database transactions.
5. A proof that if all transactions in the process of aborting obey certain restrictions on their memory footprint, then a computation executing under the OAT model cannot enter a semantic deadlock.

The ownership-aware commit mechanism is a compromise between an open-nested and a closed-nested commit. When a transaction X commits, a change to memory location ℓ is either committed globally if ℓ belongs to the module of X or propagated to X's parent transaction if ℓ does not belong to the module of X. Unlike an ordinary open-nested commit, the ownership-aware commit treats memory locations differently depending on which Xmodule owns the location.[2] An ownership-aware commit is the same as an open-nested commit if no Xmodule ever accesses data belonging to other Xmodules. Thus, one can still guarantee serializability by modules using open-nested commits when Xmodules do not share data. This observation explains why researchers [35, 105] have found it natural to use open-nested transactions, in spite of the apparent semantic pitfalls.

[2] The ownership-aware commit is still just a mechanism. Programmers must still use it in combination with abstract locks and compensating actions to implement the full methodology.

In this chapter, we distinguish between the variations of nested transactions as follows. We say that a transaction Y is ***vanilla open-nested*** (inside its parent transaction) when referring to a TM system which performs the open-nested commit of Y. We say that Y is ***safe-nested*** when referring to the ownership-aware TM system which performs the ownership-aware commit of Y. Finally, we say that a transaction Y is an open-nested transaction when we are referring to the abstract methodology, rather than a particular implementation with a specific commit mechanism.

Chapter Outline

The remainder of this chapter is organized as follows. Section 7.1 presents an overview of ownership-aware TM and highlights key features using an example application. Section 7.2 describes language constructs for specifying Xmodules and ownership. Section 7.3 extends the transactional-computation framework and the TCO operational model from Chapter 5 to create the OAT model, an operational model for a TM runtime that incorporates Xmodules and ownership. Section 7.4 gives a formal definition of serializability by modules, and a proof that the OAT model guarantees this definition. Section 7.5 provides conditions under which the OAT model does not exhibit semantic deadlocks. Section 7.6 discusses related work, and Section 7.7 concludes this chapter.

7.1 Ownership-Aware Transactions

This section gives an overview of ownership-aware TM. First, this section motivates the need for the concept of ownership in TM by describing an example application which can benefit from open nesting. Next, it introduces the notion of an Xmodule and informally explain the programming guidelines when using Xmodules. Finally, it highlights some of the key differences between ownership-aware TM and a TM that uses ordinary open-nested commits. This section presents the intuitive descriptions of the concepts in ownership-aware TM, deferring formal definitions until later sections.

The Book Application

We now investigate the book application, an example application for which one might use open-nested transactions. This example is similar to the one in [102], but it includes data sharing between nested transactions and their parents, and has more than two levels of nesting.

Since the open-nesting methodology is designed for programs with multiple levels of abstraction, the book application is a modular application. Suppose that this application concurrently accesses a database of many individuals' book collections. The database stores records in a binary search tree, keyed by name. Each node in the binary search tree corresponds to a person and stores a list of books in his/her collection. The database supports queries by name, as well as updates that add a new person or a new book to a person's collection. The database also maintains a private hash table, keyed by book title, to support a reverse query: given a book title, it returns a list of people who own the book.

Finally, the application also wants the database to log changes on disk for recoverability. Whenever the database is updated, it inserts metadata into the buffer of a logger to record the change that just took place. Periodically, the application is able to request a checkpoint operation which flushes the buffer to disk.

The book application can be decomposed into five natural modules — the user application (`UserApp`), the database (`DB`), the binary search tree (`BST`), the hash table (`HashTable`), and the logger (`Logger`). The `UserApp` module calls methods from the `DB` module when it wants to insert into the database, or query the database. The database in turn maintains internal metadata and calls the `BST` module and the `HashTable` module to answer queries and insert data. Both user application and the database may call methods from the `Logger` module.

If the modules use open-nested transactions, a TM system with vanilla open-nested commits can result in nonintuitive outcomes. Consider the example where a transactional method `A` from the `UserApp` module tries to insert a book b into the database and the insert is an open-nested transaction. The method `A` (which corresponds to transaction X) calls an insert method in the `DB` module and passes b (the `Book` object) to be inserted. This insert method generates an open-nested transaction Y. Suppose that Y writes to some field of the book b (memory location ℓ_1), and it also writes some internal database metadata (location ℓ_2). After a vanilla open-nested commit of Y, the modifications to both ℓ_1 and ℓ_2 become visible globally. Assuming that the `UserApp` does not care about the internal state of the database, committing the internal state of the `DB` (ℓ_2) is a desirable effect of open nesting, because this commit increases concurrency, since other transactions can potentially modify the database in parallel with X without generating a conflict. The `UserApp` does, however, care about changes to the book b. Thus, the commit of ℓ_1 breaks the atomicity of transaction X. A transaction Z in parallel with transaction X can access this location ℓ_1 after Y commits, before the outer transaction X commits.[3] To increase concurrency, we want the method from `DB` to commit changes to its own internal data; we do not, however, want it to commit the data that `UserApp` cares about.

To enforce this kind of restriction, we need some notion of ***ownership*** of data: if the TM system is aware of the fact that the book object "belongs" to the `UserApp`, then it can decide not to commit `DB`'s change to the book object globally. For this purpose, we introduce the notion of ***transactional modules***, or Xmodules. When a programmer explicitly defines Xmodules and specifies the ownership of data, the TM system can make the correct judgement about which data to commit globally.

Xmodules and the Ownership-Aware Commit Mechanism

The ownership-aware TM system requires that programs be organized into Xmodules. Intuitively, an Xmodule M is as a stand-alone entity that contains data and transactional methods. An Xmodule owns data that it privately manages, and it uses its methods to provide public services to other modules. During program execution, a call to a method

[3] Abstract locks [102] do not address this problem. Abstract locks are meant to disallow other transactions from noticing the fact that the book was inserted into the `DB`. They do not usually protect the individual fields of the book object itself.

from Xmodule M generates a transaction instance (e.g., X). If this method in turn calls another method from an Xmodule N, an additional transaction Y, safe nested inside X, is created only if $M \neq N$. Therefore, defining an Xmodule automatically specifies safe-nested transactions.

In the ownership-aware TM system, every memory location is owned by exactly one Xmodule. If a memory location ℓ belongs to the readset or writeset of a transaction X, the ownership-aware commit of X commits this access globally only if X is generated by the same Xmodule that owns ℓ. In this case, we say that X is "responsible" for that access to ℓ. Otherwise, the read or write to ℓ is propagated up to the readset or writeset of X's parent transaction. That is, the TM system behaves as though X was a closed-nested transaction with respect to location ℓ.

For ownership-aware TM to behave "nicely," we must restrict interactions between Xmodules. For example, in the TM system, some transaction must be "responsible" for committing every memory access. Similarly, the TM system should guarantee some form of serializability. If Xmodules could arbitrarily call methods from or access memory owned by other Xmodules, then these two properties might not be satisfied.

One way to restrict Xmodules would be to allow a transaction to access only objects that belongs to its own Xmodule. This condition might severely restrict the expressiveness of the program, however, since it does not allow an Xmodule to pass an object that it owns as a parameter to a method that belongs a different Xmodule. Ownership-aware TM is able to impose weaker restrictions on the interactions between Xmodules, while still guaranteeing desirable properties.

Rules for Xmodules

Ownership-aware TM uses Xmodules to control both the structure of nested transactions and the sharing of data between Xmodules (i.e., to limit which memory locations a transaction instance can access). Xmodules are arranged as a ***module tree***, denoted as $\mathcal{D}$. An Xmodule $N \in \mathcal{D}$ is a child of M if N is "encapsulated by" M. The root of $\mathcal{D}$ is a special Xmodule called `world`. Each Xmodule is assigned an ***xid*** by visiting the nodes of $\mathcal{D}$ in a left-to-right depth-first search order and assigning ids in increasing order, starting with $\texttt{xid}(\texttt{world}) = 0$. Therefore, `world` has the minimum `xid`, and "lower-level" Xmodules have larger `xid` numbers.

Definition 7.1. *Ownership-aware TM imposes two rules on Xmodules based on the module tree:*

1. **Rule 1**: *A method of an Xmodule M can access a memory location ℓ directly only if ℓ is either owned by M or an ancestor of M in the module tree. This rule means that an ancestor Xmodule N of M may pass data down to a method belonging to M, but a transaction from module M cannot directly access any "lower-level" memory.*

2. **Rule 2**: *A method from M can call a method from N only if N is the child of some ancestor of M and $\texttt{xid}(N) > \texttt{xid}(M)$ (i.e., if N is "to the right" of M in the module*

tree). This rule means that an Xmodule can call methods of some (but not all) lower-level Xmodules.[4]

The intuition behind these rules is as follows. Xmodules have methods to provide services to other higher-level Xmodules, and Xmodules maintain their own data in order to provide these services. Therefore, a higher-level Xmodule can pass its data to a lower-level Xmodule and ask for services. A higher-level Xmodule should not directly access the internal data belonging to a lower-level Xmodule.

If Xmodules satisfy Rules 1 and 2, TM can have a well-defined ownership-aware commit mechanism in which some transaction is always "responsible" for every memory access (proved in Section 7.3). In addition, these rules and the ownership-aware commit mechanism guarantee that transactions satisfy the property of "serializability by modules" (proved in Section 7.4).

One potential limitation of ownership-aware TM is that some "cyclic dependencies" between Xmodules are prohibited. The ability to define one module as being at a lower level than another is fundamental to the open-nesting methodology. Thus, this formalism requires that Xmodules be partially ordered: if an Xmodule M can call Xmodule N, then conceptually M is at a higher level than N (i.e., $\mathtt{xid}(M) < \mathtt{xid}(N)$), and thus N cannot call M. If two components of the program call each other, then, conceptually, neither of these components is at a higher-level than the other, and ownership-aware TM would require that these two components be combined into the same Xmodule.

Xmodules in the Book Application

Consider a Java implementation of the book application. It may have the following classes: `UserApp` as the top-level application that manages the book collections, `Person` and `Book` as the abstractions representing book owners and books, `DB` for the database, `BST` and `HashTable` for the binary search tree and hash table maintained by the database, and `Logger` for logging the metadata to disk. In addition, there are some other auxiliary classes: tree node `BSTNode` for the `BST`, `Bucket` in the `HashTable`, and `Buffer` used by the `Logger`.

For ownership-aware TM, not all of a program's classes are meant to be Xmodules. Some classes only wrap data. In this example, we can identify five Xmodules— `UserApp`, `DB`, `BST`, `HashTable`, and `Logger`— which are stand-alone entities with encapsulated data and methods. Classes such as `Book` and `Person`, on the other hand, are data types used by `UserApp`. Similarly, classes like `BSTNode` and `Bucket` are data types used by `BST` and `HashTable` to maintain their internal state.

We can organize the Xmodules of the book application into the module tree shown in Figure 7-6. `UserApp` is encapsulated by `world`, `DB` and `Logger` are encapsulated under `UserApp`, and both `BST` and `HashTable` are encapsulated under `DB`. By organizing Xmodules this way, the ownership of data falls out naturally, i.e., an Xmodule owns certain pieces of data if the data is encapsulated under the Xmodule. For example, the instances of `Person` or `Book` are owned by `UserApp` because they should only be accessed by either `UserApp` or its descendants.

[4] An Xmodule can, in fact, call methods within its own Xmodule or from its ancestor Xmodules, but ownership-aware TM models these calls differently. These cases are explained at the end of this section.

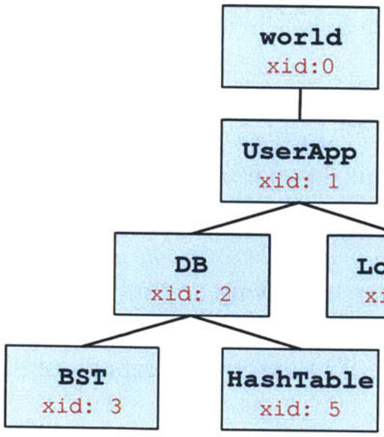


Figure 7-6: A module tree $\mathcal{D}$ for the book application described in Section 7.1. The xid's are assigned according to a left-to-right depth-first tree walk, numbering Xmodules in increasing order, starting with xid(world) = 0.

Let us consider the implications of Definition 7.1 for the example. Due to Rule 1, all of DB, BST, HashTable, and Logger can directly access data owned by UserApp, but the UserApp cannot directly access data owned by any of the other Xmodules. This rule corresponds to standard software-engineering rules for abstraction: the "high-level" Xmodule UserApp should be able to pass its data down, allowing lower-level Xmodules to access that data directly, but UserApp itself should not be able to directly access data owned by lower-level Xmodules. Due to Rule 2, the UserApp may invoke methods from DB, DB may invoke methods from BST and HashTable, and every other Xmodule may invoke methods from Logger. Thus, Rule 2 allows all the operations required by the example application. As expected, the UserApp can call the insert and search methods from the DB and can even pass its data to the DB for insertion. More importantly, notice the relationship between BST and Logger. The BST Xmodule can call methods from Logger, but the BST cannot pass data it owns directly into the Logger. It can, however, pass data owned by the UserApp to the logger, which is all this application requires.

Advantage of Ownership-Aware Transactions

One of the major problems with vanilla open-nesting is that some transactions can see inconsistent data. Say a transaction Z is open-nested inside transaction X. Let v_0 be the initial value of location ℓ, and suppose that Z writes value v_1 to location ℓ and then commits. Now, a transaction Y in parallel with X can read this location ℓ, write value v_2 to ℓ, and commit, all before X commits. Therefore, X can now read this location ℓ and see the value v_2, which is neither the initial value v_0 (the value of ℓ when X started), nor v_1 which was written by X's inner transaction Z.[5] This behavior might seem counterintuitive.

[5] This example is similar to the code in Figure 7-3 except with Y_1 and Y_2 merged together into a single transaction Y and both accessing the same location $\ell = \texttt{A[i]} = \texttt{A[b]}$.

Now, consider the same example for ownership-aware transactions. Say X is generated by a method of Xmodule M and Z is generated by a method of Xmodule N. There are two cases to consider:

1. If N owns ℓ, X cannot access ℓ, since $\texttt{xid}(M) < \texttt{xid}(N)$ (by Definition 7.1, Rule 2), and no transaction from a higher-level module can access data owned by a lower-level module (by Definition 7.1, Rule 1). Thus, the problem does not arise.

2. If N does not own ℓ, the ownership-aware commit of Z does not commit the changes to ℓ globally and ℓ will be propagated to X's writeset. Therefore, if Y tries to access ℓ before X commits, the TM system detects a conflict. Thus X cannot see an inconsistent value for ℓ.[6]

Callbacks

At first glance, ownership-aware TM makes assumptions about Xmodules which seem somewhat restrictive. For example, an Xmodule M is prohibited from making a ***callback*** — calling another transactional method from M or a proper ancestor of M. In fact, ownership-aware TM can be extended to permit some callbacks.

More precisely, if a method X from Xmodule M calls another method Y from an ancestor Xmodule N, this new call does not generate a new safe-nested transaction instance. Instead, Y is subsumed in X using flat (or closed) nesting. Recall that Rule 1 in Definition 7.1 allows X to access data belonging to N or any of its ancestors directly. Therefore, we can treat any data access by a flat (or closed) nested transaction Y as being accessed by X directly, provided that Y and its nested transactions access only memory belonging to N or N's ancestors. We say that Y is a ***proper callback*** method for Xmodule N if its nested calls are all proper callback methods belonging to Xmodules which are ancestors of N. The formal model for ownership-aware TM described in Section 7.3 assumes a system with only proper callbacks. It models these callbacks as direct memory accesses, allowing us to ignore callbacks in the formal definitions.

Closed-Nested Transactions

In ownership-aware TM, every method call that crosses an Xmodule boundary automatically generates a safe-nested transaction. Ownership-aware TM can also provide closed-nested transactions, however, with appropriate specifications of ownership. If an Xmodule M owns no memory, but only operates on memory belonging to its proper ancestors, then transactions of M are effectively closed nested. In the extreme case, if the programmer specifies that all memory is owned by the `world` Xmodule, then all changes in any transaction's readset or writeset are propagated upwards. Thus, all ownership-aware commits behave exactly as closed-nested commits.

[6] For simplicity, I have described the case where Z is directly nested inside X. The case where Z is more deeply open nested inside X behaves in a similar fashion.

7.2 Ownership Types for Xmodules

When using ownership-aware transactions, the Xmodules and data ownership in a program must be specified for two reasons. First, the ownership-aware commit mechanism depends on these concepts. Second, a TM runtime can guarantee some notion of serializability only if a program has Xmodules which conform to the rules in Definition 7.1. This section describes language constructs and the ***OAT type system*** — a type system that can be used to specify Xmodules and ownership in a Java-like language. By extending the ownership types in the ***BLS type system***, the type system of Boyapati, Liskov, and Shrira [33], the OAT type system statically enforces some of the restrictions described in Definition 7.1.

This section first reviews the BLS type system [33] and then describes how the OAT type system extends this system to enforce some of the restrictions in Definition 7.1. Next, it presents code for parts of the book application described in Section 7.1. Finally, this section gives a more formal description of the guarantees provided by the OAT type system.

The BLS Ownership Type System

The BLS type system provides a mechanism for specifying ownership of objects. This type system enforces the properties stated in the following lemma.

Lemma 7.1. *The BLS type system enforces the following properties:*

1. *Every object has a unique owner.*
2. *The owner can be either another object, or* `world`*.*
3. *The ownership relation forms an* ownership tree *(of objects) rooted at* `world`*.*
4. *The owner of an object does not change over time.*
5. *An object a can access another object b directly only if b's owner is either a, or one of a's proper ancestors in the ownership tree.*

The BLS type system requires ownership annotations to class definitions and type declarations to guarantee Lemma 7.1. Every class type `T1` has a set of associated ownership tags, denoted `T1`$\langle f_1, f_2, \ldots f_n \rangle$. The first formal f_1 denotes the owner of the current instance of the object (i.e., `this` object). The remaining formals $f_2, f_3, \ldots f_n$ are additional tags which can be used to instantiate and declare other objects within the class definition. The formals get assigned with actual owners $o_1, o_2, \ldots o_n$ when an object a of type `T1` is instantiated. By parameterizing class and method declarations with ownership tags, BLS permits owner polymorphism. Thus, one can define a class type (e.g., a generic hash table) once, but instantiate multiple instances of that class with different owners in different parts of the program.

BLS enforces the properties in Lemma 7.1 by performing the following checks:

1. Within the class definition of type `T1`, only the tags $\{f_1, f_2, \ldots f_n\} \cup \{$`this`,`world`$\}$ are visible. The `this` ownership tag represents the object itself.
2. Within a class definition, a variable c_2 with type `T2`$\langle f_2, \ldots \rangle$ can be assigned to a variable c_1 with type `T1`$\langle f_1, \ldots \rangle$ if and only if `T2` is a subtype of `T1` and $f_1 = f_2$.

3. If an object a's tags are instantiated to be $o_1, o_2, \ldots o_n$ when a is created, then in the ownership tree, o_1 must be a descendant of o_i for all i in the range $\{2, 3, \ldots n\}$ (denoted by $o_1 \preceq o_i$ henceforth).

It is shown in [33] that these type checks guarantee the properties of Lemma 7.1.

In some cases, to enable the type system to perform check 3 locally, the programmer may need to specify a `where` clause in a class declaration. For example, suppose that the class declaration of type `T1` has formal tags $\langle f_1, f_2, f_3 \rangle$, and inside `T1`'s definition, some object of type `T2` is instantiated with ownership tags $\langle f_2, f_3 \rangle$. The type system cannot determine whether or not $f_2 \preceq f_3$. To resolve this ambiguity, the programmer must specify `where (`f_2 `<=` f_3`)` at the class declaration of type `T1`. When an instance of type `T2` object is instantiated, the type system then checks that the `where` clause is satisfied.

The OAT Type System

Although the ownership tree described in [33] exhibits some of the same properties as the module tree described in Section 7.1, the BLS the type system and ownership scheme does not enforce two major requirements of ownership-aware TM:

- In BLS, any object can own other objects. In contrast, ownership-aware TM requires that only Xmodules can own other objects.
- In BLS, an object can call any of its ancestor's siblings. Definition 7.1 dictates that an Xmodule M can only call its ancestor's siblings to the right in the module tree.

With these requirements in mind, we can extend the BLS type system to create the OAT type system.

The extensions to handle the first requirement are straightforward. The OAT type system explicitly distinguishes objects and Xmodules by requiring that Xmodules extend from a special `Xmodule` class. The OAT type system only allows classes that extend `Xmodule` to use `this` as an ownership tag. In the context of the BLS ownership tree, this restriction creates a tree where all the internal nodes are Xmodules and all leaves are non-Xmodule objects. If we ignore any order imposed on the children of an Xmodule, for ownership-aware TM, the module tree (as described in Section 7.1) is essentially the ownership tree with all non-Xmodule objects removed.

The second requirement is more complicated to enforce. First, the OAT type system extends each owner instance o to have two fields: ***name***, represented by *o.name*; and ***index***, represented by *o.index*. The name field is conceptually the same as an ownership instance in the BLS type system. The index field is added to help the compiler to infer ordering between children of the same Xmodule in the module tree. The OAT type system allows the programmer to pass `this[i]` as the ownership tag (i.e., with an index i) instead of `this`. Similarly, one can use `world[i]` as an ownership tag. Indices enable the type system to infer an ordering between two sibling Xmodules M and N. For instance, if an Xmodule L instantiates M and N with owners `this[i]` and `this[i+1]`, respectively, then M appears to the left of N in the module tree.

Finally, for technical reasons, the OAT system prohibits all Xmodules M from declaring primitive fields. If M had primitive fields, then in BLS, these fields are owned by the M's parent. Since this property seems counterintuitive, the OAT system disallows primitive fields for Xmodules.

In summary, the OAT type system performs these checks:

1. Within the class definition of type `T1`, only the tags $\{f_1, f_2, \dots f_n\} \cup \{\texttt{this,world}\}$ are visible.

2. In a class declaration, a variable c_2 with type `T2`$\langle f_2, \dots \rangle$ can be assigned to a variable c_1 with type `T1`$\langle f_1, \dots \rangle$ if and only if `T2` and `T1` have the same type and all the formals match in name. In addition, if the indices are specified for the tags, then they must match.

3. For a type `T`$\langle o_1, o_2, \dots o_n \rangle$, we must have for all $i \in \{2, \dots n\}$, either $o_1.name \prec o_i.name$ or $o_1.name = o_i.name$ and $o_1.index < o_i.index$ (if both indices are known).[7]

4. The ownership tag `this` can only be used within the definition of a class that extends `Xmodule`.

5. Xmodule objects cannot have primitive-type fields.

The first three checks are analogous to the checks in BLS. The last two checks are added to enforce the additional requirements of Xmodules.

The OAT type system supports `where` clauses of the form `where` $(f_i < f_j)$. When f_i and f_j are instantiated with o_i and o_j, the type system ensures that either $o_i.name \prec o_j.name$ or $o_i.name = o_j.name$ and $o_i.index < o_j.index$. The detailed type rules for the OAT type system are described in [4].

The Book Application Using the OAT Type System

Figure 7-7 illustrates how one can specify Xmodules and ownership using ownership types. The programmer specifies an Xmodule by creating a class which extends from a special `Xmodule` class. The `DB` class has three formal owner tags: `dbO` which is the owner of the `DB` Xmodule instance, `logO` which is the owner of the `Logger` Xmodule instance that the `DB` Xmodule will use, and `dataO` which is the owner of the user data being stored in the database. When an instance of `UserApp` initializes Xmodules in lines 5–6, it declares itself as the owner of the `Logger`, the `DB`, and the user data being passed into `DB`. The indices on `this` are declaring the ordering of Xmodules in the module tree, i.e., the user data is lower-level than the `Logger`, and the `Logger` is lower level than the `DB`. Lines 11–13 illustrate how the `DB` class can initialize its Xmodules and propagate the formal owner tags (i.e., `logO` and `dataO`) down.

In order for this code to type check, the `DB` class must declare `logO < dataO` using the `where` clause in line 10. Otherwise the type check would fail at line 11 due to ambiguity of their relation in the module tree. The `where` clause in line 10 is checked whenever an instance of `DB` is created, e.g. at line 6.

[7] In the ownership tree, for any Xmodule M, the OAT type system implicitly assigns non-Xmodule children of M higher indices than the Xmodule children of M, unless the user specifies otherwise.

```
public class UserApp <appO> extends Xmodule {
    private Logger<this[1], this[2]> logger;
    private DB<this[0], this[1], this[2]> db;
     ...
    public UserApp () {
        logger = new Logger<this[1], this[2]>();
        db = new DB<this[0], this[1], this[2]>(logger);
    }
}

public class DB<dbO, logO, dataO>
        extends Xmodule where (logO < dataO) {
    private Logger<logO, dataO> logger;
    private BST <this[0], logO, dataO> bst;
    private HashTable <this[1], logO, dataO> hashtbl;
    public DB(Logger<logO, dataO> logger) {
        this.logger = logger;
         ...
    }
}
```

Figure 7-7: Example code that specifies Xmodules and ownership for the book application described in Section 7.1.

The OAT Type System's Guarantees

The following lemma about the OAT type system can be proved in a straightforward manner using Lemma 7.1.

Lemma 7.2. *The OAT type system guarantees the following properties.*

1. *An Xmodule M can access a (non-Xmodule) object b with ownership tag o_b only if $M \preceq o_b$.name.*
2. *An Xmodule M can call a method in another Xmodule N with owner o_N only if one of the following is true:*
   - *(a) $M = o_N$.name (i.e., M owns N);*
   - *(b) The least common ancestor of M and N in the module tree is o_N.name.*
   - *(c) $N \succeq M$ (i.e., N is an ancestor of M).*

Lemma 7.2 does not, however, guarantee all the properties we want from Xmodules (i.e., Definition 7.1). In particular, Lemma 7.2 does not consider any ordering of sibling Xmodules. The OAT type system can, however, provide stronger guarantees for a program which satisfies what we call the ***unique-owner-indices*** assumption: for all Xmodules M, all children of M in the module tree are instantiated with ownership tags with unique indices that can be statically determined. For such a program, the type system can order the children of every Xmodule M from smallest to largest index and assign the **`xid`** to each Xmodule as described in Section 7.1. Then the following result holds:

Theorem 7.3. *In the execution of a program with unique owner indices, consider any two Xmodules M and N. Let L be the least common ancestor Xmodule of M and N and let o_N be the ownership tag that N is instantiated with. If $L = o_N.name$, then M can call a method in N only if $\texttt{xid}(M) < \texttt{xid}(N)$.*

Proof. We prove by contradiction that if we have $L = o_N.name$ and $\texttt{xid}(M) > \texttt{xid}(N)$, then M cannot have a formal tag with value o_N. Therefore, M cannot declare a type with owner tag o_N and cannot access N.

Since $L = o_N.name$, we know that L is N's parent. Let Q be the ancestor of M which is N's sibling, and let o_Q be Q's ownership tag (i.e., the tag with which Q is instantiated). Since N and Q have the same parent (i.e. L) in the module tree, we have $o_N.name = o_Q.name = L$. Since $\texttt{xid}(M) > \texttt{xid}(N)$, M is to the right of N in the ownership tree. Therefore, Q, which is an ancestor of M, is to the right of N in the ownership tree. Therefore, we have $o_Q.index > o_N.index$, since the program has unique owner indices. If M has o_N as one of its tags, we have a contradiction, namely that $o_Q.index < o_N.index$.

Assume for contradiction that M does have o_N as one of its tags. Using Lemma 7.1, one can show that the only way for M to receive tag o_N is if Q also has a formal tag with value o_N. Thus, Q's first formal owner tag has value o_Q and another one of its formals has value o_N.

Let $P_0 = Q$, and consider the chain of Xmodule instantiations where Xmodule P_i instantiated P_{i-1}. Let $\langle f_a^i, f_b^i, \ldots \rangle$ be the formal ownership tags that P_i is instantiated with. When P_1 instantiates $Q = P_0$, we know f_a^1 has value o_Q and f_b^1 has value o_N. The first formal of P_0 must be f_a^1 since o_Q is the owner of Q, and without loss of generality, we assume that f_b^1 is the second formal since the type system does not care about the ordering of tags after the first one.

Since $o_N.name = o_Q.name = L$, this chain of instantiations must lead back to L, since L is the only Xmodule that can create ownership tags with values o_N and o_Q in its class definition (using the keyword `this`).[8] Let $P_k = L$. For the class declaration of each of the Xmodules P_i for $1 \leq i < k$, the following must hold:

- P_i must have formals f_a^i and f_b^i, with values o_Q and o_N, respectively, and P_i must pass these formals into the instantiation of P_{i-1}.
- In the type definition of P_i's class, P_i must have the constraint $f_a^i < f_b^i$ on its formal tags, either because f_a^i is the owner tag, or through a `where` clause that enforces $f_a^i < f_b^i$.

The first condition must hold for us to be able to pass both o_N and o_Q down to $P_0 = Q$. The second condition is true for the Xmodules by induction. In the base case, P_1 must know that $f_a^1 < f_b^1$, since otherwise, the type system will throw an error when it tries to instantiate $P_0 = Q$ with owner f_a^1. Then inductively, P_i must know $f_a^i < f_b^i$ to be able to instantiate P_{i-1}.

[8] Note that L could be the `world` Xmodule, in which case both o_N and o_Q were created in the `main` function using the `world` keyword.

Finally, P_{k-1} is instantiated in the class file corresponding to $P_k = L$. In this declaration, the formal f_a^k with value o_Q is instantiated with $\texttt{this}[x]$, where $x = o_Q.index$. Similarly, f_b^k with value o_N is instantiated with $\texttt{this}[y]$ with $y = o_N.index$. Since the class definition of P_k type checks, we have $f_a^k < f_b^k$, which implies $o_Q.index < o_N.index$ since the program has unique owner indices. But this fact contradicts the earlier conclusion that $o_Q.index > o_N.index$.

Thus, M cannot have an owner tag o_N, and therefore M cannot call a method in N. □

Lemma 7.2 along with Theorem 7.3 imposes restrictions on every Xmodule M which are only slightly weaker than the restrictions required by Definition 7.1. Condition 1 in Lemma 7.2 corresponds to Rule 1 of Definition 7.1. Conditions 2a and 2b are the cases permitted by Rule 2. Condition 2c, however, corresponds to the special case of callbacks or calling a method from the same Xmodule, which is not permitted by Definition 7.1. This case is modeled differently, as explained in Section 7.1.

The OAT type system is a best-effort type system that checks for the restrictions required by Definition 7.1. The OAT type system cannot fully guarantee, however, that a type-checked program does not violate Definition 7.1. Specifically, the OAT type system cannot always detect the following violations statically. First, if the program does not have unique owner indices, then L may instantiate both M and N with the same index. By Lemma 7.2, M can call methods from N and vice versa, creating cyclic dependencies between Xmodules.[9] Second, the program may perform improper callbacks. Say a method from M calls back to method B from L. An improper callback B can call a method of N, even though the type system knows that M is to the right of N. In both cases, the type system allows a program with cyclic dependency between Xmodules to pass the type checks, which is not allowed by Definition 7.1.

To have an ownership-aware TM which guarantees exactly Definition 7.1, one needs to impose additional dynamic checks. The runtime system can use the ownership tags to build a module tree during runtime and use it to perform dynamic checks to verify that every Xmodule has unique owner indices and contains only proper callbacks. The runtime system can do so by dynamically inferring indices according to which Xmodule calls which other Xmodule, reporting an error if there is any circular calling.[10]

7.3 The OAT Model

This section presents the OAT model, an abstract operational model for a TM runtime that utilizes Xmodules and ownership. The OAT model is derived from the transactional computation framework and the TCO operational model from Chapter 5. The OAT model makes three major changes to the TCO model: it incorporates the module tree into the framework, it defines special "module readsets" and "module writesets" to support an

[9] Since all non-Xmodule objects are implicitly assigned higher indices than their Xmodule siblings, these non-Xmodule objects cannot introduce cyclic dependencies between Xmodules.

[10] It is possible to statically check for unique owner indices by imposing additional restrictions on the program. OAT, however, is a more flexible programming model with weaker static guarantees.

ownership-aware commit of transactions, and it allows transactions to execute "abort actions" while a transaction is in the process of aborting. In this section, we use the transactional computation framework to formally describe each of these elements.

Xmodules and Computation Trees

We shall incorporate Xmodules into the transactional-computation framework described in Chapter 5. Formally, we consider traces $(\mathcal{C},\Phi)$ generated by a program that is organized into a set $\mathcal{N}$ of Xmodules. Each Xmodule $M \in \mathcal{N}$ has some number of methods and owns a set of memory locations.

Partition the set $\mathcal{M}$ of all memory locations into sets of memory owned by each Xmodule. Let $\texttt{modMemory}(M) \subseteq \mathcal{M}$ denote the set of memory locations owned by M. For a location $\ell \in \texttt{modMemory}(M)$, we say that $\texttt{owner}(\ell) = M$. When a method of Xmodule M is called by a method from a different Xmodule, a safe-nested transaction T is generated.[11] We use the notation $\texttt{xMod}(T) = M$ to associate the instance T with the Xmodule M. We also define the instances associated with M as

$$\texttt{modXactions}(M) = \{T \in \texttt{xactions}(\mathcal{C}) \ : \ \texttt{xMod}(T) = M\}.$$

As mentioned in Section 7.1, the Xmodules of a program are arranged as a module tree, denoted by $\mathcal{D}$. Each Xmodule is assigned an $\texttt{xid}$ according to a left-to-right depth-first tree walk, with the root of $\mathcal{D}$ being $\texttt{world}$ with $\texttt{xid} = 0$. Denote the parent of Xmodule M in $\mathcal{D}$ as $\texttt{modParent}(M)$ and the ancestors of M as $\texttt{modAnces}(M)$ (including M itself). Similarly, let $\texttt{modDesc}(M)$ be the set of M's descendants. We say that $\texttt{xMod}(\texttt{root}(\mathcal{C})) = \texttt{world}$, i.e., the root of the computation tree is a transaction associated with the $\texttt{world}$ Xmodule.

We can use the module tree $\mathcal{D}$ to restrict the sharing of data between Xmodules and to limit the visibility of Xmodule methods according to the rules given in the following definition.

Definition 7.2 (Formal Restatement of Definition 7.1). *A program with a module tree $\mathcal{D}$ generates traces $(\mathcal{C},\Phi)$ which satisfy the following rules:*

1. *For any memory operation v which accesses a location ℓ, let $T = \texttt{xParent}(v)$. Then $\texttt{owner}(\ell) \in \texttt{modAnces}(\texttt{xMod}(T))$.*

2. *Let $X,Y \in \texttt{xactions}(\mathcal{C})$ be transaction instances such that $\texttt{xMod}(X) = M$ and $\texttt{xMod}(Y) = N$. Then $X = \texttt{xParent}(Y)$ only if $\texttt{modParent}(N) \in \texttt{modAnces}(M)$, and $\texttt{xid}(M) < \texttt{xid}(N)$.*

By Rule 1, a method from Xmodule M can only directly access memory that it owns or that an ancestor Xmodule N owns. By Rule 2, a method from M can call a method from N only if N is the child of some ancestor of M and if N is "to the right" of M in the module tree. Thus, an Xmodule can only call methods of some (but not all) lower-level Xmodules.

[11] As explained in Section 7.1, callbacks are handled differently.

The OAT Model

The OAT model, which is derived from the TCO model presented in Section 5.5, models the behavior of a TM runtime as it executes a program with transactions. Conceptually, the OAT model operates in the same fashion as the TCO model: OAT maintains a set of ready nodes, and on each time step, the model nondeterministically chooses a ready node to execute an instruction. The OAT model issues the same set of instructions as the TCO model and dynamically builds a computation tree $\mathcal{C}$ in the same way. The primary difference between the OAT and TCO models is that the OAT model uses Xmodules to make some minor modifications to the transaction control instructions (xbegin, xend, xabort, sigabort), and to memory operations (read and write instructions).

In addition to basic readsets and writesets, the OAT model also defines a ***module readset*** and ***module writeset*** for all transactions $X \in \texttt{xactions}^{(t)}(\mathcal{C}) \cap \texttt{vTree}^{(t)}(\mathcal{C})$. The module readset is defined as

$$\texttt{modR}^{(t)}(X) = \left\{(\ell, v) \in \texttt{R}^{(t)}(X) \ : \ \texttt{owner}(\ell) = \texttt{xMod}(X)\right\}.$$

In other words, $\texttt{modR}^{(t)}(X)$ is the subset of $\texttt{R}^{(t)}(X)$ that accesses memory owned by X's Xmodule $\texttt{xMod}(X)$. Similarly, define the module writeset as

$$\texttt{modW}^{(t)}(X) = \left\{(\ell, v) \in \texttt{W}^{(t)}(X) \ : \ \texttt{owner}(\ell) = \texttt{xMod}(X)\right\}.$$

The OAT model maintains the same invariants on readsets and writesets (both basic and module) as the TCO model. First, for every transaction $X \in \texttt{xactions}^{(t)}(\mathcal{C})$, $\texttt{W}^{(t)}(X) \subseteq \texttt{R}^{(t)}(X)$, i.e., a write also counts as a read. Second, $\texttt{R}^{(t)}(X)$, and $\texttt{W}^{(t)}(X)$ each contain at most one pair (ℓ, v) for any location ℓ. Thus, we use the shorthand $\ell \in \texttt{R}^{(t)}(X)$ to mean that there exists a node u such that $(\ell, u) \in \texttt{R}^{(t)}(X)$, and similarly for the other readsets and writesets. We also overload the union operator: at some step t, an operation $\texttt{R}(X) = \texttt{R}(X) \cup \{(\ell, u)\}$ means we construct $\texttt{R}^{(t+1)}(X)$ by

$$\texttt{R}^{(t+1)}(X) = \{(\ell, u)\} \cup \left(\texttt{R}^{(t)}(X) - \left\{(\ell, u') \in \texttt{R}^{(t)}(X)\right\}\right).$$

In other words, we add (ℓ, u) to $\texttt{R}(X)$, replacing any $(\ell, u') \in \texttt{R}^{(t)}(X)$ that existed previously.

Ownership-Aware Transaction Commit

The OAT model employs the ***ownership-aware commit mechanism***, which contains elements of both closed-nested and open-nested commits. A PENDING transaction Y issues an xend instruction to commit Y into $X = \texttt{xParent}(Y)$. This xend commits locations from its readset and writeset that are owned by $\texttt{xMod}(Y)$ in an open-nested fashion to the root of the tree, whereas it commits locations owned by other Xmodules in a closed-nested fashion, merging those reads and writes into X's readset and writeset.

The OAT model's commit mechanism can be described more formally in terms of module readsets and writesets. Suppose that at time t, a transaction $Y \in \texttt{xactions}^{(t)}(\mathcal{C})$ with $\texttt{status}(Y) = \texttt{PENDING}$ issues an xend. This xend instruction changes readsets and writesets as follows:

$$\begin{aligned}
\mathtt{R}(\mathtt{root}(C)) &= \mathtt{R}(\mathtt{root}(C)) \cup \mathtt{modR}(Y)\,, \\
\mathtt{R}(\mathtt{xParent}(Y)) &= \mathtt{R}(\mathtt{xParent}(Y)) \cup (\mathtt{R}(Y) - \mathtt{modR}(Y))\,, \\
\mathtt{W}(\mathtt{root}(C)) &= \mathtt{W}(\mathtt{root}(C)) \cup \mathtt{modW}(Y)\,, \\
\mathtt{W}(\mathtt{xParent}(Y)) &= \mathtt{W}(\mathtt{xParent}(Y)) \cup (\mathtt{W}(Y) - \mathtt{modW}(Y))\,.
\end{aligned}$$

Definition 7.2 guarantees certain properties of the computation tree which are essential to the ownership-aware commit mechanism, namely the ***unique committer property***. Theorem 7.5 proves this property, namely that every memory operation has one and only one transaction that is responsible for committing the memory operation. Before proving this result however, we require the following lemma.

Lemma 7.4. *Given a computation tree C, for any $T \in \mathtt{xactions}(C)$, we have*

$$\mathtt{modAnces}(\mathtt{xMod}(T)) \subseteq \{\mathtt{xMod}(T') \,:\, T' \in \mathtt{xAnces}(T)\}\;.$$

Proof. We prove this fact by induction on the nesting depth of transactions T in the computation tree. In the base case, the top-level transaction $T = \mathtt{root}(C)$, and $\mathtt{xMod}(\mathtt{root}(C)) = \mathtt{world}$. Thus, the fact holds trivially.

For the inductive step, define σ_T as the set $\{\mathtt{xMod}(T') \,:\, T' \in \mathtt{xAnces}(T)\}$ and assume that $\mathtt{modAnces}(\mathtt{xMod}(T)) \subseteq \sigma_T$ holds for any transaction T at depth d. We show that this inductive hypothesis holds for any $T^* \in \mathtt{xactions}(C)$ at depth $d+1$.

For any such T^*, we know $T = \mathtt{xParent}(T^*)$ is at depth d. By Rule 2 of Definition 7.2, we have $\mathtt{modParent}(\mathtt{xMod}(T^*)) \in \mathtt{modAnces}(\mathtt{xMod}(T))$. Thus, we know that $\mathtt{modAnces}(\mathtt{xMod}(T^*)) \subseteq \mathtt{modAnces}(\mathtt{xMod}(T)) \cup \{\mathtt{xMod}(T^*)\}$. By construction of the set σ_T, we have $\sigma_{T^*} = \sigma_T \cup \{\mathtt{xMod}(T^*)\}$. Therefore, using the inductive hypothesis, we have $\mathtt{modAnces}(\mathtt{xMod}(T^*)) \subseteq \sigma_{T^*}$. □

Lemma 7.4 allows us to prove the unique-committer property.

Theorem 7.5. *If a memory operation v accesses a memory location ℓ, then there exists a unique transaction $T^* \in \mathtt{xAnces}(v)$ such that*

1. *$\mathtt{owner}(\ell) = \mathtt{xMod}(T^*)$, and*
2. *for all transactions $X \in \mathtt{pAnces}(T^*) \cap \mathtt{xactions}(C)$, X cannot directly access location ℓ.*

*This transaction T^* is the **committer** of memory operation v, denoted $\mathtt{committer}(v)$.*

Proof. This result follows from the properties of the module tree and computation tree stated in Definition 7.2. Let $T = \mathtt{xParent}(v)$. From Rule 1 of Definition 7.2, we have that $\mathtt{owner}(\ell) \in \mathtt{modAnces}(\mathtt{xMod}(T))$. We know by Lemma 7.4 that $\mathtt{modAnces}(\mathtt{xMod}(T)) \subseteq \{\mathtt{xMod}(T') \,:\, T' \in \mathtt{xAnces}(T)\}$. Thus, there exists some transaction $T^* \in \mathtt{xAnces}(T)$ such that $\mathtt{owner}(\ell) = \mathtt{xMod}(T^*)$. We can use Rule 2 to show that T^* is unique. Let X_i be the chain of ancestor transactions of T, i.e., let $X_0 = T$, and let $X_i = \mathtt{xParent}(X_{i-1})$ up until $X_k = \texttt{root}(C)$. By Rule 2, we know $\texttt{xid}(\texttt{xMod}(X_i)) < \texttt{xid}(\texttt{xMod}(X_{i-1}))$, that is, the xids strictly decrease walking up the tree from T. Thus, there can only be one ancestor transaction T^* of T with $\texttt{xid}(\texttt{xMod}(T^*)) = \texttt{xid}(\texttt{owner}(\ell))$.

To check the second condition, consider any $X \in \texttt{pAnces}(T^*) \cap \texttt{xactions}(C)$. By Rule 1, X can access ℓ directly only if $\texttt{owner}(\ell) \in \texttt{modAnces}(\texttt{xMod}(X))$, or equivalently, only if $\texttt{xid}(\texttt{owner}(\ell)) \leq \texttt{xid}(\texttt{xMod}(X))$. But we know that $\texttt{owner}(\ell) = \texttt{xMod}(T^*)$ and $\texttt{xid}(\texttt{xMod}(T^*)) > \texttt{xid}(\texttt{xMod}(X))$. Thus, X cannot directly access ℓ. □

Intuitively, $T^* = \texttt{committer}(v)$ is the transaction which "belongs" to the same Xmodule as the location ℓ which v accesses, and is "responsible" for committing v to memory and making it visible to the world. The second condition of Theorem 7.5 states that no ancestor transaction of T^* in the call stack can ever directly access ℓ, and hence it is "safe" for T^* to commit ℓ.

Transaction Conflicts and Aborts

The OAT model performs eager conflict detection in the same way as the TCO model. The only difference in the OAT model is that a transaction with a status of PENDING_ABORT is still allowed to continue issuing instructions because it may need to compensate for the commit of its open-nested transactions.

More formally, the OAT model uses same definition of conflict (Definition 5.17) for determining when transactions should abort. As in the TCO model, if a read or write generates a potential memory operation v that would cause a conflict according to Definition 5.17, then the runtime does not perform v but instead triggers the sigabort of a transaction.[12] Otherwise, a successful memory operation adds v to the appropriate readset and/or writeset.

Because a PENDING_ABORT transaction can still issue instructions, it is useful to define its "abort actions."

Definition 7.3. *Define* $\texttt{abortactions}(X)$ *as the set of operations issued by a transaction* X *or descendants of* X *after* $\texttt{status}(X)$ *changes to PENDING_ABORT. We call* $\texttt{abortactions}(X)$ *the* ***abort actions*** *of* X.

The PENDING_ABORT status allows X to compensate for the safe-nested transactions that may have committed. If transaction Y is nested inside X, then the abort actions of X contain the compensating action of Y. Eventually a PENDING_ABORT transaction issues an xend instruction, which changes its status from PENDING_ABORT to ABORTED.

If a potential memory operation v generates a conflict with a transaction X, and the status of X is PENDING, then the OAT model can nondeterministically choose to abort either $\texttt{xParent}(v)$, or X. In the latter case, v waits for X to finish aborting (i.e., change its status to ABORTED) before continuing. If X's status is PENDING_ABORT, then v just waits for X to finish aborting before trying to issue read or write again. As described later in Section 7.5, some restrictions on the abort actions of a transaction may be necessary to avoid deadlock.

[12] The readsets and writesets are defined for transactions X even if the status of X is PENDING_ABORT. Thus, the sets $\texttt{readers}^{(t)}(\ell)$ and $\texttt{writers}^{(t)}(\ell)$ in Definition 5.17 automatically include these transactions X that might be in the process of aborting.

7.4 Serializability by Modules

This section defines ***serializability by modules***, a definition inspired by the database notion of multilevel serializability (e.g., as described in [124]). First, we consider the definition of serializability in transactional computations from Section 5.3 to incorporate Xmodules and the ownership-aware commit mechanism. Because this definition is too restrictive to allow the kind of program interleavings that we want with open-nested transactions, we then investigate a less restrictive correctness condition, called "serializability by modules." Next, we prove that the OAT model guarantees this serializability by modules. Finally, we explore the relationship between the definition of serializability by modules and the notion of abstract serializability for the methodology of open nesting.

Xmodules and Content Sets

To generalize the definition of serializability from Section 5.3 (Definition 5.11), we need to modify the notion of the "content" of a transaction X (Definition 5.3) and the hidden relation (Definition 5.5) to account for Xmodules and the ownership-aware commit mechanism.

We define "content" sets for every transaction X by partitioning $\mathtt{memOps}(X)$ — all the memory operations enclosed inside X including those belonging to its nested transactions — into three sets: $\mathtt{cContent}(X)$, $\mathtt{oContent}(X)$ and $\mathtt{aContent}(X)$. For any $u \in \mathtt{memOps}(X)$, the content set that contains u is defined based on the final status of transactions in C that one visits when walking up the tree from u to X.

Definition 7.4. *For any transaction X and memory operation $u \in \mathtt{memOps}(C)$, define the sets $\mathtt{cContent}(X)$, $\mathtt{oContent}(X)$, and $\mathtt{aContent}(X)$ according to the following $\mathtt{ContentType}(u,X)$ procedure:*

```
ContentType$(u,X)$        // For any $u \in \mathtt{memOps}(X)$
$Z \leftarrow \mathtt{xParent}(u)$
while $(Z \neq X)$
    if ($Z$ is ABORTED) return $u \in \mathtt{aContent}(X)$
    if $(Z = \mathtt{committer}(u))$ return $u \in \mathtt{oContent}(X)$
    $Z \leftarrow \mathtt{xParent}(Z)$
return $u \in \mathtt{cContent}(X)$
```

Recall that in the OAT model, the safe-nested commit of X commits some memory operations in an open-nested fashion to $\mathtt{root}(C)$ and some operations in a closed-nested fashion to $\mathtt{xParent}(X)$. Intuitively, Definition 7.4 generalizes Definition 5.3 by adding the set $\mathtt{oContent}(X)$ — the set of memory operations enclosed by transaction X which are committed by open-nested transactions inside X.[13] Before, in Section 5.2, when all transactions were closed-nested, we had $\mathtt{oContent}(X) = \emptyset$ for all transactions X.

[13] Definition 7.4 considers only an a posteriori analysis of a computation, and does not include the set $\mathtt{vContent}(X)$ from Definition 5.18. In fact, one can generalize Definition 7.4 to model dynamic executions, as discussed in Section C.3. Definition 7.4 is also similar to Definition 5.6, but Definition 7.4 is intended for TM with an ownership-aware commit instead of an open-nested commit mechanism.

Using this definition of content sets in Definition 7.4, we can modify the hidden relation from Definition 5.5.

Definition 7.5. *Consider a computation C executed by an ownership-aware TM system. For $u \in \texttt{memOps}(C)$ and $v \in V(C)$, let $X = \texttt{xLCA}(u, v)$. We say that u is **hidden** from v, denoted uHv, if $u \in \texttt{aContent}(X)$.*

By Definition 7.5, a memory operation $u \in V(X)$ may not be hidden from a memory operation v running in parallel with X if $u \in \texttt{cContent}(Z)$ for some transaction Z which is an open-nested transaction. For example, consider the case where Z is open-nested inside Y and Y is closed-nested inside X. If Z commits and $Z = \texttt{committer}(u)$, then $u \in \texttt{cContent}(Z)$ can be visible to a v outside X even if Y aborts, since $u \in \texttt{oContent}(X)$.

With the new hidden relation from Definition 7.5, we can keep the same definition of transactional last writer from Definition 5.9. The transactional last writer in turn allows us to define a memory model of serializability, as in Definition 5.11.

Unfortunately, serializability from Definition 5.11 is too restrictive for executions that we want to permit using open-nested transactions. For example, in Figure 7-2, suppose that the outer transactions X and Y belong to an Xmodule M, but the `Insert2` methods belong to a lower-level Xmodule N. From the perspective of M, we would like to permit a schedule which executes the fragments in the order

$$X_1, A_1, X_2, Y_1, B_1, Y_2, A_2, B_2, X_3, Y_3 ,$$

because M should not care about the memory locations accessed inside the open-nested transactions A_1, B_1, A_2 and B_2. Serializability, according to Definition 5.11, would enforce the dependencies from the chain of writes to the global counter inside the `Insert2` method, i.e., B_2 depends on A_2, which depends on B_1, which depends on A_1. Thus, by Definition 5.11, this schedule is not serializable, since there is no way to reorder the execution of X completely before Y, or vice versa.

More generally, redefining the hidden relation also automatically generalizes the memory models of serializability, race freedom, and prefix-race freedom defined in Section 5.3. It turns out that the OAT model guarantees that executions are prefix-race-free according to this generalized definition. One can easily show, however, that with open nesting (using either an open-nested commit mechanism or an ownership-aware commit mechanism), prefix-race freedom is no longer equivalent to serializability. For example, see [8] for a simple execution trace that demonstrates the distinctness of the models. Thus, one would like to define a correctness condition weaker than serializability that is satisfied by prefix-race-free executions.

Defining Serializability by Modules

To allow a TM system to ignore memory accesses from transactions performed by lower-level Xmodules when detecting conflicts in higher-level Xmodules, we can define "serializability by modules," a memory model which checks for serializability one Xmodule at a time.

For this new definition, given a trace $(\mathcal{C},\Phi)$, for each Xmodule M, transform the tree $\mathcal{C}$ into a new tree $\texttt{mTree}(\mathcal{C},M)$, called the ***projection of*** $\mathcal{C}$ ***for Xmodule*** M. This projected tree $\texttt{mTree}(\mathcal{C},M)$ is constructed in such a way as to ignore memory operations of Xmodules which are lower-level than M and also all operations which are hidden from transactions of M. For each Xmodule M, this definition checks that the transactions of M in the trace $(\texttt{mTree}(\mathcal{C},M),\Phi)$ are serializable. If the check holds for all Xmodules, then trace $(\mathcal{C},\Phi)$ is said to be serializable by modules.

Definition 7.6 formalizes the construction of $\texttt{mTree}(\mathcal{C},M)$.

Definition 7.6. *For any computation tree* $\mathcal{C}$*, define the* ***projection of*** $\mathcal{C}$ ***for*** M*, denoted* $\texttt{mTree}(\mathcal{C},M)$*, as the result of modifying* $\mathcal{C}$ *according to the following steps:*

1. *For all memory operations* $u \in \texttt{memOps}(\mathcal{C})$ *with* u *accessing* ℓ*, if* $\texttt{owner}(\ell) = N$ *for some* $\texttt{xid}(N) > \texttt{xid}(M)$*, convert* u *into a nop.*
2. *For all transactions* $X \in \texttt{modXactions}(M)$*, convert all* $u \in \texttt{aContent}(X)$ *into nops.*

The intuition behind Definition 7.6 is the following. When looking at Xmodule M, in Step 1 we throw away memory operations belonging to a lower-level Xmodule N, since by Theorem 7.5, transactions of M can never directly access the same memory as those operations anyway. In Step 2, we ignore the content of any aborted transactions nested inside transactions of M. Those transactions might access the same memory locations as operations which we did not turn into nops, but those operations are aborted with respect to transactions of M.

The next lemma argues that if a trace $(\mathcal{C},\Phi)$ is sequentially consistent (i.e., it satisfies Definition 5.10 using the hidden relation in Definition 7.5), then $(\texttt{mTree}(\mathcal{C},M),\Phi)$ is a valid trace. More precisely, any operation u that remains in the trace never attempts to observe a value from a $\Phi(u)$ which was turned into a nop due to Definition 7.6. In addition, the transformed trace is also sequentially consistent.

Lemma 7.6. *Let* $(\mathcal{C},\Phi)$ *be any trace and let* $\mathcal{S}$ *be any topological sort of* $\mathcal{G}(\mathcal{C})$ *such that* $\Phi = \mathcal{X}_{\mathcal{S}}$*. Then for any Xmodule* M*, we have the following:*

1. *If* $u \in \texttt{memOps}(\texttt{mTree}(\mathcal{C},M))$*, then* $\Phi(u) \in \texttt{memOps}(\texttt{mTree}(\mathcal{C},M))$*, and*
2. $\mathcal{S}$ *is a valid topological sort of* $\mathcal{G}(\texttt{mTree}(\mathcal{C},M))$*, with* $\Phi = \mathcal{X}_{\mathcal{S}}$*.*

In other words, $(\texttt{mTree}(\mathcal{C},M),\Phi)$ *is a valid trace.*

Proof. First, we verify Condition 1. In $\texttt{mTree}(\mathcal{C},M)$, pick any $u \in \texttt{memOps}(\texttt{mTree}(\mathcal{C},M))$ which remains. Assume for contradiction that $v = \Phi(u)$ was turned into a nop in one of Steps 1 and 2.

If v was turned into a nop in Step 1 of Definition 7.6, then we know that v accessed a memory location ℓ with $\texttt{xid}(\texttt{owner}(\ell)) > \texttt{xid}(M)$. Since u must access the same location ℓ, u must also be converted into a nop.

If v was turned into a nop in Step 2 of Definition 7.6, then $v \in \texttt{aContent}(T)$ for some $\texttt{xMod}(T) = M$. Then we can show that either vHu or u should have also been turned into a nop. Let $X = \texttt{xLCA}(v,u)$. Since X and T are both ancestors of v, either T is a proper ancestor of X, or X is an ancestor of T.

1. First, suppose that T is a proper ancestor of X. Let $Y_0, Y_1, \ldots Y_k$ be the path of transactions from v to T, i.e., $Y_0 = \texttt{xParent}(v)$, $\texttt{xParent}(Y_i) = Y_{i+1}$, and $\texttt{xParent}(Y_k) = T$. Since $v \in \texttt{aContent}(T)$, for some Y_j for $0 \le j \le k$ must have $\texttt{status}(Y_j) = \texttt{ABORTED}$. Since T is a proper ancestor of X, we have $X = Y_x$ for some x satisfying $0 \le x \le k$.
   (a) If $\texttt{status}(Y_j) = \texttt{ABORTED}$ for any j satisfying $0 \le j < x$, then we know that $v \in \texttt{aContent}(X)$, and thus vHu. Since we assumed that $(\mathcal{C}, \Phi)$ is sequentially consistent and $\Phi(v) = u$, by Definition 5.9 we know $\neg(vHu)$, leading to a contradiction.

   (b) If Y_j is $\texttt{ABORTED}$ for any j satisfying $x \le j \le k$, then $\texttt{status}(Y_j) = \texttt{ABORTED}$ implies that $u \in \texttt{aContent}(T)$, and thus, u should have been turned into a nop, contradicting the original setup of the statement.

2. Next, consider the case where X is an ancestor of T. Since $v \in \texttt{aContent}(T)$, we have $v \in \texttt{aContent}(X)$. Therefore, this case is analogous to Case 1a above.

To check Condition 2 of the lemma, since the construction in Definition 7.6 does not remove any nodes from $\mathcal{C}$, $\mathcal{G}(\texttt{mTree}(\mathcal{C}, M))$ still has the same nodes and edges as $\mathcal{G}(\mathcal{C})$, and thus $\mathcal{S}$ is still a valid topological sort. Since Φ is the transactional last writer according to $\mathcal{S}$ for $(\mathcal{C}, \Phi)$, it is still the transactional last writer for $(\texttt{mTree}(\mathcal{C}, M), \Phi)$, because the memory operations which are not turned into nops remain in the same relative order. Thus, Condition 2 is satisfied. □

Lemma 7.6 *depends on* the restrictions on Xmodules described in Definition 7.2. Without this structure of modules and ownership, the construction of Definition 7.6 is not guaranteed to generate a valid trace.

Also, the set of memory operations turned into nops for $\texttt{mTree}(\mathcal{C}, M)$ strictly increases as we decrease $\texttt{xid}(M)$. If L is the lowest-level Xmodule, then Definition 7.6 keeps all memory operations for L, i.e., $\texttt{mTree}(\mathcal{C}, L) = \mathcal{C}$. Once a memory operation u is turned into a nop for Xmodule M, it is turned into a nop for all Xmodules N with $\texttt{xid}(N) < \texttt{xid}(M)$.

Finally, we can define serializability by modules.

Definition 7.7. *A trace $(\mathcal{C}, \Phi)$ is **serializable by modules** if*

1. *There exists a topological sort $\mathcal{S}$ of $\mathcal{G}(\mathcal{C})$ such that $\Phi = \mathcal{X}_{\mathcal{S}}$, and*
2. *For all Xmodules M in $\mathcal{D}$, there exists a topological sort $\mathcal{S}_M$ of $\mathcal{C}_M = \texttt{mTree}(\mathcal{C}, M)$ such that:*
   (a) $\mathcal{S}$ is a topological sort of $\mathcal{G}(\mathcal{C}_M)$ and $\Phi = \mathcal{X}_{\mathcal{S}_M}$,

   (b) For all transactions $T \in \texttt{modXactions}(M)$ and for all $v \in V(\mathcal{C}_M)$, if we have $\texttt{source}(T) \le_{\mathcal{S}_M} v \le_{\mathcal{S}_M} \texttt{sink}(T)$, then $v \in V(T)$.

Informally, Condition 1 of Definition 7.7 requires that a trace $(\mathcal{C}, \Phi)$ be sequentially consistent (as in Definition 5.10). Condition 2 requires that $\texttt{mTree}(\mathcal{C}, M)$ is serializable (as in Definition 5.11) for every module M when we consider only transactions of M, i.e., that there exists a sequentially-consistent order $\mathcal{S}_M$ that has all transactions of M appearing contiguous.

OAT Model Guarantees Serializability by Modules

We can show that the OAT model described in Section 7.3 generates traces $(\mathcal{C},\Phi)$ that are serializable by modules, i.e., that satisfy Definition 7.7. The proof of this fact consists of three steps. First, we show that the OAT model generates traces which are prefix-race-free using a generalized version of Definition 5.15. Then, we show that any trace which is free from prefix races is also serializable by modules.

To define prefix-race freedom, we utilize the same definition of prefix races from Definition 5.14, except using the hidden relation from Definition 7.5. The following theorem states that the OAT model generates prefix-race-free traces.

Theorem 7.7. *Suppose that the OAT model generates a trace $(\mathcal{C},\Phi)$ with an execution order $\mathcal{S}$. Then $\mathcal{S}$ is a prefix-race-free sort of $(\mathcal{C},\Phi)$.*

Proof. This proof, which is an induction on the steps of the OAT model, is essentially the same as the proof of Theorem 5.7. The technical details of both proofs are presented in Appendix C. □

The next theorem shows that a trace $(\mathcal{C},\Phi)$ which is prefix-race free is also serializable by modules.

Theorem 7.8. *Any trace $(\mathcal{C},\Phi)$ which is prefix-race free is also serializable by modules.*

Proof. First, Definition 7.6 and Lemma 7.6 imply that any prefix-race-free sort $\mathcal{S}$ of a trace $(\mathcal{C},\Phi)$ is also prefix-race-free sort of the trace $(\texttt{mTree}(\mathcal{C},M),\Phi)$ for any Xmodule M, since converting operations into nops does not introduce any prefix races. We shall argue that for any Xmodule M, we can transform $\mathcal{S}$ into $\mathcal{S}_M$ such that all transactions in $\texttt{xactions}(M)$ appear contiguous in $\mathcal{S}_M$.

Consider a prefix-race-free sort $\mathcal{S}$ of $(\texttt{mTree}(\mathcal{C},M),\Phi)$ which has $k > 0$ nodes that violate the second condition of Definition 7.7. We can construct a new order $\mathcal{S}'$ which is still a prefix-race-free sort of $(\texttt{mTree}(\mathcal{C},M),\Phi)$, but which has only $k-1$ violations.

We reduce the number of violations according to the following procedure:

1. For a computation $\mathcal{C}$ and sort $\mathcal{S}$, define the set of transactions $\mathcal{Q}(\mathcal{C},\mathcal{S})$ as

   $$\{Y \in \texttt{xactions}(\mathcal{C}) \ : \ \exists v \in V(\mathcal{C}) - V(Y) \text{ such that } \texttt{source}(Y) \leq_{\mathcal{S}} v \leq_{\mathcal{S}} \texttt{sink}(Y)\}.$$

   For the prefix-race-free sort $\mathcal{S}$ of $(\texttt{mTree}(\mathcal{C},M),\Phi)$, choose the $T \in \mathcal{Q}(\mathcal{C},\mathcal{S})$ such that for all $Y \in \mathcal{Q}(\mathcal{C},\mathcal{S})$, $\texttt{sink}(Y) \leq_{\mathcal{S}} \texttt{sink}(T)$. Intuitively, T is the transaction in $\mathcal{Q}(\mathcal{C},\mathcal{S})$ with the sink node that occurs latest in the order $\mathcal{S}$.

2. In T, pick the first $v \in V(\mathcal{C}) - V(T)$ which causes a violation, i.e., for all $v' \in V(\mathcal{C}) - V(T)$ such that $\texttt{source}(T) <_{\mathcal{S}} v' <_{\mathcal{S}} \texttt{sink}(T)$, we have $v \leq_{\mathcal{S}} v'$.

3. Create a new sort $\mathcal{S}'$ by moving v to be immediately before $\texttt{source}(T)$.

In order to argue that $\mathcal{S}'$ is still a prefix-race-free sort of $(\texttt{mTree}(\mathcal{C},M),\Phi)$, we need to show that moving v does not generate any new prefix races or create a sort $\mathcal{S}'$ which is no longer sequentially consistent with respect to Φ (i.e., that Φ is still the transactional last writer according to $\mathcal{S}'$). There are three cases: v can be a memory operation, $\texttt{source}(T')$ for some transaction T', or $\texttt{sink}(T')$ for some transaction T'.

1. Suppose that v is a memory operation which accesses location ℓ. For all operations w such that $\texttt{source}(T) <_{\mathcal{S}} w <_{\mathcal{S}} v$, we argue that w cannot access the same location ℓ unless both w and v read from ℓ. Since we chose v to be the first memory operation with $\texttt{source}(T) <_{\mathcal{S}} v <_{\mathcal{S}} \texttt{sink}(T)$ and $v \notin V(T)$, we know that $w \in V(T)$. We know by construction of $\texttt{mTree}(\mathcal{C},M)$ that $w \in \texttt{cContent}(T)$, since if we had $w \in \texttt{oContent}(T)$ or $w \in \texttt{aContent}(T)$, then steps 1 or 2, respectively, in Definition 7.6 would turn w into a nop. Therefore, by Definition 5.14, unless w and v both read from ℓ, v has a prefix race with T, contradicting the fact that $\mathcal{S}$ is a prefix-race-free sort of the trace. Thus, moving v to be before $\texttt{source}(T)$ cannot generate any new prefix races.

   Also, after moving v, we cannot have $v <_{\mathcal{S}'} \Phi(v)$. This situation would require $\texttt{source}(T) <_{\mathcal{S}} \Phi(v)$, which is impossible because $\Phi(v)$ writes to ℓ, and we would have a memory operation $w = \Phi(v)$ with $\texttt{source}(T) <_{\mathcal{S}} v <_{\mathcal{S}} v$.

   Thus, $\mathcal{S}'$ is a prefix-race-free sort of the trace $(\texttt{mTree}(\mathcal{C},M),\Phi)$.

2. Next, suppose that $v = \texttt{source}(T')$. Moving $\texttt{source}(T')$ cannot generate any new prefix races with T', because the only operations u which satisfy $\texttt{source}(T) <_{\mathcal{S}} u <_{\mathcal{S}} \texttt{source}(T')$ also satisfy $u \in V(T)$ and $V(T) \cap \texttt{cContent}(T') = \emptyset$. Also, moving $\texttt{source}(T')$ does not change the transactional last writer for any node v because the move preserves the relative order of all memory operations. Therefore, $\mathcal{S}'$ is still a prefix-race-free sort.

3. Finally, suppose that $v = \texttt{sink}(T')$. By moving $\texttt{sink}(T')$ to be before $\texttt{source}(T)$, we can only lose prefix races with T' that already existed in $\mathcal{S}$ because we are moving nodes out of the interval $[\texttt{source}(T'), \texttt{sink}(T')]$. As with $\texttt{source}(T')$, moving $\texttt{sink}(T')$ does not change any transaction last writers. Therefore, $\mathcal{S}'$ is still a prefix-race-free sort of the trace.

Since we can eliminate violations of the second condition of Definition 7.7 one at a time, we can construct a sort $\mathcal{S}_M$ which satisfies serializability by modules by eliminating all violations. □

Finally, the following theorem combines the previous results to prove that the OAT model guarantees serializability by modules.

Theorem 7.9. *Any trace $(\mathcal{C},\Phi)$ generated by the OAT model is serializable by modules.*

Proof. By Theorem 7.7, the OAT model generates only trace $(\mathcal{C},\Phi)$ which are prefix-race free. By Theorem 7.8, any trace $(\mathcal{C},\Phi)$ which is prefix-race free is serializable by modules. □

Abstract Serializability

By Theorem 7.9, the OAT model guarantees serializability by modules. This definition is related to the notion of "abstract serializability" used in multilevel database systems [124]. As discussed at the beginning of this chapter, the ownership-aware commit mechanism is a part of a methodology which includes abstract locks and compensating actions. This section argues that OAT model provides enough flexibility to accommodate abstract locks and compensating actions. In addition, if a program is "properly locked and compensated," then serializability by modules guarantees abstract serializability.

The definition of abstract serializability in [124] assumes that a program is divided into levels, where a transaction at level i can only call a transaction at level $i+1$.[14] In addition, transactions at a particular level have predefined commutativity rules, i.e., some transactions of the same Xmodule can commute with each other and some cannot. The transactions at the lowest level (say k) are naturally serializable. Call this schedule $\mathcal{Z}_k$. Given a serializable schedule $\mathcal{Z}_{i+1}$ of level-$(i+1)$ transactions, the schedule is said to be ***serializable at level*** i if all transactions in $\mathcal{Z}_{i+1}$ can be reordered, obeying all commutativity rules, to obtain a serializable order $\mathcal{Z}_i$ for level-i transactions. The original schedule is said to be ***abstractly serializable*** if it is serializable for all levels.

These commutativity rules might be specified using abstract locks [105]: if two transactions cannot commute, then they grab the same abstract lock in a conflicting manner. In the book application described in Section 7.1, for instance, transactions calling `insert` and `remove` on the `BST` using the same key do not commute and should grab the same write lock. Although abstract locks are not explicitly modeled in the OAT model, we can model transactions acquiring the same abstract lock as transactions writing to a common memory location ℓ.[15] Locks associated with an Xmodule M are owned by $\texttt{modParent}(M)$. A module M is said to be ***properly locked*** if the following condition holds for all transactions T_1, T_2 with $\texttt{xMod}(T_1) = \texttt{xMod}(T_2) = M$: if T_1 and T_2 do not commute, then they access some memory location $\ell \in \texttt{modMemory}(\texttt{modParent}(M))$ in a conflicting manner.

If all transactions are properly locked, then serializability by modules implies abstract serializability (as defined above) in the special case when the module tree is a chain (i.e., each nonleaf module has exactly one child). Let $\mathcal{S}_i$ be the sort $\mathcal{S}$ in Definition 7.7 for Xmodule M with $\texttt{xid}(M) = i$. This $\mathcal{S}_i$ corresponds to $\mathcal{Z}_i$ in the definition of abstract serializability.

In the general case for ownership-aware TM, however, by Rule 2 of Definition 7.1, we know that a transaction at level i might call transactions from multiple levels $x > i$, not just $x = i+1$. Thus, we must change the definition of abstract serializability slightly. Instead of reordering just $\mathcal{Z}_{i+1}$ while serializing transactions at level-i, we must potentially reorder $\mathcal{Z}_x$ for all x where transactions at level i can call transactions at level x. Even in this case, if every module is properly locked (by the same definition as above), one can show serializability by modules guarantees abstract serializability.

The methodology of open nesting often requires the notion of compensating actions or inverse actions. For instance, in a `BST`, the inverse of `insert` is `remove` with the same

[14] This work assumes that level number increases going from a higher level to a lower-level to be consistent with the numbering of `xid`. In the literature (e.g. [124]), levels typically go in the opposite direction.

[15] More complicated locks can be modeled by generalizing the definition of conflict.

key. When a transaction T aborts, all the changes made by its subtransactions must be inverted. Again, although the OAT model does not explicitly model compensating actions, it allows an aborting transaction with status `PENDING_ABORT` to perform an arbitrary but finite number of operations before changing the status to `ABORTED`. Therefore, an aborting transaction can compensate for all its aborted subtransactions.

7.5 Deadlock Freedom

This section argues that the OAT model described in Section 7.3 can never enter a "semantic deadlock" if suitable restrictions are imposed on the memory accessed by a transaction's abort actions. More precisely, an abort action generated by a transaction T from $\texttt{xMod}(T)$ should read from a memory location ℓ belonging to $\texttt{modAnces}(\texttt{xMod}(T))$ only if ℓ is already in $\texttt{R}(T)$, and similarly it should write to an ℓ belonging to $\texttt{modAnces}(\texttt{xMod}(T))$ only if $\ell \in \texttt{W}(T)$. Under these conditions, one can show that the OAT model can always "finish" reasonable computations.

An ordinary TM without open nesting and with eager conflict detection never enters a semantic deadlock because it is always possible to finish aborting a transaction T without generating additional conflicts. In particular, a scheduler in the TM runtime can abort all transactions and then complete the computation by running the remaining transactions serially.

Using the OAT model, however, a TM system can enter a semantic deadlock because it can enter a state in which it is impossible to finish aborting two parallel transactions T_1 and T_2 which both have status `PENDING_ABORT`. If T_1's abort action generates a memory operation u which conflicts with T_2, then u will wait for T_2 to finish aborting (i.e., wait for the status of T_2 to become `ABORTED`). Similarly, T_2's abort action can generate an operation v which conflicts with T_1 and waits for T_1 to finish aborting. Thus T_1 and T_2 can both wait on each other, and neither transaction will ever finish aborting.

Defining Semantic Deadlock

Intuitively, we say that the OAT model exhibits a semantic deadlock if it causes the TM system to enter a state in which it is impossible to finish a computation because of transaction conflicts. A computation might not finish for other reasons, such as an infinite loop or livelock. This section defines semantic deadlock precisely and distinguishes it from these other reasons that a computation might fail to complete.

Conceptually, the execution of a computation can be modeled using two entities: the program, and a generic operational model $\mathcal{F}$ representing the runtime system. At any time t, given a ready node $X \in \texttt{ready}(\mathcal{C})$, the program chooses an instruction, and has X issue the instruction. If the program issues an infinite number of instructions, then $\mathcal{F}$ cannot complete the program no matter what it does. To eliminate programs which have infinite loops, we only consider ***bounded programs***.

Definition 7.8. *A program is **bounded** for an operational model $\mathcal{F}$ if any computation tree that $\mathcal{F}$ generates for that program has a finite depth and there exists a finite number K*

such that at any time t, every node $B \in \texttt{nodes}^{(t)}(C)$ has at most K children with status PENDING *or* COMMITTED.

Even if the program is bounded, it might run forever if it ***livelocks***. We use the notion of a ***schedule*** to distinguish livelocks from semantic deadlocks.

Definition 7.9. *A* ***schedule*** *Γ on some time interval $[t_0, t_1]$ is the sequence of nondeterministic choices made by an operational model in the interval.*

An operational model $\mathcal{F}$ makes two types of nondeterministic choices. First, at any step t, $\mathcal{F}$ nondeterministically chooses which ready node $X \in \texttt{ready}(C)$ executes an instruction. This choice models nondeterminism in the program due to interleaving of the parallel executions. Second, while performing a memory operation u which generates a conflict with transaction T, $\mathcal{F}$ nondeterministically chooses to abort either $\texttt{xParent}(u)$ or T. This nondeterministic choice models the contention manager of the TM runtime. A program may livelock if $\mathcal{F}$ repeatedly makes "bad" scheduling choices.

Intuitively, an operational model deadlocks if it allows a *bounded computation* to reach a state where *no schedule* can complete the computation after this point.

Definition 7.10. *Consider an operational model $\mathcal{F}$ executing a bounded computation. We say that $\mathcal{F}$ exhibits a* ***semantic deadlock*** *if there exists a finite sequence of t_0 instructions that $\mathcal{F}$ can issue to generate an intermediate computation C_0 such that no finite schedule of instructions Γ on $[t_0, t_1]$ can bring the computation tree from C_0 into a rest state C_1, i.e., $\texttt{ready}(C_1) = \{\texttt{root}(C_1)\}$.*

This definition is sufficient, since once the computation tree has only the root node as ready, $\mathcal{F}$ can complete the computation by executing each transaction serially.

Restrictions to Avoid Semantic Deadlock

The general OAT model described in Section 7.3 exhibits semantic deadlock because it may enter a state where two parallel aborting transactions T_1 and T_2 keep each other from completing their aborts. For a restricted set of programs however, where a PENDING_ABORT transaction T never accesses new memory belonging to Xmodules at $\texttt{xMod}(T)$'s level or higher, we show that the OAT model is free of semantic deadlock.

More formally, for all transactions T, consider the following restriction on the memory footprint of $\texttt{abortactions}(T)$.

Definition 7.11. *A computation C has* ***abort actions with limited footprint*** *if for all transactions $T \in \texttt{aborted}(C)$, whenever a memory operation $v \in \texttt{abortactions}(T)$ accesses location ℓ on a step t and $\texttt{owner}(\ell) \in \texttt{modAnces}(\texttt{xMod}(T))$, then we have the following:*

1. *If v is a read, then $\ell \in R^{(t)}(T)$.*
2. *If v is a write then $\ell \in W^{(t)}(T)$.*

Intuitively, Definition 7.11 requires that once the status of a transaction T changes to PENDING_ABORT, any memory operation v which T or a nested transaction inside T performs to finish aborting T cannot read from (or write to) any location ℓ which is owned by any Xmodules which are ancestors of $\texttt{xMod}(T)$ (including $\texttt{xMod}(T)$ itself), unless ℓ was already in the readset (or writeset) of T.

First, we can show that the properties of Xmodules from Theorem 7.5 in combination with the ownership-aware commit mechanism imply that transaction readsets and writesets exhibit nice properties. In particular, the following corollary states that a location ℓ can appear in the readset of a transaction T only if T's Xmodule is a descendant of $\texttt{owner}(\ell)$ in the module tree $\mathcal{D}$.

Corollary 7.10. *In an ownership-aware TM, consider any transaction T. If $\ell \in R(T)$, then we have $\texttt{xMod}(T) \in \texttt{modDesc}(\texttt{owner}(\ell))$.*

Proof. The proof follows from Definition 7.1 and Theorem 7.5 and induction on how a location ℓ can propagate into readsets and writesets using the ownership-aware commit mechanism. □

If all abort actions have a limited footprint, the following lemma implies that the operations of an abort action of an Xmodule M can only generate conflicts with a "lower-level" Xmodule.

Lemma 7.11. *Suppose that the OAT model generates an computation whose abort actions have limited footprint. For any transaction T, consider a potential memory operation $v \in \texttt{abortactions}(T)$. If v conflicts with transaction T' according to Definition 5.17, then $\texttt{xid}(\texttt{xMod}(T')) > \texttt{xid}(\texttt{xMod}(T))$.*

Proof. Suppose that $v \in \texttt{abortactions}(T)$ accesses a memory location ℓ with $\texttt{owner}(\ell) = M$. Since we have $\texttt{abortactions}(T) \subseteq \texttt{memOps}(T)$, by the properties of Xmodules in Definition 7.2, we know that either $M \in \texttt{modAnces}(\texttt{xMod}(T))$, or $\texttt{xid}(M) > \texttt{xid}(\texttt{xMod}(T))$. If $M \in \texttt{modAnces}(\texttt{xMod}(T))$, then by Definition 7.11, T already had ℓ in its readset or writeset. Therefore, using Definition 5.17, v cannot generate a conflict with T' because then T would already have had a conflict with T' before v occurred, contradicting the eager conflict detection of the OAT model.

Thus, we have $\texttt{xid}(M) > \texttt{xid}(\texttt{xMod}(T))$. If v conflicts with some other transaction T', then T' has ℓ in its readset or writeset. Therefore, from Corollary 7.10, $\texttt{xMod}(T') \in \texttt{modDesc}(M)$. Thus, we have $\texttt{xid}(\texttt{xMod}(T')) > \texttt{xid}(M) > \texttt{xid}(\texttt{xMod}(T))$. □

Theorem 7.12. *The OAT model is free from semantic deadlock for computations whose abort actions have limited footprint.*

Proof. Let $\mathcal{C}_0$ be the computation tree after any finite sequence of t_0 instructions. We describe a schedule Γ which finishes aborting all transactions in the computation by executing abort actions and transactions serially.

Without loss of generality, assume that at time t_0, $\texttt{status}(T) = \texttt{PENDING\_ABORT}$ for all active transactions T. Otherwise, the first phase of the schedule Γ is to make this status change for all active transactions T.

For a module tree $\mathcal{D}$ with $k = |\mathcal{D}|$ Xmodules (including the $\texttt{world}$), we construct a schedule Γ with k phases, numbered $k-1, k-2, \ldots 1, 0$. The invariant we maintain is that immediately before phase i, we bring the computation tree into a state $\mathcal{C}^{(i)}$ which has no active transaction instance T with $\texttt{xid}(\texttt{xMod}(T)) > i$, i.e., no instance T from an Xmodule with `xid` larger than i. During phase i, we finish aborting every active transaction instance T with $\texttt{xid}(\texttt{xMod}(T)) = i$. By Lemma 7.11, for any transaction instance T with $\texttt{xid}(\texttt{xMod}(T)) = i$, any abort action for such a T can conflict with a transaction instance T' only from a lower-level Xmodule, i.e., $\texttt{xid}(\texttt{xMod}(T')) > i$. Since the schedule Γ executes serially, and by the invariant on $\mathcal{C}^{(i)}$ we have already finished all active transaction instances from lower levels, phase i can finish without generating any conflicts. □

Restrictions on Compensating Actions

If transactions $Y_1, Y_2, \ldots Y_j$ are nested inside a transaction X that aborts, the abort actions of X typically consist of the compensating actions for $Y_1, Y_2, \ldots Y_j$. Thus, restrictions on abort actions translate in a straightforward manner to restrictions on compensating actions: a compensating action for a transaction Y_i (which is part of the abort action of X) should not read (write) any memory owned by $\texttt{xMod}(X)$ or its ancestor Xmodules unless the memory location is already in X's readset (writeset). Assuming that abstract locks are modeled as accesses to memory locations, a similar restriction applies: a compensating action cannot acquire new locks that were not already acquired by the transaction it is compensating for.

7.6 Related Work

This section describes other work in the literature on open-nested transactions. In particular, this section focuses on two related approaches for improving open-nested transactions and distinguishes them from ownership-aware transactions.

The database community has produced an extensive literature on nested transactions. Moss [100] credits Davies [41] with inventing nested transactions, and he credits Reed [109] as providing the first implementation of what we now call closed transactions. Gray [55] describes what we now call open transactions. The terms "open" and "closed" nesting were coined by Traiger [117] in 1983.

Ni et al. [105] propose using an `open_atomic` class to specify open-nested transactions in a Java-like language with transactions. Since the private fields of an object with an `open_atomic` class type cannot be directly accessed outside of that class, one can think of the `open_atomic` class as defining an Xmodule. This mapping is not exact, however, because neither the language or TM system restrict exactly what memory can be passed into a method of an `open_atomic` class, and the TM system performs a vanilla open-nested commit for a nested transaction, not a safe-nested commit. Thus, it is unclear what exact guarantees are provided with respect to serializability and/or deadlock freedom.

Herlihy and Koskinen [62, 63] describe a technique of transactional boosting which allows transactions to call methods from a nontransactional module M. Roughly, as long as M is linearizable and its methods have well-defined inverses, the authors show that the execution appears to be "abstractly serializable." Boosting does not, however, address the cases when the lower-level module M writes to memory owned by the enclosing higher-level module or when programs have more than two levels of modules. The theoretical results for ownership-aware TM do not directly apply to transactional boosting because M

can use module-specific mechanisms for synchronization. It seems difficult to provide any provable guarantees for an arbitrary nontransactional module M without requiring additional structure, e.g., limiting M to accessing only memory that it owns.

7.7 Conclusions

In this chapter, I have described ownership-aware TM, which enables a TM system to provide support for open-nested transactions, but still provide clean memory-level semantics and guarantees of abstract serializability for transactions. Ownership-aware TM bridges the gap between the intent and the execution of open-nested transactions. Open-nested transactions enable a TM runtime to ignore low-level memory conflicts while checking for conflicts between high-level transactions. Ownership-aware TM incorporates the notions of Xmodules and ownership into a TM system, thereby enabling the system to make the right decisions about which memory conflicts should be ignored.

In particular, this chapter described precise restrictions that a programming platform should impose on the interactions between Xmodules. If a program obeys these restrictions and a TM system uses an ownership-aware commit mechanism, then the TM system guarantees the correctness condition of serializability by modules. Thus, ownership-aware TM demonstrates that a parallel-programing platform can support composable transaction-based synchronization using open-nested transactions while still providing provable guarantees of safety and correctness.

Chapter 8

Conclusions

In this dissertation, I have described the design of composable abstractions for synchronization in dynamic-threading platforms based on the ideas of task-graph execution, helper locks, and transactional memory. I have also presented provably efficient runtime schedulers for supporting these abstractions.

Chapter 2 presented Nabbit, the first library in a fork-join dynamic-threading platform for provably efficient parallel execution of task graphs with arbitrary dependencies. Because Nabbit can be implemented without making any modifications to the runtime system for a fork-join platform, it enhances the composability of such platforms by enabling programmers to exploit both fork-join parallelism and task-graph parallelism simultaneously in the same program. Using Nabbit, programmers are able to compose code that utilizes these different forms of parallelism, i.e., a programmer can exploit parallelism within the COMPUTE of a task graph node using the `spawn` and `sync` constructs, or build and execute a task graph as a subcomputation of a fork-join program.

Chapter 3 presented helper locks, the first synchronization abstraction in a dynamic-threading platform to effectively exploit asynchronous task parallelism inside locked critical sections. Helper locks enhance the composability of dynamic-threading platforms by allowing programmers to invoke parallel functions from inside a locked critical section without forcing these functions to execute serially. Chapter 4 also showed that the parallel region construct used to provide runtime support for helper locks can also enhance compatibility with legacy code in Cilk-like platforms. In particular, I demonstrated that parallel regions can be used to implement a form of restricted work-stealing, which in turn enables a legacy-C function to call back to a parallel function in MIT Cilk. Thus, the parallel region construct allows programmers to compose legacy library functions and Cilk functions more effectively.

Chapters 5 through 7 explored synchronization based on transactional memory, studying the semantics and behavior of nested transactions. Chapter 5 presented the framework of transactional computations, which is useful for understanding the semantics and behavior of nested transactions in a dynamic-threading platform. Chapter 6 presented CWSTM, the first design of TM that supports closed-nested transactions with nested parallelism and nested parallel transactions of unbounded nesting depth. Chapter 7 described ownership-aware TM, the first design of TM that supports open-nested transactions with provable correctness guarantees. Support for both closed-nested and open-nested transactions enhances the composability of TM systems, as it enables a programmer to invoke library functions which have been parallelized using transactions from a transaction within their own code.

In summary, in this dissertation I have described designs for synchronization abstractions that demonstrate that dynamic-threading platforms can support composable synchronization without sacrificing provable guarantees of performance and correctness.

Appendix A

Dynamic Threading in Cilk

This appendix presents background material on Cilk [51], a dynamic-threading language for parallel programming. Section A.1 gives an overview of the Cilk programming model. Section A.2 reviews the "computation DAG" model for Cilk, an abstract model which is useful for analyzing the execution of programs using Cilk's scheduler. Section A.3 uses this computation DAG model to state the provable bounds on program completion time guaranteed by Cilk's randomized work-stealing scheduler. Section A.4 presents the "computation tree" model, another abstract model for Cilk. Section A.5 uses this alternative model to understand the bounds on stack space usage provided by Cilk. Section A.6 briefly discusses other work related to Cilk.

This dissertation uses the computation DAG and tree models to explore extensions to Cilk. Chapters 2 and 3 uses the computation DAG model to prove completion-time bounds for the Nabbit and HELPER systems, respectively. Chapter 4 uses computation trees to describe a lower-bound example computation. Chapters 5 through 7 use computation trees to understand the semantics of transactional memory in a Cilk-like dynamic-threading platform. For convenience, Appendix B summarizes the notation defined in this dissertation.

A.1 Overview of Cilk

This section reviews the programming model provided by Cilk, a dynamic-threading language for parallel programming. This section first summarizes the key language constructs in Cilk and briefly reviews the algorithm and data structures used in Cilk's runtime scheduler. It then describes how Cilk supports the "cactus stack" abstraction, a generalization of the usual stack abstraction from serial programs to parallel programs.

The Cilk Language

Cilk programmers write parallel code in the various dialects [51, 73, 93] by writing ***Cilk functions*** — functions that can use the `spawn` and `sync` keywords.[1] Within a Cilk function

[1] This dissertation uses the term "Cilk" to refer to three primary dialects: MIT Cilk (also called Cilk-5) [51], Cilk++ [93], and Intel Cilk Plus [73]. Although Cilk++ and Intel Cilk Plus have minor syntactic differences compared to MIT Cilk, they exhibit mostly the same fundamental characteristics in terms of the

A, a spawn of a function instance F indicates that the instance of F can potentially run in parallel with the code within A after the spawn, until a sync within A is reached. For example, for the Cilk function A shown in Figure A-1, the spawn in line 3 indicates that the instance of F can potentially run in parallel with lines 4 through 7. Control cannot continue past the sync in line 8 however, until all previously spawned functions (lines 3 and 5) have completed. In contrast, for a function G that is ***called*** from A (e.g., line 11), G must complete before the code after the call (e.g., line 12) can be executed.

For each sync statement in a Cilk function, we define its ***sync block*** as the code between the first spawn statement associated with the sync and the sync statement itself. For example, the function A in Figure A-1 has two sync blocks, lines 3–8 and lines 10–13. Figure A-2 captures the parallel structure of A as a ***directed acyclic graph***, or ***DAG***.

Cilk is an example of a ***dynamic-threading platform*** — a platform for parallel programming that utilizes an integrated dynamic runtime scheduler.[2] In a platform that supports dynamic threading, or ***dthreading*** for short, programmers only expose *potential* parallelism in a program, e.g., using spawn and sync. To execute program, users specify at runtime a value for P, the number of ***worker threads*** the scheduler should use to execute the program. During execution, the platform dynamically schedules the program on these P worker threads, i.e., it determines how different tasks in a program are split between worker threads. Dynamic threading platforms have proliferated as multicore technology has taken hold. Examples of such platforms include Cilk-1 [29],[3] Cilk-5 [51], Cilk++ [93], Cilk Plus [73], Fortress [13], Hood [31], Java Fork/Join Framework [90], JCilk [40], OpenMP 3.0 [106], Task Parallel Library (TPL) [92], Threading Building Blocks (TBB) [110], and X10 [37].

To schedule computations efficiently, Cilk's scheduler maintains a double-ended queue, or ***deque***, for each worker thread, and utilizes randomized ***work-stealing*** to load-balance work between worker threads. In Cilk, a worker thread p normally pushes and pops work from the ***tail*** (bottom) of its deque. When worker p finds its own deque empty, however, it chooses a victim worker p' uniformly at random from one of the $P-1$ other workers, and tries to steal work from the ***head*** (top) of the deque of p'. As discussed in greater detail in Section A.2, this scheduler provides strong provable guarantees on performance.

More precisely, normally in Cilk, workers push and pop "continuations" of functions to and from deques. Whenever a worker p encounters a spawn of a function F within a function A, p begins executing F, and pushes the ***continuation*** of F (in A) onto the bottom of its deque. For example, in the code in Figure A-1, a worker p executing the spawn in line 3 starts executing F(0) and pushes the continuation (i.e., execution of A beginning at line 4) onto its deque. Usually, when worker p returns from F(0), it discovers that this continuation has not been stolen, and p then pops the continuation off the bottom of its

language and runtime scheduling. Cilk++ [93] uses the keywords cilk_spawn and cilk_sync (instead of spawn and sync). Cilk++ also provides a cilk_for keyword for coding parallel loops. A cilk_for loop can be implemented using cilk_spawn and cilk_sync. Intel Cilk Plus [73] uses similar keywords to Cilk++. MIT Cilk also requires programmers to use the cilk keyword as a type qualifier in front of Cilk functions. Cilk++ [93] and Intel Cilk Plus [73] eliminate the cilk keyword through additional compiler support.

[2] In the literature, Cilk's programming model is also sometimes classified as fork-join parallelism or (lightweight) task parallelism. By this definition, these models are specific instances of dynamic threading.

[3] Called "Cilk" in [29], but renamed "Cilk-1" in [51] and other MIT documentation.

```
 1 int A() {
 2     int x = 0;
 3     spawn F(x);         // F(0)
 4     x+=1;               // x == 1
 5     spawn F(x);         // F(1)
 6     x+=2;               // x == 3
 7     G(x);               // G(3)
 8     sync;
 9     x+=3;               // x == 6
10     spawn F(x);         // F(6)
11     G(x);               // G(6)
12     x+=4;               // x == 10
13     sync;
14     x+=5;               // x == 15
15     return x;
16 }
```

Figure A-1: A simple program demonstrating the `spawn` and `sync` keywords in Cilk. Comments in the program track the value of `x` as `A()` executes; all updates to `x` in `A()` are totally ordered.

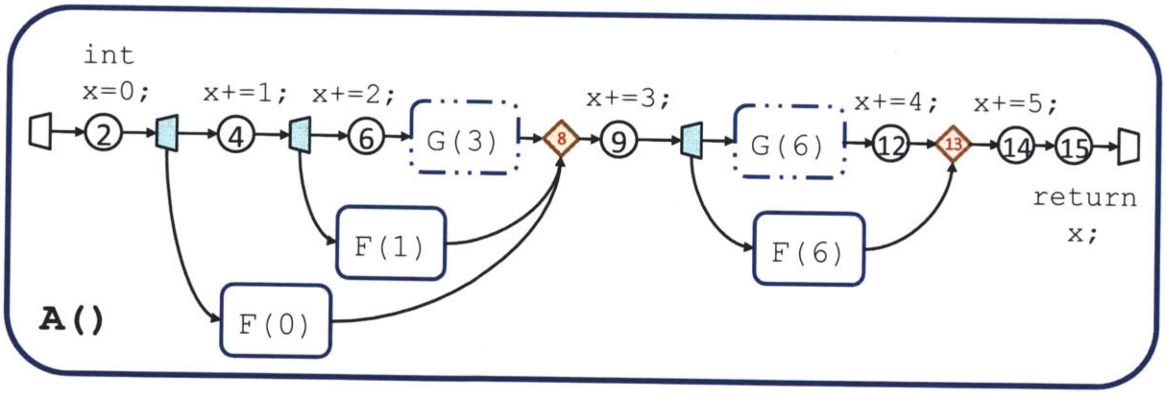


Figure A-2: The computation DAG for the Cilk function `A()` from Figure A-1. The labels for circles correspond to line numbers in Figure A-1. Functions which are spawned (`F`) have a solid border, while functions that are called (`G`) have a dashed border. Filled trapezoids and diamonds correspond to `spawn` and `sync` statements, respectively.

deque and resume execution of A at line 4.

On the other hand, worker p may discover that the continuation has been stolen when it returns from a spawned function F. In this case, p conceptually jumps to the corresponding sync for the sync block, and then proceeds according to one of two cases: p can either finish a sync block, or it can stall and begin work-stealing. In the first case, if all the work in the sync block of the sync statement has completed, then p resumes execution after the sync. Otherwise, in the second case, p stalls at the sync, and then p tries to steal a continuation via randomized work-stealing.[4]

The Cactus-Stack Abstraction

Cilk supports the "cactus stack" abstraction, a generalization of the ordinary stack used in serial programs to parallel code.

An execution of a serial Algol-like language, such as C [78] or C++ [114], can be viewed as a "walk" of a dynamically unfolding ***invocation tree***, a tree that relates function instances by the "calls" relation. More precisely, if function instance A calls function instance B, then A is a ***parent*** of the ***child*** B in the invocation tree. Such serial languages use a ***linear-stack representation*** — a simple array-based stack for allocating function activation frames, where frames for caller and callee are allocated in contiguous space. With a linear stack, the stack pointer is advanced as a function is invoked and restored as the function returns. This scheme is space-efficient, because all the children of a given function can use and reuse the same region of the stack. This reuse depends critically, however, on the property of a serial language that a function has at most one extant child function at any time.

In a dynamic-threading language such as Cilk, a parent function can also spawn a child, thereby creating parallelism. The notion of an invocation tree can be extended to include spawns as well as calls. Unlike the serial walk of an invocation tree, however, a parallel execution unfolds the invocation tree more haphazardly and in parallel. Since multiple children of a function may be extant simultaneously, a linear-stack data structure no longer suffices for storing activation frames. Instead, the tree of extant activation frames forms a ***cactus stack*** [61]. In a cactus stack, a function F can access the stack variables in its frame and in any frames which are ancestors (in the invocation tree) of F's frame. The stack frame for a given function F might be shared among multiple workers if more than one descendant frame of F is active at the same time.

MIT Cilk [51] and Cilk++ [93] support the cactus stack abstraction by allocating frames for Cilk functions in noncontiguous space, where each frame is linked to its parent frame. These frames in the noncontiguous memory are referred as ***shadow frames***, to differentiate them from the ***activation frames*** in the linear stacks. During program execution, each worker pushes on and pops off shadow frames from the bottom of its own deque. The shadow frames in the deque mirror the activation frames in the worker's linear stack for the corresponding Cilk functions.

The call / return linkage for a Cilk function (henceforth referred to as the ***Cilk linkage***)

[4] In Cilk, this second case occurs only if p has an empty deque, since once p reaches a sync in a function F, F must be at the bottom of p's deque.

differs from the ordinary C linkage. Cilk passes parameters and returns values via shadow frames instead of activation frames. Thus, if a parent passes a pointer of its local variable to its child, the pointer refers to the location in the shadow frame. When a worker's deque is empty, its corresponding linear stack is empty as well, i.e., the worker can freely pop off the suspended activation frames in its linear stack. With this strategy, multiple extant children can share a single view of their parent frame simultaneously, as required by the cactus stack abstraction. As discussed later in Section A.5, this implementation allows MIT Cilk and Cilk++ to guarantee a good stack space bound. Intuitively, since a worker only steals when its deque is empty, each worker uses no more stack space than the space used by the serial execution of the program.

Unfortunately, because MIT Cilk and Cilk++ use shadow frames, they also use an interprocedural calling convention to `spawn` a function that is incompatible with the normal stack-based calling convention of serial languages. Therefore, MIT Cilk and Cilk++ do not exhibit the property of "SP-reciprocity", as defined by Lee et al. in [91]. As discussed in Chapter 4, a lack of SP-reciprocity can make it difficult to compose Cilk code with legacy (and third-party) serial binaries that are compiled assuming an ordinary C linkage.

A.2 Computation DAG Model

This section reviews the ***computation DAG model***, an abstract model which is useful for modeling the execution and analyzing the running time of Cilk programs. First, we review how one can construct an *a posteriori* model of a Cilk program execution as an "execution DAG." Then, we consider an abstract execution model that explains how a computation DAG is conceptually generated by the execution of a Cilk program. Chapter 3 extends this computation DAG model for Cilk to apply to the HELPER system.

Definitions

For the purposes of theoretical analysis, one can conceptually consider the execution of a Cilk program as generating an ***execution DAG*** $\mathcal{G}$ with nodes $V(\mathcal{G})$ and edges $E(\mathcal{G})$ [38, p. 777]. This execution DAG is not given *a priori* (e.g., from a static analysis of the program), but is an *a posteriori* representation of the program after it executes.

To describe the structure of execution DAGs, it is convenient to define some notation. For any node u in $V(\mathcal{G})$, let $\texttt{inDeg}(u)$ and $\texttt{outDeg}(u)$ denote the in-degree and out-degree of u, respectively. For any node $u \in V(\mathcal{G})$, let $\texttt{ipred}(v)$ be the set of ***immediate predecessors*** of v in $\mathcal{G}$, i.e., $u \in \texttt{ipred}(v)$ if and only if there is an edge $(u,v) \in E(\mathcal{G})$. When it is clear that $\texttt{ipred}(v)$ has exactly one element (e.g., $\texttt{ipred}(v) = \{u\}$), we abuse notation and say that $\texttt{ipred}(v) = u$. Similarly, let $\texttt{isucc}(u)$ be the set of ***immediate successors*** of u, i.e., $v \in \texttt{isucc}(u)$ if and only if there is an edge $(u,v) \in E(\mathcal{G})$.

We assume that the execution of a Cilk program generates a canonical execution DAG $\mathcal{G}$ that satisfies three types of constraints. First, we assume that every node u in $V(\mathcal{G})$ falls into one of three categories: u is either a spawn node, a sync node, or a serial node. A ***spawn node*** u is a node with $\texttt{inDeg}(u) = 1$ and $\texttt{outDeg}(u) = 2$. A ***sync node*** v is a node with $\texttt{inDeg}(v) > 1$ and $\texttt{outDeg}(v) = 1$. All other nodes w are ***serial nodes***—nodes with

$\texttt{inDeg}(w) = \texttt{outDeg}(w) = 1$. In a Cilk program, the spawn and sync statements correspond to spawn and sync nodes in $\mathcal{G}$, respectively. We assume in $\mathcal{G}$ that there are no edges (u,v) directly from a spawn node u to another spawn or sync node v, i.e., there is always at least one serial node in the continuation of any spawn or sync statement.

Because Cilk computation DAGs have this canonical form, one can show that $\mathcal{G}$ is always a minimal DAG representation, i.e., $\mathcal{G}$ is equal to its transitive reduction [12]. Since $\mathcal{G}$ has a minimal number of edges representing dependencies in the computation, the sets $\texttt{ipred}(u)$ and $\texttt{isucc}(u)$ for a node u are well-defined.

Second, we assume that invocations of functions are enclosed between special source and sink nodes in the execution DAG $\mathcal{G}$. For each function invocation F, the subDAG of $\mathcal{G}$ corresponding to F is enclosed between a ***source node*** $\texttt{source}(F)$ and a ***sink node*** $\texttt{sink}(F)$. For every F, $\texttt{source}(F)$ and $\texttt{sink}(F)$ are serial nodes. For every function F which is spawned, $\texttt{ipred}(\texttt{source}(F))$ is a spawn node and $\texttt{isucc}(\texttt{sink}(F))$ is its corresponding sync node. Thus, for any sync node v, all but one of the nodes $u \in \texttt{ipred}(v)$ are sink nodes of spawned functions.

Finally, because functions in Cilk are properly nested, one can associate every node $u \in V(\mathcal{G})$ with a unique function F that "owns" u, and many functions that "contain" u. More formally, we say that a function F ***contains*** all nodes u that are along any path between $\texttt{source}(F)$ and $\texttt{sink}(F)$. We say that F ***owns*** u, or that u ***belongs to*** F, if F is the most deeply nested function that contains u.

Execution Model

To analyze the properties of the Cilk platform, we shall utilize the Cilk computation model, an abstract execution model that explains how a computation DAG $\mathcal{G}$ is conceptually generated by an execution of a Cilk program. This model is based off the abstract model of Arora et al. described in [16].

The ***Cilk computation model*** models the execution of a Cilk program on P processors as a parallel traversal of an execution DAG $\mathcal{G}$. In this model, each processor is mapped to a single ***worker*** thread p. On each time step, each worker p can either be executing a node from $V(\mathcal{G})$ which is "ready," or be looking for work through randomized work-stealing. More formally, a node v is ***ready*** if all of v's predecessors have already been executed. Each worker p maintains a ***deque*** of ready nodes, denoted by $p.dq$, and an ***assigned*** node, denoted by $p.assigned$. In any time step for which a worker p has work available, $p.assigned$ represents the node p executes on that step. Otherwise, we say that $p.assigned$ is NULL.

In the Cilk computation model, each p can execute one the ***instructions*** described in Figure A-3 on each time step. Instructions 1 through 3 are the instructions that are required for serial programs. Instructions 4 through 7 represent instructions that Cilk introduces for parallel execution. A worker can steal (Instruction 7) only if $p.assigned = \text{NULL}$, i.e., only on steps when p has no work.

The actions taken by the instructions in the Cilk computation model are mostly straightforward, with the most complicated instructions being the sReturn and the sync instructions. The sReturn (instruction 6) can perform one of three actions, depending on how the computation is scheduled:

(a) **Finish**: If executing the assigned node u makes $v = \texttt{isucc}(u)$ ready, i.e., all the work in the sync block is completed, then $p.assigned$ is set to v.

(b) **Stall**: If executing the assigned node u does not make $\texttt{isucc}(u)$ ready, and $p.dq$ is empty, then $p.assigned$ is set to NULL and p begins work-stealing in the next step.

(c) **Continue**: Otherwise, $p.dq$ is not empty, and $p.assigned$ is set to the node x popped from the bottom of $p.dq$.

The `sync` instruction for p behaves similarly to the `sReturn` instruction, except without the continue action. More precisely, if a worker p reaches a `sync` statement in a function F, but stalls because the sync block is not yet complete, Cilk guarantees that $p.dq$ must be empty.

A.3 Completion-Time Bound for Cilk

This section states the completion-time bounds [30] for Cilk computations using the computation DAG model described in Section A.2. This section also summarizes one technique for proving these bounds [16]. Chapters 2, 3, and 6 generalize this completion-time bound for extensions of Cilk.

To state the completion-time bound, we require some definitions. In an execution DAG $\mathcal{G}$, we assume all operations have been expanded so that each node $\mathcal{G}$ represents a unit amount of work. This assumption allows us to bound the running time of a Cilk program in terms of properties of its DAG $\mathcal{G}$. More formally, for an execution DAG $\mathcal{G}$, define the ***work*** of $\mathcal{G}$, denoted by $T_1(\mathcal{G})$, as the number of nodes in $\mathcal{G}$, i.e., $T_1(\mathcal{G}) = |V(\mathcal{G})|$. Define the ***span*** (or critical path length) of $\mathcal{G}$, denoted by $T_\infty(\mathcal{G})$, as the number of nodes along a ***critical*** (longest) path through $\mathcal{G}$ from $\texttt{source}(\mathcal{G})$ to $\texttt{sink}(\mathcal{G})$. More generally, we can define work and span for any subDAG of $\mathcal{G}$. For example, in Figure A-4, if F corresponds to the subDAG for `F(0)`, then $T_1(F) = 6$ and $T_\infty(F) = 5$. As an abuse of notation, we sometimes omit the argument from the functions when referring to the work and span of the entire execution DAG, i.e., $T_1 = T_1(\mathcal{G})$ and $T_\infty = T_\infty(\mathcal{G})$.

Work and span are useful quantities for bounding completion time. In the Cilk computation model, $T_1(\mathcal{G})$ is the time required for a single processor to execute $\mathcal{G}$. Similarly, $T_\infty(\mathcal{G})$ is the time required to execute $\mathcal{G}$ using an infinite number of processors, assuming there is no synchronization or communication overhead between processors. Any scheduler requires at least $\max\{T_1(\mathcal{G})/P, T_\infty(\mathcal{G})\}$ time to execute $\mathcal{G}$ on P processors: the first term is a lower bound on the number of steps needed to execute all the nodes in $\mathcal{G}$, and the second term is a lower bound on the time to execute nodes along a critical path in $\mathcal{G}$.

Blumofe and Leiserson show in [30] that a Cilk-like randomized work-stealing scheduler can execute computations efficiently, i.e., they prove the result in Theorem A.1.

Theorem A.1. *Let $\mathcal{G}$ be a computation executed on P processors with work $T_1 = T_1(\mathcal{G})$ and span $T_\infty = T_\infty(\mathcal{G})$. Then Cilk's randomized work-stealing scheduler can execute $\mathcal{G}$ in time*

$$O\left(\frac{T_1}{P} + T_\infty + \lg P + \lg(1/\varepsilon)\right)$$

with probability at least $1 - \varepsilon$.

	Instruction	Precondition	Updates
1.	**`primOp`**: Primitive operation, e.g., an arithmetic or memory operation.	$p.\mathit{assigned} = u$ u is a serial node	$p.\mathit{assigned} = \texttt{isucc}(u)$
2.	**`call`** Call of a function F.	$p.\mathit{assigned} = u$ $u = \texttt{source}(F)$ u is a serial node	$p.\mathit{assigned} = \texttt{isucc}(u)$
3.	**`cReturn`** Return from a called function F.	$p.\mathit{assigned} = u$ $u = \texttt{sink}(F)$ u is a serial node	$p.\mathit{assigned} = \texttt{isucc}(u)$
4.	**`spawn`** Spawn of a function F.	$p.\mathit{assigned} = u$ $\texttt{isucc}(u) = \{v, w\}$ $w = \texttt{source}(F)$	$p.\mathit{assigned} = w$ push v onto $p.\mathit{dq}$

5.	**`sync`**
	Control reaches a sync inside a function F.
		$p.\mathit{assigned} = u$
		$v = \texttt{isucc}(u)$
		v is a sync node
		$S = \{\texttt{ipred}(v)\}$
			if all $w \in S$ have executed:
			  $p.\mathit{assigned} = v$
			else:
			  $p.\mathit{assigned} = \text{NULL}$

6.	**`sReturn`**
	Return from a spawned function F.
		$p.\mathit{assigned} = u$
		$u = \texttt{sink}(F)$
		$v = \texttt{isucc}(u)$
		v is a sync node
		$S = \{\texttt{ipred}(v)\}$
			if all $w \in S$ have executed:
			  $p.\mathit{assigned} = v$
			else:
			  if $p.\mathit{dq}$ is empty:
			    $p.\mathit{assigned} = \text{NULL}$
			  else:
			    pop x from $p.\mathit{dq}$
			    $p.\mathit{assigned} = x$

7.	**`steal`** Worker p attempts to steal from a randomly chosen victim $p' \neq p$	$p.\mathit{assigned} = \text{NULL}$	**if** $p'.\mathit{dq}$ is not empty: x is top node in $p'.\mathit{dq}$ $p.\mathit{assigned} = x$ **else**: $p.\mathit{assigned} = \text{NULL}$

Figure A-3: Instructions for the Cilk computation model. On each time step, a worker p executes one of these instructions whose precondition is satisfied.

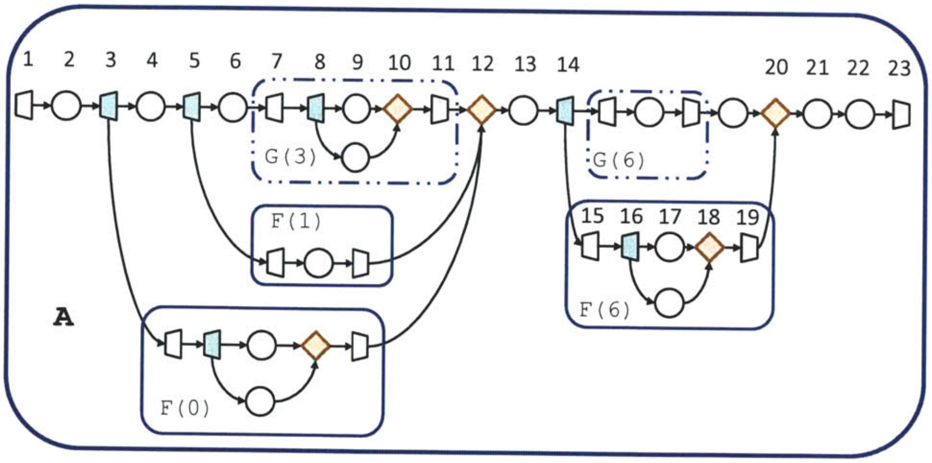


Figure A-4: An execution DAG illustrating work T_1 and span T_∞. This DAG corresponds to the code in Figure A-2, except with nested functions expanded. In this DAG, we have $T_1(A) = 38$ and $T_\infty(A) = 23$.

Theorem A.1 implies that Cilk computations which have sufficient "parallelism" will speed up linearly as more processors are added. More precisely, define the ***parallelism*** of $\mathcal{G}$ as $T_1(\mathcal{G})/T_\infty(\mathcal{G})$. If the parallelism is sufficiently large, i.e., $T_1/T_\infty \gg P$, then (ignoring the $\lg P + \lg(1/\varepsilon)$ terms) the completion time in Theorem A.1 is dominated by the work term, and the running time is $O(T_1/P)$. In this situation, we say that Cilk achieves ***linear speedup*** on $\mathcal{G}$. In practice, the constant on the work term T_1/P in Theorem A.1 is nearly 1 for Cilk, with most of the scheduling overheads appearing in the constant on the T_∞ term.

Analysis of Randomized Work-Stealing

To prove the time bound in Theorem A.1, we utilize the Cilk computation model described in Section A.2 and the potential-function argument presented by Arora et al. in [16]. Chapter 3 extends this model to analyze the performance of HELPER, which allows programs to have parallel critical regions protected by locks.

In the Cilk computation model, each instruction except the `steal` instruction works on an assigned node u and replaces it with either a successor of u or NULL. Since each node u can be assigned for at most one step, there can be at most T_1 processor-steps spent on these instructions. Thus, the key challenge for a theoretical analysis is bounding the time spent stealing.

To bound the number of steal attempts, Arora et al. [16] use a potential function $\Phi(\mathcal{G})$ that accounts for the nodes in $\mathcal{G}$ that are ready or currently executing. In particular, they show that $\Phi(\mathcal{G})$ only decreases with time and that with probability at least $1-\varepsilon$, at most $O(PT_\infty + P\lg(1/\varepsilon))$ steal attempts can occur before the potential reduces to 0. Thus, steal attempts add only the terms $O(T_\infty + \lg(1/\varepsilon))$ to the time bound in Theorem A.1.

To define $\Phi(\mathcal{G})$, we require two auxiliary definitions. For every node $u \in V(\mathcal{G})$, define the ***depth*** $d(u)$ in $\mathcal{G}$ as the length of the longest path in $\mathcal{G}$ from `source`($\mathcal{G}$) to u. Define the

weight of a node u as $w(u) = T_\infty(\mathcal{G}) - d(u)$.[5]

Definition A.1. *Define the **potential** of node u as*

$$\Phi(u) = \begin{cases} 3^{2w(u)-1} & \textit{if } u \textit{ is assigned,} \\ 3^{2w(u)} & \textit{if } u \textit{ is ready,} \\ 0 & \textit{if } u = \text{NULL}. \end{cases}$$

*Similarly, let $q = p.dq$ and let $u = p.assigned$. Define the **deque potential** as*

$$\Phi(q) = \Phi(u) + \sum_{v \in q} \Phi(v) .$$

*Finally, define the **potential** of an execution DAG $\mathcal{G}$, denoted by $\Phi(\mathcal{G})$, as the sum of the deque potentials $\Phi(q)$ for the deques of all P processors.*

The remainder of this section gives a brief sketch of Arora et al.'s potential-function argument. Chapter 3 describes more details of the analysis in [16] and generalizes it to apply to HELPER.

For Cilk-like schedulers, where each processor pushes and pops from the bottom of its deque but steals from the top of a deque, we have the property that the top node on every deque q contains a constant fraction of the total potential $\Phi(q)$. Intuitively, as shown in [16] using a weighted balls-and-bins argument, after a "round" of P steal attempts have occurred, the potential of the computation reduces by a constant fraction with at least a constant probability. Since the initial potential begins at 3^{2T_∞}, only decreases over time, and is always an integer, we can conclude that the potential reduces to 0 after $O(T_\infty)$ rounds of steal attempts in expectation. Thus, we have at most $O(PT_\infty)$ steal attempts in expectation when executing $\mathcal{G}$.

A.4 Computation-Tree Framework

This section describes the ***computation-tree*** model, an alternative model for representing Cilk computations which is based on the notion of a "series-parallel parse tree." Chapters 5 through 7 use computation trees to investigate transaction-based synchronization in dynamic-threading platforms such as Cilk.

Series-Parallel Parse Trees

Feng and Leiserson [45] demonstrate that the execution of a Cilk computation can be modeled as a ***series-parallel parse tree*** — a tree C whose internal nodes model the parallel control structure of a series-parallel computation and whose leaves represent instructions or strands of serial execution.[6] Intuitively, each internal node X in a tree C is either an

[5] This definition is slightly simpler than the one given in [16] because $\mathcal{G}$ is guaranteed to be a "series-parallel" DAG.

[6] Conceptually, a series-parallel parse tree can also be used to model not only Cilk computations, but generic fork-join computations.

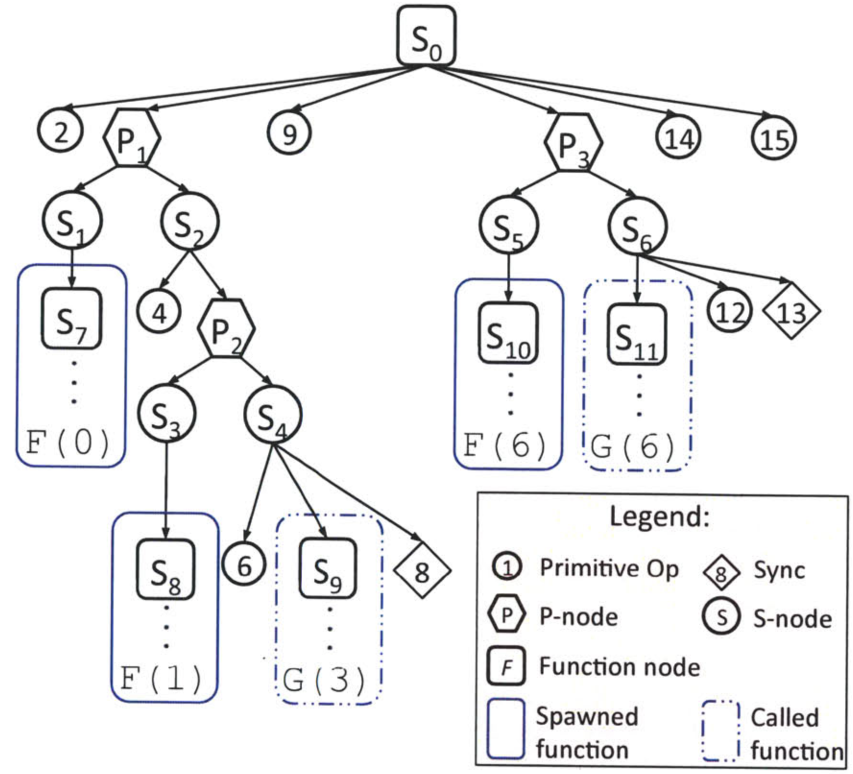


Figure A-5: A series-parallel parse tree C for the Cilk function in Figure A-1.

S-node or a P-node. For an S-node X, all the child subtrees of X must be executed in series, from left to right, but the child subtrees of a P-node are allowed to execute in parallel. For example, Figure A-5 shows a Cilk computation tree C corresponding to the computation DAG in Figure A-2.

To define these concepts more formally, a ***series-parallel parse tree*** C (or ***SP-tree*** for short) is an ordered tree with two types of nodes: ***primitive-operation nodes*** $\texttt{primOps}(C)$ at the leaves and ***control nodes*** $\texttt{spNodes}(C)$ as internal nodes. Let $\texttt{nodes}(C)$ denote the set of all nodes in C, i.e., $\texttt{nodes}(C) = \texttt{primOps}(C) \cup \texttt{spNodes}(C)$. Conceptually, we assume that each primitive operation $u \in \texttt{primOps}(C)$ represents a basic unit of work in the original computation. The control nodes $\texttt{spNodes}(C)$ can be partitioned into two sets: ***S-nodes***, denoted by $\texttt{sNodes}(C)$, and ***P-nodes***, denoted by $\texttt{pNodes}(C)$. A control node represents a control construct from the original Cilk computation. Roughly, a spawn operation corresponds to a P-node, whereas an ordinary function call corresponds to an S-node.

It is useful to define several structure notations on an SP-tree. Denote the ***root*** of the tree C as $\texttt{root}(C)$. For any node $B \in \texttt{nodes}(C)$, let $\texttt{parent}(B)$ denote the ***parent*** of B in C, or NULL if $B = \texttt{root}(C)$. Similarly, let $\texttt{children}(B)$ denote the ordered set of B's ***children***, or NULL if B is a leaf. For any tree node $B \in \texttt{nodes}(C)$, let $\texttt{ances}(B)$ denote the set of all ***ancestors*** of B in C, and let $\texttt{desc}(B)$ denote the set of all B's ***descendants***. Denote the set of ***proper ancestors*** (and proper descendants) of B by $\texttt{pAnces}(B)$ (and $\texttt{pDesc}(B)$). Denote the ***least common ancestor*** of two nodes $B_1, B_2 \in \texttt{nodes}(C)$ by $\texttt{LCA}(B_1, B_2)$.

For any set of computation-tree nodes $J \subseteq \texttt{nodes}(\mathcal{C})$, it is useful to define the ***leaf set*** of J as

$$\texttt{leafSet}(J) = \{X \in J \ : \ \texttt{pDesc}(X) \cap J = \emptyset\} \ .$$

Conceptually, $\texttt{leafSet}(J)$ is the subset of J which forms the "leaves" of J. All other nodes $Y \in J - \texttt{leafSet}(Y)$ are on the path from X to $\texttt{root}(\mathcal{C})$ for some $X \in \texttt{leafSet}(J)$. For sets J that have $|\texttt{leafSet}(J)| = 1$, we define $\texttt{leaf}(J)$ as the unique element in $\texttt{leafSet}(J)$. For example, $\texttt{leafSet}(\texttt{ances}(X))$ has a single element, $\texttt{leaf}(\texttt{ances}(X)) = X$. Similarly, we can define a ***root set*** of J which considers ancestors instead of descendants, that is,

$$\texttt{rootSet}(J) = \{X \in J \ : \ \texttt{pAnces}(X) \cap J = \emptyset\} \ .$$

For sets J with $|\texttt{rootSet}(J)| = 1$, define $\texttt{root}(J)$ as the unique element in $\texttt{rootSet}(J)$.

Since every subtree of an SP-tree is itself an SP-tree, we shall sometimes overload notation and use a subtree and its root interchangeably. For example, if $X = \texttt{root}(\mathcal{C})$, then $\texttt{primOps}(X)$ refers to all nodes in $\texttt{primOps}(\mathcal{C}) \cap \texttt{desc}(X)$.

Given any SP-tree $\mathcal{C}$, we can formally construct a computation DAG $\mathcal{G} = (V(\mathcal{C}), E(\mathcal{C}))$ using the following procedure.

1. First, construct the vertices $V(\mathcal{C})$ of $\mathcal{G}$. For every internal node $X \in \texttt{spNodes}(\mathcal{C})$, create and place two corresponding vertices, $\texttt{source}(X)$ and $\texttt{sink}(X)$ in $V(\mathcal{C})$. For every leaf node $u \in \texttt{primOps}(\mathcal{C})$, place the single node u in $V(\mathcal{C})$. For convenience in defining the edges of the computation DAG, for each $u \in \texttt{primOps}(\mathcal{C})$, we define $\texttt{source}(u) = \texttt{sink}(u) = u$.

   More formally, the vertices of the graph $V(\mathcal{C})$ are defined as follows:

   $$V(\mathcal{C}) = \texttt{primOps}(\mathcal{C}) \cup \left(\bigcup_{X \in \texttt{spNodes}(\mathcal{C})} \{\texttt{source}(X), \texttt{sink}(X)\} \right) \ .$$

2. Next, construct the edges $E(X)$ for each node X in the SP-tree $\mathcal{C}$ recursively. For the base case, if $X \in \texttt{primOps}(\mathcal{C})$, then define $E(X) = \emptyset$. For the inductive case, consider an $X \in \texttt{spNodes}(\mathcal{C})$ with $\texttt{children}(X) = \{Y_1, Y_2, \ldots, Y_k\}$. If X is an S-node, then

   $$\begin{aligned} E(X) = \left(\bigcup_{i=1}^{k} E(Y_i) \right) \ &\cup \ \{(\texttt{source}(X), \texttt{source}(Y_1)), (\texttt{sink}(Y_k), \texttt{sink}(X))\} \\ &\cup \left(\bigcup_{i=1}^{k-1} \{(\texttt{sink}(Y_i), \texttt{source}(Y_{i+1}))\} \right) \ . \end{aligned}$$

   If X is a P-node, then

   $$E(X) = \left(\bigcup_{i=1}^{k} E(Y_i) \right) \cup \left(\bigcup_{i=1}^{k} \{(\texttt{source}(X), \texttt{source}(Y_i)), (\texttt{sink}(Y_i), \texttt{sink}(X))\} \right) \ .$$

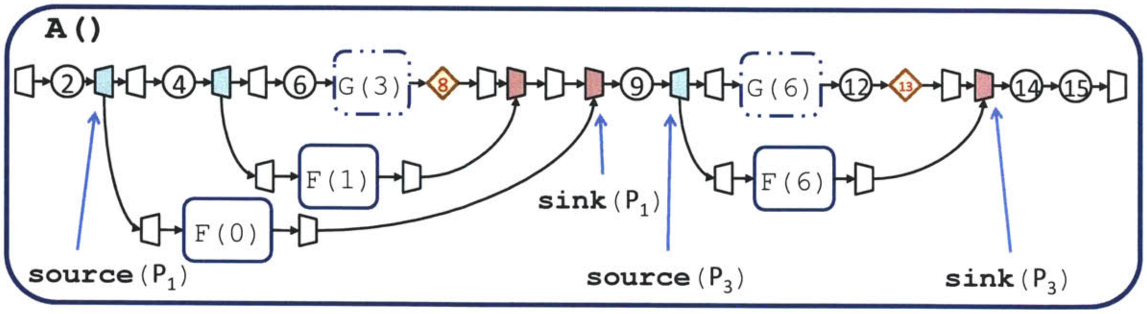


Figure A-6: The computation DAG $\mathcal{G}$ constructed from the series-parallel parse tree $\mathcal{C}$ in Figure A-5. To construct a canonical Cilk DAG (Figure A-2), the nodes between the first sync node (8) and $\texttt{sink}(\texttt{P}_1)$ must be contracted into a single node for the first sync block, and the nodes between the second sync node (13) and $\texttt{sink}(\texttt{P}_3)$ must be contracted into a single node for the second sync block.

Since there is a direct mapping from a computation tree $\mathcal{C}$ to its DAG $\mathcal{G}(\mathcal{C})$, we can also apply all the definitions from Section A.2 to computation trees.

To illustrate this conversion more concretely, Figure A-6 shows the equivalent computation DAG for the tree in Figure A-5. This DAG is equivalent to the canonical Cilk computation DAG Figure A-2, except with some extra source and sink nodes. Figure A-6 can be converted into Figure A-2 by contracting nodes after each sync.

Computation Trees

In this dissertation, we are primarily interested in working with a ***computation tree*** — a special kind of series-parallel parse tree that has a canonical structure which corresponds to the execution of a Cilk program.

In a computation tree $\mathcal{C}$, we partition $\texttt{sNodes}(\mathcal{C})$ into two disjoint sets — function nodes, denoted by $\texttt{functions}(\mathcal{C})$, and task nodes, denoted by $\texttt{tasks}(\mathcal{C})$. A ***function node*** $X \in \texttt{functions}(\mathcal{C})$ corresponds to a function invocation in the original Cilk program. A ***task node*** $Y \in \texttt{tasks}(\mathcal{C})$ represents a task associated with a $\texttt{spawn}$ or the continuation of a spawned function. A task node Y is always either $\texttt{root}(\mathcal{C})$ or the child of a P-node. We assume that $\texttt{root}(\mathcal{C}) \in \texttt{functions}(\mathcal{C})$, i.e., the root of $\mathcal{C}$ is a function node.

For any $B \in \texttt{nodes}(\mathcal{C})$, define the ***function parent*** of B as

$$\texttt{fParent}(B) = \begin{cases} \texttt{parent}(B) & \text{if } B \in \texttt{functions}(\mathcal{C}) \cup \{\texttt{root}(\mathcal{C})\} \\ \texttt{fParent}(\texttt{parent}(B)) & \text{if } B \notin \texttt{functions}(\mathcal{C}) \cup \{\texttt{root}(\mathcal{C})\} \end{cases} .$$

Conceptually, $\texttt{fParent}(B)$ is the closest proper ancestor of B in the tree which is a function node, or equivalently, the enclosing function that B is executed from. We can also define

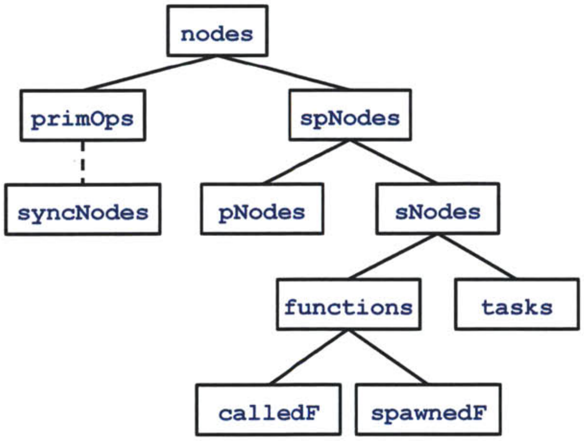


Figure A-7: Summary of the types of nodes in a computation tree $\mathcal{C}$. The children of a set $\mathtt{S}(\mathcal{C})$ in the tree connected by solid edges correspond to a partition of the $\mathtt{S}(\mathcal{C})$, whereas a child connected by a dashed edge is only a subset. For example, $\mathtt{nodes}(\mathcal{C})$ is partitioned into $\mathtt{primOps}(\mathcal{C})$ and $\mathtt{spNodes}(\mathcal{C})$, but we have only $\mathtt{syncNodes}(\mathcal{C}) \subseteq \mathtt{primOps}(\mathcal{C})$.

the ***function children*** of a node Z as

$$\mathtt{fChildren}(Z) = \{B \in \mathtt{nodes}(\mathcal{C}) \ : \ \mathtt{fParent}(B) = Z\} \ .$$

Finally, for any node B, define the ***function depth*** of B as

$$\mathtt{fDepth}(B) = |\mathtt{functions}(\mathcal{C}) \cap \mathtt{pAnces}(B)| \ .$$

Similarly, we define the ***task parent*** $\mathtt{tParent}(X)$ and the ***task children*** $\mathtt{tChildren}(Z)$ in an analogous fashion.

$$\mathtt{tParent}(B) = \begin{cases} \mathtt{parent}(B) & \text{if } B \in \mathtt{tasks}(\mathcal{C}) \cup \{\mathtt{root}(\mathcal{C})\} \\ \mathtt{tParent}(\mathtt{parent}(B)) & \text{if } B \notin \mathtt{tasks}(\mathcal{C}) \cup \{\mathtt{root}(\mathcal{C})\} \end{cases}$$
$$\mathtt{tChildren}(Z) = \{B \in \mathtt{nodes}(\mathcal{C}) \ : \ \mathtt{tParent}(B) = Z\} \ .$$

It is sometimes useful to refer apply a parent function repeatedly. For example, define $\mathtt{fParent}^k(B)$ as k applications of the $\mathtt{fParent}()$ function, i.e.,

$$\mathtt{fParent}^k(B) = \begin{cases} \mathtt{fParent}(\mathtt{fParent}^{k-1}(B)) & \text{if } k > 0 \ , \\ B & \text{if } k = 0 \text{ or } B = \text{NULL} \ . \end{cases}$$

The set of function nodes can be partitioned into two sets based on whether a function is invoked via a spawn or a call. More precisely, the set $\mathtt{functions}(\mathcal{C})$ can be partitioned into two sets — ***spawned functions*** $\mathtt{spawnedF}(\mathcal{C})$ and ***called functions*** $\mathtt{calledF}(\mathcal{C})$. Figure A-7 summarizes the different sets of nodes defined in a canonical computation tree.

In a canonical Cilk computation tree $\mathcal{C}$, nodes in $\mathtt{pNodes}(\mathcal{C}) \cup \mathtt{tasks}(\mathcal{C})$ can be also organized into sync blocks — groups of P-nodes and task nodes that are related by a sync within some function $F \in \texttt{functions}(C)$. Each sync block Q is terminated by a special kind of primitive operation — a ***sync node*** $y \in \texttt{syncNodes}(C)$, where $\texttt{syncNodes}(C) \subseteq \texttt{primOps}(C)$ denotes the set of all sync nodes in a computation C. More formally, functions and sync blocks satisfy the properties in Definition A.2.

Definition A.2. *For any function node $F \in \texttt{functions}(C)$, define the **sync blocks** of F as the set $\texttt{syncBlocks}(F) = \texttt{fChildren}(F) \cap (\texttt{pNodes}(C) \cup \texttt{tasks}(C))$. For every function F in a canonical Cilk computation tree, $\texttt{syncBlocks}(F)$ can be partitioned into n disjoint sets $Q_1, Q_2, \ldots, Q_n$, which satisfy the following properties:*

1. *Each set Q_i forms a **sync block** with $3k(i)$ elements, denoted as*

$$Q_i = \{\mathsf{P}_1, \mathsf{S}_1, \mathsf{S}_2, \mathsf{P}_2, \mathsf{S}_3, \mathsf{S}_4, \ldots, \mathsf{P}_{k(i)}, \mathsf{S}_{2k(i)-1}, \mathsf{S}_{2k(i)}\}.$$

   *Let $\texttt{root}(Q_i) = \mathsf{P}_1$ denote the **root** of the spine Q_i.*

2. *Each set Q_i has the following structure:*
   - *(a) $\mathsf{P}_j \in \texttt{pNodes}(C)$,*
   - *(b) $\{\mathsf{S}_{2j-1}, \mathsf{S}_{2j}\} \subseteq \texttt{tasks}(C)$,*
   - *(c) P_j has exactly two children: S_{2j-1} as a left child and S_{2j} as a right child.*
   - *(d) $\texttt{parent}(\mathsf{P}_1) = F$.*
   - *(e) For all j in the range $2 \le j \le k(i)$, we have $\texttt{parent}(\mathsf{P}_j) = \mathsf{S}_{2j-1}$ and P_j is the rightmost (last) child of S_{2j-1}.*
   - *(f) For all i in the range $1 \le j \le k(i)$, we have $\texttt{children}(\mathsf{S}_{2j-1}) = \{F_j\}$ for some function node $F_j \in \texttt{spawnedF}(C)$, i.e., each task node S_{2j-1} is the spawn of a function F_j.*

3. *For all $1 \le i < n$, $\texttt{root}(Q_i)$ is a child of F which is to the left of $\texttt{root}(Q_{i+1})$, i.e., the sync block spine Q_i executes serially before the spine Q_{i+1}.*

4. *The rightmost child of $\mathsf{S}_{2k(i)}$ is a special **sync node**, denoted by $\texttt{syncNode}(Q)$, satisfying $\texttt{syncNode}(Q) \in \texttt{syncNodes}(C)$.*

Figure A-8 gives a simple Cilk function F and its corresponding computation tree for F, highlighting the structure of the two sync blocks of F. By Definition A.2, every P-node P_i in a canonical computation tree C has exactly two task nodes S_{2i-1} and S_{2i} as children and all the spawn statements within a given sync block fall along a single spine.

Finally, it is useful to define the notion of a "frame size" for each function node F. Relating the computation tree C back to the execution of a Cilk program, each function node $F \in \texttt{functions}(C)$ corresponds to a stack frame which is active while that function F is executing, and which consumes some amount of space in the system. Thus, for all functions $F \in \texttt{functions}(C)$, define the ***frame size*** of F, denoted as $\texttt{frameSize}(F)$, as the size of this stack frame for F in the execution of the computation C. We use this definition later in Section A.5 when we discuss the stack-space usage of Cilk.

```
int F() {
  int x = 0;
  int y = 0;
  spawn G1();
  x += 1;
  spawn G2();
  x += 2;
  G3();
  sync;
  y += 1;
  spawn H1();
  y += 2;
  spawn H2();
  spawn H3();
  sync;
  return x + y;
}
```

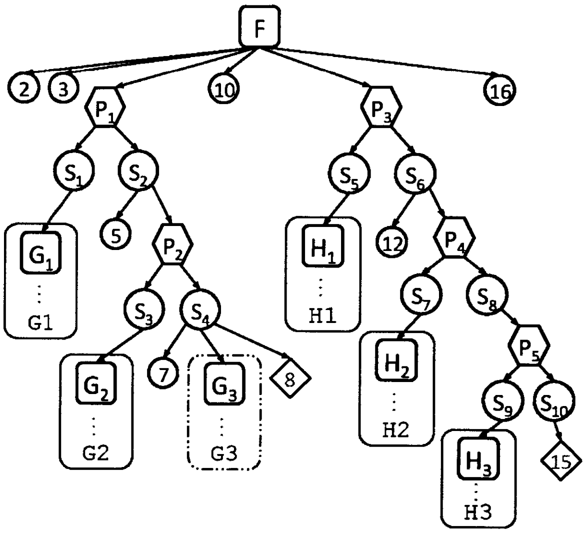


Figure A-8: A Cilk function F with two sync blocks (left), and its corresponding canonical computation tree C (right). The root of C is a function node F, which has two sync blocks. The first block is $Q_1 = \{P_1, S_1, S_2, P_2, S_3, S_4\}$, and the second block is $Q_2 = \{P_3, S_5, S_6, P_4, S_7, S_8, P_5, S_9, S_{10}\}$.

Dynamic Traversal of Computation Trees

To understand the properties of a Cilk program while it is executing, it is useful to model the program's execution as generating a computation tree C online, or equivalently, as a dynamic traversal of a computation tree C whose structure is known ahead of time. This section presents the ***Cilk computation-tree*** execution model, or ***CCT*** model for short. In a fashion similar to the computation DAG execution model described in Section A.2, the *CCT* model dynamically generates a computation tree C by taking a series of "steps," with each step executing an "instruction" that potentially changes the computation tree C.

To describe the *CCT* model more precisely, we first extend the definitions and notation for a computation tree C to allow C to change as ***steps*** execute. Generalizing the definition of computation-tree nodes $\texttt{nodes}(C)$, we define a dynamic set $\texttt{nodes}^{(t)}(C)$ as the set of tree nodes in C after taking step t. Conceptually, as a program executes, nodes are added, but never deleted from a computation tree C.

Similarly, for each of the subsets of $\texttt{nodes}(C)$ we defined earlier (e.g., $\texttt{sNodes}(C)$, $\texttt{functions}(C)$, etc.), we define a step-dependent set (i.e., $\texttt{sNodes}^{(t)}(C)$, $\texttt{functions}^{(t)}(C)$, etc.) representing a subset of computation-tree nodes after taking a particular step t. In general, these sets are monotonically increasing as the step count increases, i.e. for all t, $\texttt{nodes}^{(t)}(C) \subseteq \texttt{nodes}^{(t+1)}(C)$.

The *CCT* model also conceptually maintains a ***status field*** for each $X \in \texttt{nodes}^{(t)}(C)$ at each step t, denoted by $\texttt{status}^{(t)}(X)$, which stores runtime information about X. The exact value of this status field varies, depending on the kind of system we are attempting to model with a computation-tree traversal. For the execution of a normal Cilk program, the status of a tree node X will normally be one of RUNNING, QUEUED , or DONE. An node X is

RUNNING if X is currently executing or an ancestor of X is: currently executing. A node X is QUEUED if X is a task node that is sitting in the deque of some processor, waiting to be stolen and/or executed. Finally, a node X is DONE if the program has finished executing X (e.g., returned from a function or completed a task). Chapter 5, which considers transactional memory systems, also uses $\texttt{status}^{(t)}(X)$ to represent the status of function nodes $X \in \texttt{functions}^{(t)}(\mathcal{C})$ which are transactions.

We also define sets tracking which nodes have a given status on a particular time step. Define these sets as:

$$\begin{aligned}
\texttt{RUNNING}^{(t)}(\mathcal{C}) &= \left\{X \in \texttt{nodes}^{(t)}(\mathcal{C}) \ : \ \texttt{status}^{(t)}(X) = \texttt{RUNNING}\right\} \\
\texttt{QUEUED}^{(t)}(\mathcal{C}) &= \left\{X \in \texttt{nodes}^{(t)}(\mathcal{C}) \ : \ \texttt{status}^{(t)}(X) = \texttt{QUEUED}\right\} \\
\texttt{DONE}^{(t)}(\mathcal{C}) &= \left\{X \in \texttt{nodes}^{(t)}(\mathcal{C}) \ : \ \texttt{status}^{(t)}(X) = \texttt{DONE}\right\}
\end{aligned}$$

The sets $\texttt{RUNNING}^{(t)}(\mathcal{C})$ and $\texttt{QUEUED}^{(t)}(\mathcal{C})$ are *not* monotonically increasing, since nodes can be removed from these sets as their status switches to DONE.

To model a program execution, at the beginning of each step t, the *CCT* model maps each processor p to an ***producer*** S-node, denoted by $\texttt{producer}^{(t)}(p)$. Producer nodes must be RUNNING S-nodes, i.e., for any producer node $X = \texttt{producer}^{(t)}(p)$, we must have $X \in \texttt{sNodes}(\mathcal{C})$ with $\texttt{status}(X) = \texttt{RUNNING}$. It is also convenient to define the set of producer nodes over all workers p as

$$\texttt{producerNodes}^{(t)}(\mathcal{C}) = \bigcup_{\text{All } p} \texttt{producer}^{(t)}(p).$$

In the *CCT* model, a producer node X on step t may execute an ***instruction***. In particular, a processor p with producer node $X \in \texttt{sNodes}(\mathcal{C})$ can execute one of the instructions described in Figure A-9. Conceptually, the *CCT* model is analogous to the Cilk computation model described in Figure A-3, with the same set of instructions. The main difference is that for every worker, the *CCT* model maintains a producer node from the computation tree instead of an assigned node from the computation DAG.

From the definition of these instructions, one can show that after every step t, the *CCT* model generates a computation tree with the following structural properties.

Theorem A.2. *After every step t, the CCT model maintains the following invariants on a computation tree $\mathcal{C}$:*

1. *The set $\mathit{RUNNING}^{(t)}(\mathcal{C}) \cup \mathit{QUEUED}^{(t)}(\mathcal{C})$ corresponds to a connected subgraph of $\mathcal{C}$, which we refer to as the **active tree** of $\mathcal{C}$, denoted by $\texttt{vTree}^{(t)}(\mathcal{C})$.*
2. *We have $\texttt{producerNodes}^{(t)}(\mathcal{C}) \subseteq \texttt{leafSet}\left(\texttt{vTree}^{(t)}(\mathcal{C})\right)$, i.e., the producer nodes are leaves in the active tree.*
3. *We have $\mathit{QUEUED}^{(t)}(\mathcal{C}) \subseteq \texttt{leafSet}\left(\texttt{vTree}^{(t)}(\mathcal{C})\right)$, i.e., the queued nodes are leaves in the active tree.*

Instruction	Precondition	Updates
1. **primOp** Primitive op. u	$X = \texttt{producer}(p)$	$\textsc{NewNode}(u, \texttt{memOps}(C), X, \texttt{DONE})$
2. **call** Call of F	$X = \texttt{producer}(p)$	$\textsc{NewNode}(F, \texttt{calledF}(C), X, \texttt{RUNNING})$ $\texttt{producer}(p) \leftarrow F$
3. **cReturn** Return from F	$F = \texttt{producer}(p)$ $F \in \texttt{functions}(C)$	$\texttt{status}(F) \leftarrow \texttt{DONE}$ $\texttt{producer}(p) \leftarrow \texttt{parent}(F)$
4. **spawn** Spawn of F	$X = \texttt{producer}(p)$ p has deque q	$\textsc{NewNode}(\mathsf{P}, \texttt{pNodes}(C), X, \texttt{RUNNING})$ $\textsc{NewNode}(\mathsf{S}_1, \texttt{tasks}(C), \mathsf{P}, \texttt{RUNNING})$ $\textsc{NewNode}(\mathsf{S}_2, \texttt{tasks}(C), \mathsf{P}, \texttt{QUEUED})$ $\textsc{NewNode}(F, \texttt{spawnedF}(C), \mathsf{S}_1, \texttt{RUNNING})$ $\texttt{producer}(p) \leftarrow F$ push S_2 onto q
5. **sync** Sync in F	$X = \texttt{producer}(p)$ $X \in Q \subseteq \texttt{syncBlocks}(F)$ $F = \texttt{parent}(\texttt{root}(Q))$ $K = \texttt{leafSet}(Q)$	$\textsc{NewNode}(y, \texttt{syncNodes}(C), X, \texttt{DONE})$
(a) Finish Q	$\forall Z \in K - \{X\}$: $\texttt{status}(Z) = \texttt{DONE}$	for all $Y \in Q$: $\texttt{status}(Y) \leftarrow \texttt{DONE}$ $\texttt{producer}(p) \leftarrow F$
(b) Stall	$\exists Z \in K - \{X\}$: $\texttt{status}(Z) \neq \texttt{DONE}$	$\texttt{status}(X) \leftarrow \texttt{DONE}$ $\texttt{producer}(p) \leftarrow \text{NULL}$
6. **sReturn** Return from F to F'	$F = \texttt{producer}(p)$ $F \in \texttt{spawnedF}(C)$ $\mathsf{S}_1 = \texttt{parent}(F)$ $\texttt{children}(\mathsf{P}) = \{\mathsf{S}_1, \mathsf{S}_2\}$ $\mathsf{S}_1 \in Q \subseteq \texttt{syncBlocks}(F')$ $K = \texttt{leafSet}(Q)$	$\texttt{status}(F) \leftarrow \texttt{DONE}$
(a) Finish Q	$\forall Z \in K - \{\mathsf{S}_1\}$: $\texttt{status}(Z) = \texttt{DONE}$	for all $Y \in Q$: $\texttt{status}(Y) \leftarrow \texttt{DONE}$ $\texttt{producer}(p) \leftarrow F'$
(b) Stall	$\exists Z \in K - \{\mathsf{S}_1\}$: $\texttt{status}(Z) \neq \texttt{DONE}$ $\texttt{status}(\mathsf{S}_2) \neq \texttt{QUEUED}$	$\texttt{status}(\mathsf{S}_1) \leftarrow \texttt{DONE}$ $\texttt{producer}(p) \leftarrow \text{NULL}$
(c) Continue	$\texttt{status}(\mathsf{S}_2) = \texttt{QUEUED}$ p has deque q S_2 on bottom of q	$\texttt{status}(\mathsf{S}_1) \leftarrow \texttt{DONE}$ pop S_2 from q $\texttt{status}(\mathsf{S}_2) \leftarrow \texttt{RUNNING}$ $\texttt{producer}(p) \leftarrow \mathsf{S}_2$
7. **steal** Steal from p'	p' has deque q'	
(a) Successful	q' is not empty S on top of q' $\texttt{status}(\mathsf{S}) = \texttt{QUEUED}$	remove S from q' $\texttt{status}(\mathsf{S}) \leftarrow \texttt{RUNNING}$ $\texttt{producer}(p) \leftarrow \mathsf{S}$
(b) Unsuccessful	q' is empty	$\texttt{producer}(p) \leftarrow \text{NULL}$

Figure A-9: Instructions for the *CCT* model. On each step, a worker p executes the instruction whose precondition is satsified. The method $\textsc{NewNode}(B, \texttt{S}(C), X, \textsc{stat})$ adds a new computation-tree node B into the set $\texttt{S}(C)$, with parent X and initial status $\texttt{status}(B) = \textsc{stat}$. The `sync`, `sReturn`, and `steal` instructions can perform different updates, depending on which precondition is satisfied.

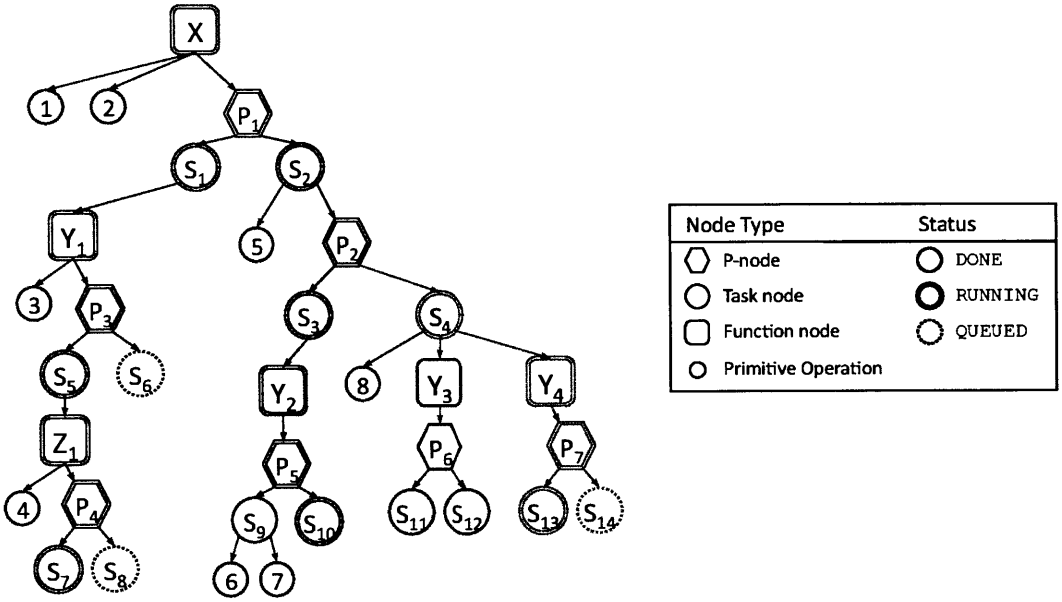


Figure A-10: A dynamic computation tree C on a step t in the CCT model. The tree C is being executed on 3 workers p_1,p_2, and p_3. The workers have producer nodes $\texttt{producer}^{(t)}(p_1) = \mathsf{S}_7$, $\texttt{producer}^{(t)}(p_2) = \mathsf{S}_{10}$, and $\texttt{producer}^{(t)}(p_3) = \mathsf{S}_{13}$. Worker p_1 has two QUEUED task nodes on its deque: S_6 at the head, and S_8 at the tail.

4. *$\mathsf{S} \in QUEUED^{(t)}(C)$ if and only if S is in the deque of some worker p on step t.*
5. *Let $Y_1, Y_2, \ldots, Y_n$ be the QUEUED nodes in the deque of worker p, from the head of the deque down to the tail, and let $\mathsf{P}_i = \texttt{parent}(Y_i)$. Then P_i is an ancestor of P_{i+1}.*
6. *$\left\{\texttt{parent}(\mathsf{S}) \ : \ \mathsf{S} \in QUEUED^{(t)}(C)\right\} \subseteq RUNNING^{(t)}(C) \cap pNodes(C)$, i.e., for any node S that is QUEUED, $\texttt{parent}(\mathsf{S})$ is a P-node with status of RUNNING.*

Proof. This result follows by induction on the different kinds of instructions that the CCT model can execute. □

Figure A-10 gives an example of a computation tree C generated by the CCT model. It is not difficult to verify that this tree C satisfies the invariants given in Theorem A.2.

Finally, scheduling in Cilk satisfies what is known as the "busy-leaves property" [27]. Roughly, this property states that at most P RUNNING functions can be leaves in the active tree. As discussed in Section A.5, this property enables Cilk to provide good theoretical bounds on space usage.

Theorem A.3. *The CCT model satisfies the* ***busy-leaves property****. More formally, consider a computation C executing on P workers. For all nodes $B \in \texttt{spNodes}^{(t)}(C)$, define the node $\texttt{blOwner}^{(t)}(B)$ as*

$$\texttt{blOwner}^{(t)}(B) = \begin{cases} B & \text{if } B \in \texttt{functions}^{(t)}(C) \\ B & \text{if } B \in \texttt{leafSet}\left(Q^{(t)}\right) \text{ for sync block } Q^{(t)} \\ \texttt{fParent}(B) & \text{if } B \in Q^{(t)} - \texttt{leafSet}\left(Q^{(t)}\right) \text{ for sync block } Q^{(t)} \end{cases} .$$

Then, for all nodes $B \in \texttt{vTree}^{(t)}(C)$, $\texttt{blOwner}^{(t)}(B) \in \texttt{ances}(\texttt{producer}^{(t)}(p))$ for some worker p.

Proof. Blumofe [27] proves this result for Cilk satisfies the busy-leaves property. We can prove this result for the *CCT* model by induction on the instructions the model can issue.

In the base case, the statement holds because we begin with $\texttt{root}(C)$ as the only member of $\texttt{vTree}(C)$ and the producer node for some worker p. For the inductive step, assume that on step $t-1$, for all nodes $B \in \texttt{vTree}^{(t-1)}(C)$, there exists a worker p such that $B \in \texttt{ances}(\texttt{producer}^{(t-1)}(C))$. Then, we can verify that each instruction preserves the inductive hypothesis for step t.

1. A `primOp` instruction maintains the inductive hypothesis because it does not modify the sets $\texttt{vTree}(C)$ or $\texttt{producerNodes}(C)$.

2. A `call` of a function F on a worker p changes the producer node of p from $X = \texttt{producer}^{(t-1)}(p)$ to $F = \texttt{producer}^{(t)}(p)$. The inductive hypothesis is maintained for all existing nodes in the active tree, i.e., for all $B \in \texttt{vTree}^{(t-1)}(C)$, because $\texttt{ances}(X) \subseteq \texttt{ances}(F)$. The inductive hypothesis is also maintained for the function F, the only node added to $\texttt{vTree}(C)$, because $F = \texttt{producer}^{(t)}(p)$ and we have $F = \texttt{blOwner}^{(t)}(F)$.

3. A `cReturn` from a function F on a worker p on step t changes the producer node from $\texttt{producer}^{(t-1)}(p) = F$ to $\texttt{producer}^{(t)}(p) = \texttt{parent}(F)$. To violate the inductive hypothesis, there must exist a $B \in \texttt{vTree}^{(t)}(C)$ such that $\texttt{blOwner}^{(t)}(B) \in \texttt{ances}(F)$, but $\texttt{blOwner}^{(t)}(B) \notin \texttt{ances}(\texttt{parent}(F))$. We have $\texttt{ances}(F) - \texttt{ances}(\texttt{parent}(F))$ equal to $\{F\}$ however. Also, $F \notin \texttt{vTree}^{(t)}(C)$ since the `cReturn` removes F from the active tree. Thus, no such node B can exist and the inductive hypothesis is preserved.

4. A `spawn` of a function F on a worker p creates three nodes with status `RUNNING`— a new node P as a child of $X = \texttt{producer}^{(t-1)}(p)$, S_1 as a child of P, and F as a child of S_1, and then sets $\texttt{producer}^{(t)}(p)$ to F.

   This case is analogous to the `call` instruction. The inductive hypothesis is maintained for all existing nodes $B \in \texttt{vTree}^{(t-1)}(C)$ since $\texttt{ances}(X) \subseteq \texttt{ances}(F)$. The inductive hypothesis is also satisfied for all new nodes B added to $\texttt{vTree}(C)$, since we have $\texttt{blOwner}^{(t)}(\mathsf{P}) = \texttt{blOwner}^{(t)}(X)$, $\texttt{blOwner}^{(t)}(\mathsf{S}_1) = \mathsf{S}_1$, $\texttt{blOwner}^{(t)}(F) = F$, and $\left\{\texttt{blOwner}^{(t)}(X), \mathsf{S}_1, F\right\} \subseteq \texttt{ances}(F)$.

5. Consider a `sync` instruction at a task node $X = \texttt{producer}^{(t)}(p)$, in a sync block Q in a function F. Two cases are possible:

   (a) If the `sync` finishes the sync block Q, then $\texttt{producer}(p)$ is set to F. Then, as with a `cReturn`, we can violate the inductive hypothesis only if there exists a node $B \in \texttt{vTree}^{(t)}(C)$ such that $\texttt{blOwner}(B) \in \texttt{ances}(X) - \texttt{ances}(F)$. But we know that after the `sync` completes, all nodes $B \in \text{spNodes}^{(t)}(F) - \{F\}$ have status `DONE`, and thus are removed from the active tree. Thus, the inductive hypothesis is preserved.

(b) If worker p stalls at the sync instruction, then $\texttt{producer}(p)$ is changed from X to NULL, and we know there exists some active task node Z in Q on step t, i.e., $Z \in \texttt{vTree}^{(t)}(C)$ and $Z \in \texttt{leafSet}\left(Q^{(t)}\right)$. By the inductive hypothesis, there exists some worker p' such that $\texttt{blOwner}(Z) \in \texttt{ances}(\texttt{producer}^{(t-1)}(p'))$. Furthermore, since Z is a leaf in the sync block Q, we know that $\texttt{blOwner}(Z) = Z$.

To show the inductive hypothesis is preserved, it suffices to show that for any node $B \in \texttt{vTree}^{(t-1)}(C)$ such that $\texttt{blOwner}^{(t-1)}(B) \in \texttt{ances}(X)$, we either have $B \notin \texttt{vTree}^{(t)}(C)$ or $\texttt{blOwner}^{(t)}(B) \in \texttt{ances}(\texttt{producer}^{(t)}(p'))$. In other words, we need to show that for any B which is an ancestor of X, either B is removed from the active tree in step t, or that worker p' preserves the invariant for B.

For any node $B \in \texttt{vTree}^{(t-1)}(C)$, let $Y_B = \texttt{blOwner}^{(t-1)}(B)$, and suppose that $Y_B \in \texttt{ances}(X)$. We know $F = \texttt{fParent}(X) = \texttt{parent}(\texttt{root}(Q))$. There are two possibilities for Y_B: either $Y_B \in \texttt{ances}(F)$ or $Y_B \in \texttt{ances}(X) - \texttt{ances}(F)$.

i. Suppose that $Y_B \in \texttt{ances}(F)$. We know $F \in \texttt{ances}(Z)$ since Z is in the sync block Q. As we argued earlier, we also have $Z \in \texttt{ances}(\texttt{producer}^{(t)}(p'))$. Thus, we can conclude that $Y_B \in \texttt{ances}(\texttt{producer}^{(t)}(p'))$, and hence the inductive hypothesis holds.

ii. If $Y_B \in \texttt{ances}(X) - \texttt{ances}(F)$, then by the canonical structure of sync blocks, either $Y_B \notin \texttt{leafSet}\left(Q^{(t)}\right)$ or $Y_B = X \in \texttt{leafSet}\left(Q^{(t)}\right)$. The first case is impossible, since $Y_B = \texttt{blOwner}^{(t-1)}(B)$ and by definition, we should have had $\texttt{blOwner}^{(t-1)}(B) = F$. In the second case, we have $Y_B = B$, and we know that the sync instruction sets $\texttt{status}(B)$ to DONE. Thus, since $B \notin \texttt{vTree}^{(t)}(C)$, the inductive hypothesis holds.

6. For an sReturn of a function F to F', there are three cases to consider.

(a) If the sReturn instruction finishes the sync block Q, then this case is similar to case (a) for the sync instruction; the inductive hypothesis is preserved because $\texttt{producer}(p)$ is set to F', and all other nodes in the subtree of F' are removed from the active tree.

(b) If the sReturn instruction stalls in the sync block Q, then as in case (b) of the sync instruction, $\texttt{producer}(p)$ is changed from F to NULL, and there exists some active task node Z in Q on step t, i.e., $Z \in \texttt{vTree}^{(t)}(C)$ and $Z \in \texttt{leafSet}\left(Q^{(t)}\right)$.

The proof is also analogous to the proof for case (b) of the sync. For any node $B \in \texttt{vTree}^{(t-1)}(C)$, let $Y_B = \texttt{blOwner}^{(t-1)}(B)$ and suppose that $Y_B \in \texttt{ances}(F)$. Let $F' = \texttt{fParent}(F) = \texttt{parent}(\texttt{root}(Q))$. There are two possibilities for Y_B: either $Y_B \in \texttt{ances}(F')$, or $Y_B \in \texttt{ances}(F) - \texttt{ances}(F')$.

i. Suppose that $Y_B \in \texttt{ances}(F')$. This case is the same as for the sync, and thus we conclude that $Y_B \in \texttt{ances}(\texttt{producer}^{(t)}(p'))$.

ii. If $Y_B \in \texttt{ances}(F) - \texttt{ances}(F')$, then by the canonical structure of sync blocks, either $Y_B \notin \texttt{leafSet}\left(Q^{(t)}\right)$, $Y_B = \texttt{parent}(F) \in \texttt{leafSet}\left(Q^{(t)}\right)$, or $Y_B = F$. The first case is impossible, since $Y_B = \texttt{blOwner}^{(t-1)}(B)$ and by definition, we should have had $\texttt{blOwner}^{(t-1)}(B) = F'$. In the second and third cases, we know that $Y_B = B$ and that the `sReturn` instruction sets $\texttt{status}(B)$ to `DONE`. Since $B \notin \texttt{vTree}^{(t)}(\mathcal{C})$, the inductive hypothesis holds.

(c) If the `sReturn` continues execution in the sync block, then the producer node changes from $\texttt{producer}^{(t-1)}(p) = F$ to $\texttt{producer}^{(t)}(p) = \mathsf{S}_2$. Also, we know $\texttt{parent}(\mathsf{S}_2) = \texttt{parent}(\texttt{parent}(F)) = \mathsf{P}$ is a common P-node. The only nodes B for which the inductive hypothesis could be violated are $B \in (\texttt{ances}(\mathsf{S}_2) - \texttt{ances}(F))$. Only two nodes fall into this category however, namely $B = F$ or $B = \texttt{parent}(F)$. The inductive hypothesis is preserved because both of these nodes are removed from $\texttt{vTree}(\mathcal{C})$ by the `sReturn` instruction.

7. The `steal` only changes $\texttt{producer}(p)$ from `NULL` to a task node S which is not `NULL`. Adding a node to $\texttt{producerNodes}^{(t)}(\mathcal{C})$ cannot violate the inductive hypothesis.

□

A.5 Stack-Space Usage in Cilk

This section uses the computation tree model from Section A.4 to state the stack-space bounds guaranteed by MIT Cilk [51] and Cilk++ [93].

To analyze stack-space usage in a computation, we require some additional definitions. On any step t, let $\texttt{vFunc}^{(t)}(\mathcal{C}) = \texttt{vTree}^{(t)}(\mathcal{C}) \cap \texttt{functions}(\mathcal{C})$ denote the set of ***active functions*** in the *CCT* model. Let $\texttt{stackSpace}^{(t)}(\mathcal{C})$ be

$$\texttt{stackSpace}^{(t)}(\mathcal{C}) = \sum_{F \in \texttt{vFunc}^{(t)}(\mathcal{C})} \texttt{frameSize}(F) ,$$

that is, the stack space in $\mathcal{C}$ in use on step t. Finally, define the ***stack-space usage*** for $\mathcal{C}$, denoted by $\mathcal{S}_P(\mathcal{C})$, as the maximum over all steps t of $\texttt{stackSpace}^{(t)}(\mathcal{C})$ assuming that $\mathcal{C}$ is executed using P processors. Note that $\mathcal{S}_1$ is the serial stack-space usage, i.e., the stack space used by a serial execution of the computation $\mathcal{C}$.

Theorem A.4. *For any computation $\mathcal{C}$ executed by the CCT model on P processors, we have $\mathcal{S}_P(\mathcal{C}) \leq P\mathcal{S}_1(\mathcal{C})$.*

Proof. This result is proved by Blumofe [27] for Cilk. By Theorem A.3, we know that every function $F \in \texttt{vFunc}^{(t)}(\mathcal{C})$ is an ancestor of a producer node $\texttt{producer}^{(t)}(p)$ for some worker p. More formally, let $A_F(t,p)$ be the set

$$A_F(t,p) = \texttt{ances}(\texttt{producer}^{(t)}(p)) \cap \texttt{vFunc}^{(t)}(\mathcal{C}) .$$

Then using Theorem A.3, we have

$$\texttt{vFunc}^{(t)}(\mathcal{C}) = \bigcup_{\text{all workers } p} A_F(t,p) .$$

Since some of the sets $A_F(t,p)$ can overlap, we have

$$\sum_{F \in \texttt{vFunc}^{(t)}(\mathcal{C})} \texttt{frameSize}(F) \leq \sum_{\text{all workers } p} \left(\sum_{F \in A_F(t,p)} \texttt{frameSize}(F) \right) .$$

The term on the left is $\texttt{stackSpace}^{(t)}(\mathcal{C})$, the stack-space usage on any step t. Choosing the step t^* where this quantity is a maximum, the left side becomes $\mathcal{S}_P(\mathcal{C})$. We know, however, for any worker and on any step, the sum of $\texttt{frameSize}(F)$ for all $F \in A_F(t,p)$ is at most $\mathcal{S}_1(\mathcal{C})$. Thus, summing over all workers p gives $\mathcal{S}_P(\mathcal{C}) \leq P\mathcal{S}_1(\mathcal{C})$. □

A.6 Chapter Notes

This section briefly discusses other work related to Cilk.

Cilk follows the "lazy task creation" strategy of Kranz, Halstead, and Mohr [82], where the worker suspends the parent when a child is spawned and begins work on the child. An alternative strategy is for the worker to continued working on the parent, and have thieves steal spawned children. Cilk-1 [28], TBB [110], and TPL [92] employ this strategy, in large part because creating function continuations requires compiler support. This strategy can require unbounded bookkeeping space to execute a dthreaded program, however, even when the program executes on a single processor.

The computation DAG model described in Section A.2 resembles model described in [38, Chapter 27]. The bound on completion time stated in Theorem A.1 was originally proved by Blumofe and Leiserson [30] using a delay-sequence argument. The analysis by Arora, Blumofe, and Plaxton [16] is a variant of the original analysis in [30] which instead uses a potential-function argument. Arora et al. [16] apply their analysis to a nonblocking work-stealing algorithm, and thus the model does not consider contention on locks on deques when stealing. To model this contention, one can apply the recycling-game analysis as described in [30], which contributes the additional $\lg P$ term in Theorem A.1.

The computation tree model described in Section A.4 is adapted from the work of Feng and Leiserson [45], which represents Cilk program executions using series-parallel trees in order to perform efficient race detection. The notion of a series-parallel graph commonly appears in the literature in the context of electrical networks (e.g., [43]). Algorithms for recognizing series-parallel DAGs were described by Valdes in [120], and subsequently in [121]. The series-parallel tree notation used in Section A.4 dates back to at least [120, 121]: the authors refer a tree with S-nodes and P-nodes as the ***binary decomposition tree*** for a series-parallel DAG.

Appendix B

Summary of Notation

This appendix summarizes the notation used in this dissertation.

Notation	Description	Page #
Δ	maximum degree, i.e., $\max_{A \in V_{\mathcal{D}}} (\texttt{inDeg}(A) + \texttt{outDeg}(A))$	34
Δ_i	maximum indegree, i.e., $\max_{A \in V_{\mathcal{D}}} \texttt{inDeg}(A)$	33
Δ_o	maximum outdegree, i.e., $\max_{A \in V_{\mathcal{D}}} \texttt{outDeg}(A)$	33
$\Phi(u)$	observer function	130
$\Phi(u)$	potential of a node u in $\mathcal{G}$	84, 228
$\psi_W(A)$	processor-steps spent in COMPUTEANDNOTIFY(A) for atomic decrements for notifying successors of A	30
$\psi_S(A,B)$	processor-steps spent in COMPUTEANDNOTIFY(A) for atomic decrement for edge (A,B)	31
$\tau_1(A)$	the region work of region A	77
$\tau_\infty(A)$	the region span of region A	77

Table B.1: Summary of Notation (Greek Alphabet).

Notation	Description	Page #
$\mathcal{C}$	series-parallel parse tree, computation tree	126, 228
$d(u)$	depth of a node u in $\mathcal{G}$	83, 227
$\mathcal{D}$	a task graph $\mathcal{D} = (V_{\mathcal{D}}, E_{\mathcal{D}})$	29
$\mathcal{D}$	module tree	191
$E_{\mathcal{D}}$	set of task graph edges	29
$E(\mathcal{G})$	edges of DAG $\mathcal{G}$	223
$\mathcal{G}$	computation DAG	29, 223
$\mathcal{G}$	a computation DAG for execution of $\mathcal{D}$	29
$\mathcal{J}$	set of all traces for CWSTM	165
$\mathcal{J}(\mathcal{C})$	trace tree	166
$\mathcal{M}$	set of all memory locations	127
M	number of edges in region graph	78
$\mathcal{N}$	set of all Xmodules	201
N	number of regions, i.e., $\lvert\{\texttt{regions}(\mathcal{C})\}\rvert$	78
$R(u,\ell)$	read predicate (u reads from ℓ)	127
$\mathcal{S}_P(A)$	stack space used by region A in a P-processor execution	91
$\mathcal{S}_P(\mathcal{C})$	maximum stack space for a execution of $\mathcal{C}$ on P processors	240
$\widetilde{\mathcal{S}}_1$	aggregate serial space usage $(\sum_{A\in\texttt{regions}(\mathcal{C})} \mathcal{S}_1(A))$	91
$s_{\mathcal{D}}$	source node of task graph $\mathcal{D}$	29
$T_P(\mathcal{D})$	time to execute $\mathcal{D}$ on P processors	33
$T_1(A)$	work of subDAG for region A	77
$T_1(\mathcal{D})$	work to execute task graph $\mathcal{D}$	33
$T_1(\mathcal{G})$	work of DAG $\mathcal{G}$, $\lvert V(\mathcal{G})\rvert$	225
$T_\infty(A)$	span of subDAG for region A	77
$T_\infty(\mathcal{D})$	span of task graph $\mathcal{D}$	33
$T_\infty(\mathcal{G})$	span (or critical path) of DAG $\mathcal{G}$	225
$\widetilde{T}_\infty$	aggregate region span $(\sum_{A\in\texttt{regions}(\mathcal{C})} \tau_\infty(A))$	78
$t_{\mathcal{D}}$	sink node of task graph $\mathcal{D}$	29
$V_{\mathcal{D}}$	set of task graph nodes	29
V_∞	number of nodes on longest path in $\mathcal{D}$ from $s_{\mathcal{D}}$ to $t_{\mathcal{D}}$	33
$V(\mathcal{G})$	nodes of DAG $\mathcal{G}$	223
$W(u,\ell)$	write predicate (u writes to ℓ)	127
$w(u)$	weight of a node u in $\mathcal{G}$	83, 228

Table B.2: Summary of Notation (English Alphabet)

Notation	Description	Page #
$\texttt{abortactions}(X)$	abort actions of a transaction X	204
$\texttt{aborted}(\mathcal{C})$	aborted transactions	128
$\texttt{ABORTED}^{(t)}(\mathcal{C})$	nodes with status ABORTED on step t	142
$\texttt{aContent}(X)$	aborted content of transaction X	205
$\texttt{ances}(B)$	ancestors of B in $\mathcal{C}$	127, 229
$\texttt{calledF}(\mathcal{C})$	called functions in $\mathcal{C}$	232
$\texttt{cContent}(X)$	closed content of transaction X	205
$\texttt{children}(B)$	ordered set of children of B	127, 229
$\texttt{committed}(\mathcal{C})$	committed transactions	128
$\texttt{COMMITTED}^{(t)}(\mathcal{C})$	nodes with status COMMITTED on step t	142
$\texttt{content}(X)$	content of a transaction X	128
$\texttt{desc}(B)$	descendants of B in $\mathcal{C}$	127, 229
$\texttt{DONE}^{(t)}(\mathcal{C})$	nodes with status DONE on step t	142, 235
$\texttt{fChildren}(F)$	function children of a function $F \in \texttt{functions}(\mathcal{C})$	232
$\texttt{fDepth}(B)$	function depth of a node B	232
$\texttt{fParent}(B)$	function parent of a node B	231
$\texttt{fParent}^{k}(B)$	k applications of $\texttt{fParent}(B)$	232
$\texttt{frameSize}(F)$	frame size of function $F \in \texttt{functions}(\mathcal{C})$	233
$\texttt{functions}(\mathcal{C})$	function nodes of $\mathcal{C}$	231
$\texttt{head}[U]$	head of a trace U	165
$\texttt{holders}(B)$	holders of a node B	128
$\texttt{inDeg}(A)$	$\lvert\texttt{ipred}(A)\rvert$, i.e., in-degree of task node $A \in V_{\mathcal{D}}$	29
$\texttt{inDeg}(u)$	in-degree of node $u \in V(\mathcal{G})$	223
$\texttt{ipred}(A)$	immediate predecessors of task $A \in V_{\mathcal{D}}$	29
$\texttt{ipred}(v)$	immediate predecessors of v	71, 223
$\texttt{ipred}(v)$	immediate predecessor of v if $\texttt{inDeg}(v) = 1$	71, 223
$\texttt{isucc}(A)$	immediate successors of task $A \in V_{\mathcal{D}}$	29
$\texttt{isucc}(u)$	immediate successors of u	71, 223
$\texttt{isucc}(u)$	immediate successor of u if $\texttt{outDeg}(u) = 1$	71, 223
$\texttt{lastReaders}(\ell)$	$\texttt{readers}(\ell) \cap \texttt{desc}(\texttt{leaf}(\texttt{writers}(\ell)))$	153
$\texttt{LCA}(B_1, B_2)$	least common ancestor of B_1 and B_2 in $\mathcal{C}$	127, 229
$\texttt{leaf}(B)$	the leaf of a set B when $\lvert\texttt{leafSet}(B)\rvert = 1$	230
$\texttt{leafSet}(B)$	the leaf set of a set $B \subseteq \texttt{nodes}(\mathcal{C})$	230
$\texttt{loops}(A)$	set of all pairs of paths in $\mathcal{D}$ from any node X to A	54

Table B.3: Notation in typewriter font, A through L.

Notation	Description	Page #
$\texttt{memOps}(\mathcal{C})$	memory operation nodes in $\mathcal{C}$	127
$\texttt{modAnces}(M)$	ancestors of M in $\mathcal{D}$	201
$\texttt{modDesc}(M)$	descendants of M in $\mathcal{D}$	201
$\texttt{modMemory}(M)$	memory locations owned by Xmodule M	201
$\texttt{modParent}(M)$	parent of M in $\mathcal{D}$	201
$\texttt{modR}(X)$	module readset for X	202
$\texttt{modW}(X)$	module writeset for X	202
$\texttt{modXactions}(M)$	transactions corresponding to Xmodule M	201
$\texttt{mTree}(\mathcal{C},M)$	the projection of $\mathcal{C}$ onto Xmodule M	207
$\texttt{nodes}(\mathcal{C})$	$\texttt{primOps}(\mathcal{C}) \cup \texttt{spNodes}(\mathcal{C})$	126, 229
$\texttt{oContent}(X)$	open content of transaction X	205
$\texttt{outDeg}(A)$	$\lvert\texttt{isucc}(A)\rvert$, i.e., out-degree of task node $A \in V_{\mathcal{D}}$	29
$\texttt{outDeg}(u)$	out-degree of node $u \in V(\mathcal{G})$	223
$\texttt{owner}(\ell)$	Xmodule owning location ℓ	201
$\texttt{pAnces}(B)$	proper ancestors of B in $\mathcal{C}$	127, 229
$\texttt{parent}(B)$	parent of B in $\mathcal{C}$	126, 229
$\texttt{paths}(A,B)$	set of paths in $\mathcal{D}$ from A to B	29
$\texttt{paths}(u,v)$	set of all paths from u to v in $\mathcal{G}(\mathcal{C})$	77
$\texttt{pDesc}(B)$	proper descendants B in $\mathcal{C}$	127, 229
$\texttt{PENDING}^{(t)}(\mathcal{C})$	nodes with status $\texttt{PENDING}$ on step t	142
$\texttt{PENDING\_ABORT}^{(t)}(\mathcal{C})$	nodes with status $\texttt{PENDING\_ABORT}$ on step t	142
$\texttt{pNodes}(\mathcal{C})$	P-nodes of $\mathcal{C}$	126, 229
$\texttt{primOps}(\mathcal{C})$	primitive operation nodes of $\mathcal{C}$	126, 229
$\texttt{producer}^{(t)}(p)$	producer node for worker p	235
$\texttt{producerNodes}^{(t)}(\mathcal{C})$	set of producer nodes for all workers	235
$\texttt{QUEUED}^{(t)}(\mathcal{C})$	nodes with status $\texttt{QUEUED}$	235

Table B.4: Notation in typewriter font, $\texttt{M}$ through $\texttt{Q}$.

Notation	Description	Page #
$\mathtt{R}(X)$	readset of transaction X	121
$\mathtt{R}^{(t)}(X)$	readset of X on step t	143
$\mathtt{readers}^{(t)}(\ell)$	readers of ℓ on step t	143
$\mathtt{regions}(\mathcal{C})$	set of all parallel regions in a computation $\mathcal{C}$	71
$\mathtt{rgStackSpace}^{(t)}(\mathcal{C})$	stack space used by A on step t	91
$\mathtt{rgVFunc}^{(t)}(A)$	active functions within a region A on step t	91
$\mathtt{rg\_path\_length}(z,A)$	length of path z counting nodes belonging to A	77
$\mathtt{rg\_owner}(X)$	region owner of a node $X \in \mathtt{nodes}(\mathcal{C})$	71
$\mathtt{rg\_owner}(u)$	region owner of a DAG node $u \in V(\mathtt{nodes}(\mathcal{C}))$	71
$\mathtt{rg\_pred}(v)$	region predecessor of a node v in $\mathcal{G}(\mathcal{C})$	72
$\mathtt{rg\_succ}(u)$	region successor of a node u in $\mathcal{G}(\mathcal{C})$	72
$\mathtt{root}(B)$	the root of a set B when $\lvert\mathtt{rootSet}(B)\rvert = 1$	230
$\mathtt{root}(\mathcal{C})$	root of $\mathcal{C}$	126, 229
$\mathtt{root}(Q)$	root of a sync block Q	233
$\mathtt{rootSet}(B)$	the root set of a set $B \subseteq \mathtt{nodes}(\mathcal{C})$	230
$\mathtt{RUNNING}^{(t)}(\mathcal{C})$	nodes with status $\mathtt{RUNNING}$ on step t	142, 235
$\mathtt{sink}(F)$	sink node of F	127, 224
$\mathtt{sNodes}(\mathcal{C})$	S-nodes of $\mathcal{C}$	126, 229
$\mathtt{source}(F)$	source node of F	127, 224
$\mathtt{spawnedF}(\mathcal{C})$	spawned functions in $\mathcal{C}$	232
$\mathtt{spNodes}(\mathcal{C})$	control nodes of $\mathcal{C}$	126, 229
$\mathtt{stackSpace}^{(t)}(\mathcal{C})$	stack space used by $\mathcal{C}$ on step t	240
$\mathtt{status}^{(t)}(X)$	status of node $X \in \mathtt{nodes}^{(t)}(\mathcal{C})$	141, 234
$\mathtt{syncBlocks}(F)$	$\mathtt{fChildren}(F) \cap (\mathtt{pNodes}(\mathcal{C}) \cup \mathtt{tasks}(\mathcal{C}))$ for a function F	233
$\mathtt{syncNode}(Q)$	sync node for a sync block Q	233
$\mathtt{syncNodes}(\mathcal{C})$	sync nodes in $\mathcal{C}$	233
$\mathtt{tasks}(\mathcal{C})$	task nodes of $\mathcal{C}$	126, 231
$\mathtt{tChildren}(X)$	task children of a task $X \in \mathtt{tasks}(\mathcal{C})$	232
$\mathtt{tParent}(B)$	task parent of a node B	166, 232
$\mathtt{tParent}(U)$	$\mathtt{tParent}(\mathtt{head}[U])$ for a trace U	166

Table B.5: Notation in typewriter font, $\mathtt{R}$ through $\mathtt{U}$.

Notation	Description	Page #
$\texttt{vFunc}^{(t)}(\mathcal{C})$	active functions, i.e., $\texttt{vTree}^{(t)}(\mathcal{C}) \cap \texttt{functions}^{(t)}(\mathcal{C})$	240
$\texttt{vTree}^{(t)}(\mathcal{C})$	active tree on a step t, i.e., $\texttt{RUNNING}^{(t)}(\mathcal{C}) \cup \texttt{QUEUED}^{(t)}(\mathcal{C})$	142, 235
$\texttt{W}(X)$	writeset of transaction X	121
$\texttt{W}^{(t)}(X)$	writeset of X on step t	143
$\texttt{world}$	world Xmodule, root of $\mathcal{D}$	191
$\texttt{writers}^{(t)}(\ell)$	writers of ℓ on step t	143
$\texttt{xactions}(\mathcal{C})$	set of transactions	127
$\texttt{xAnces}(B)$	$\texttt{ances}(B) \cap \texttt{xactions}(\mathcal{C})$	128
$\texttt{xDesc}(B)$	$\texttt{desc}(B) \cap \texttt{xactions}(\mathcal{C})$	128
$\texttt{xid}(M)$	id of an Xmodule M	191
$\texttt{xMod}(X)$	Xmodule for transaction X	201
$\texttt{xParent}(B)$	transactional parent of B	128
$\texttt{xLCA}(B_1,B_2)$	least-common ancestor transaction	128

Table B.6: Notation in typewriter font, $\texttt{V}$ through $\texttt{Z}$.

Notation	Description	Page #
$CN^{\mathcal{G}}(A)$	subgraph of computation DAG $\mathcal{G}$ for $\textsc{ComputeAndNotify}(A)$	29
$CN(A)$	maximum of $CN^{\mathcal{G}}(A)$ over all possible DAGs $\mathcal{G}$	29
$CN^{*}(A)$	computation DAG for $\textsc{ComputeAndNotify}^{*}(A)$, a modification of $\textsc{ComputeAndNotify}(A)$ that always makes all recursive calls	31
$com^{\mathcal{G}}(A)$	subgraph of $\mathcal{G}$ for $\textsc{Compute}(A)$	29
$IC^{\mathcal{G}}(A)$	subgraph of computation DAG $\mathcal{G}$ for $\textsc{InitAndCompute}(A)$	54
$IC(A)$	maximum of $IC^{\mathcal{G}}(A)$ over all possible DAGs $\mathcal{G}$	54
$init^{\mathcal{G}}(A)$	subgraph of $\mathcal{G}$ for $\textsc{Init}(A)$	54

Table B.7: Miscellaneous definitions for Nabbit from Chapter 2.

Notation	Description	Page #
$A.done$	flag for signaling completion of region A	72
$A.dqpool$	deque pool for a parallel region A	72
$A.psize$	number of workers assigned to pool $A.dqpool$	72
$A.valid[p]$	TRUE when p is assigned to A's pool	72
$dq(p,A)$	deque for worker p in region A	72
$p \rightarrow activeDQ$	the active deque of p (bottom of deque chain)	74
$q \rightarrow assigned$	assigned node of deque q	80
$q \rightarrow child$	child deque of q in deque chain	74
$q \rightarrow parent$	parent deque of q in deque chain	74
$q \rightarrow region$	parallel region for a deque q	74
u ***belongs to*** *F*	F owns u	224
F ***contains*** *u*	u along some path between $\mathtt{source}(F)$ and $\mathtt{sink}(F)$	224
F ***owns*** *u*	F is the most deeply nested function containing u	224
parallelism of $\mathcal{G}$	$T_1(\mathcal{G})/T_\infty(\mathcal{G})$	227
serial node *u*	$\mathtt{inDeg}(u) = 1$ and $\mathtt{outDeg}(u) = 1$	223
spawn node *u*	$\mathtt{inDeg}(u) = 1$ and $\mathtt{outDeg}(u) = 2$	223
sync node *u*	$\mathtt{inDeg}(u) > 1$ and $\mathtt{outDeg}(u) = 1$	223

Table B.8: Miscellaneous definitions for HELPER (Chapter 3) and computation DAGs.

Notation	Description	Page #
$\mathtt{A}^{(t)}(\mathcal{C})$	a set $\mathtt{A}(\mathcal{C})$ after completing step t $\mathtt{A}$ can be any generic set	141, 234
uHv	hidden relation (u is hidden from v)	130
$\mathtt{R}(X) \leftarrow \mathtt{R}(X) \cup \{(\ell, u)\}$	Overloaded union operator for $\mathtt{R}(X)$ and $\mathtt{W}(X)$	143

Table B.9: Miscellaneous definitions for transactional computations.

Appendix C

Prefix-Race Freedom of TCO and OAT

This appendix presents the details of the proofs for the TCO model from Section 5.6 and the OAT model from Section 7.4. To prove these results, we first argue that the TCO operational model preserves several structural invariants on transaction readsets and writesets. Next, we use these invariants to prove Theorem 5.6, that the execution order $\mathcal{S}$ generated by the TCO model is "sequentially consistent" according to Definition 5.10. Finally, we extend these results for the TCO model to the OAT model and prove Theorem 7.7, that the OAT model generates traces $(\mathcal{C},\Phi)$ which are prefix-race-free as according to Definition 5.15.

C.1 Invariants on Readsets and Writesets

For the TCO model, we can characterize when a transaction X has a pair (ℓ,u) in its readset or writeset. These invariants on readsets and writesets for the TCO model also apply to the OAT model. In the proof, for any particular memory location ℓ, we can effectively reduce the OAT model to the TCO model by ignoring all transactions for Xmodules that can never have ℓ in their readset or writeset because of ownership requirements.

First, we show that the readsets of transactions act as caches for pairs (ℓ,u) stored in writesets.

Theorem C.1. *On any step t, for any transaction $Y \in \mathtt{readers}^{(t)}(\ell)$, suppose that $(\ell,u) \in R^{(t)}(Y)$. Let $X = \mathtt{leaf}(\mathtt{xAnces}(Y) \cap \mathtt{writers}^{(t)}(\ell))$. Then we have $(\ell,u) \in W^{(t)}(X)$.*

Proof. The proof is by induction on the instructions issued by the TCO model.

In the base case, at the initial step $t = 0$, for all memory locations $\ell \in \mathcal{M}$, we start with $\mathtt{readers}^{(0)}(\ell) = \mathtt{writers}^{(0)}(\ell) = \{\mathtt{root}(\mathcal{C})\}$ and $\mathtt{R}(\mathtt{root}(\mathcal{C})) = \mathtt{W}(\mathtt{root}(\mathcal{C}))$. Since we have $X = Y = \mathtt{root}(\mathcal{C})$, Theorem C.1 is satisfied in the base case.

For the inductive step, consider the operation of the various instructions.

- The spawn, sync, and sReturn instructions preserve the invariant because they do not change transaction readsets or writesets.

- Consider a successful read instruction v, and let $Z = \mathtt{xParent}(v)$. The only transaction whose readset changes is Z; v adds a pair (ℓ,u) into $\mathtt{R}^{(t+1)}(Z)$. This value comes from the readset of a transaction $Y = \texttt{leaf}(\texttt{readers}^{(t)}(\ell) \cap \texttt{ances}(v))$, or equivalently, $Y = \texttt{leaf}(\texttt{readers}^{(t)}(\ell) \cap \texttt{ances}(Z))$. By induction, we know that the invariant is satisfied for Y, i.e., $(\ell, u) \in \texttt{W}^{(t)}(X)$, where

$$X = \texttt{leaf}(\texttt{xAnces}(Y) \cap \texttt{writers}^{(t)}(\ell)) \ .$$

Using these facts, we can show that the transaction

$$X' = \texttt{leaf}(\texttt{xAnces}(Z) \cap \texttt{writers}^{(t)}(\ell))$$

must satisfy $X' = X$, i.e., the invariant is satisfied for Z.

First, since $X' \in \texttt{writers}^{(t)}(\ell) \subseteq \texttt{readers}^{(t)}(\ell)$, we know that $X' \notin \texttt{pDesc}(Y)$, since that would contradict our definition of Y as the closest ancestor of Z that contains ℓ in its readset. Thus, we must have $X' \in \texttt{xAnces}(Y)$. Also, we must have $X' \notin \texttt{pDesc}(X)$ to avoid contradicting our definition of X as the closest ancestor of Y that contains ℓ in its writeset. Therefore, that leaves $X' \in \texttt{xAnces}(X)$. But since $X \in \texttt{writers}^{(t)}(\ell)$, by definition of X', we must also have $X' \in \texttt{desc}(X)$. Thus, we have $X' = X$, and the invariant must be satisfied.

- Consider a successful `write` instruction v, and let $Z = \texttt{xParent}(v)$. We can show that v cannot break the invariant without generating a conflict. Suppose for contradiction that Theorem C.1 is violated for some transaction Y with $(\ell, u) \in \texttt{R}^{(t+1)}(Y)$. In other words, if $X = \texttt{leaf}(\texttt{xAnces}(Y) \cap \texttt{writers}^{(t+1)}(\ell))$, then suppose that $(\ell, u) \notin \texttt{W}^{(t+1)}(X)$. We must also have $(\ell, u) \in \texttt{R}^{(t)}(Y)$. Otherwise, if we had $(\ell, u) \notin \texttt{R}^{(t)}(Y)$, then $X = Y = Z$, since Z is the only transaction whose readset changes by a `write` instruction.

  To cause a violation, the writeset of X must have changed between during step t; either $(\ell, u) \in \texttt{W}^{(t)}(X)$, or $\ell \notin \texttt{W}^{(t)}(X)$ but (ℓ, u) was in the writeset of some ancestor transaction of X. Thus, we must have $X = Z$, since Z is the only transaction whose readset or writeset changes on step t. Furthermore, we must have $X \in \texttt{pAnces}(Y)$, since otherwise, if $X = Y$, then the `write` instruction would have replaced (ℓ, u) with (ℓ, v) in $\texttt{R}^{(t+1)}(Y)$. We know $v \notin \texttt{ances}(Y)$ since $\texttt{xParent}(v) = Z = X$. Therefore, on step t, we should have a conflict between v and Y, since $W(v, \ell)$, $(\ell, u) \in \texttt{R}^{(t)}(Y)$, and $v \notin \texttt{ances}(Y)$.

- An `xbegin` instruction creates a new transaction X with $\texttt{R}(X) = \emptyset$ and $\texttt{W}(X) = \emptyset$. Thus the invariant is trivially satisfied.

- Suppose that on step t, a transaction Y issues an `xend` and commits. The only transaction X which has its readset or writeset change after the `xend` (i.e., for which we could have $\texttt{R}^{(t)}(X) \neq \texttt{R}^{(t-1)}(X)$ or $\texttt{W}^{(t)}(X) \neq \texttt{W}^{(t-1)}(X)$) is $X = \texttt{xParent}(Y)$. The `xend` merges $\texttt{R}(Y)$ and $\texttt{W}(Y)$ into $\texttt{R}(X)$ and $\texttt{W}(X)$, respectively. Using Theorem 5.4, it is straightforward to show that the invariant is preserved for X.

  For X, consider the cases for memory operations:

1. If $(\ell, v) \in \mathtt{W}(Y)$, then (ℓ, v) is added to both $\mathtt{R}(X)$ and $\mathtt{W}(X)$, and thus the invariant for ℓ on X is trivially satisfied.
2. If $(\ell, v) \in \mathtt{R}(Y)$, but $\ell \notin \mathtt{W}(Y)$, then if $\ell \in \mathtt{R}(X)$, then we must have already had $(\ell, v) \in \mathtt{R}(X)$, since by the inductive hypothesis, both Y and X would have the same pair in their readsets.
3. If $\ell \notin \mathtt{R}(Y)$, then any values for ℓ in $\mathtt{R}(X)$ or $\mathtt{W}(X)$ are unaffected, and thus the invariant is preserved.

For all other transactions $Z \in \mathtt{readers}^{(t)}(\ell)$ and $Z \neq X$, consider two cases, namely $Z \in \mathtt{pAnces}(X)$ and $Z \in \mathtt{pDesc}(X)$. If $Z \in \mathtt{pAnces}(X)$, then the invariant is preserved for Z because the writesets of all transactions in $\mathtt{xAnces}(Z)$ remain the same. Similarly, if $Z \in \mathtt{pDesc}(X)$, then we can also argue that no writeset for any transaction in $\mathtt{xAnces}(Z)$ changes. The xend can only change the writeset of X, and only if $(\ell, v) \in \mathtt{W}(Y)$. But in this case, since Y is neither an ancestor or a descendant of Z, we would have had a conflict between Y and Z.

- Consider an xabort by a transaction $Z \in \mathtt{xactions}^{(t)}(\mathcal{C})$. The only way an xabort of a transaction Z can affect the invariant is by removing Z from $\mathtt{readers}(\ell)$ and $\mathtt{writers}(\ell)$, since the readsets and writesets of all other transactions remain the same. Clearing $\mathtt{W}(Z)$ cannot break the invariant for any other transaction X because Z must be a leaf of $\mathtt{vTree}^{(t)}(\mathcal{C})$, i.e., to abort Z, all $X \in \mathtt{pDesc}(Z)$ must already have been finished.

□

Theorem C.2 characterizes when a transaction X can have a location ℓ in its writeset.

Theorem C.2. *On any step t, consider any $X \in \mathtt{vTree}^{(t)}(\mathcal{C}) \cap \mathtt{xactions}^{(t)}(\mathcal{C})$ and any memory location $\ell \in \mathcal{M}$. Let $S_\ell(t) = \left\{u \in \mathtt{memOps}^{(t)}(\mathcal{C}) \ : \ W(u, \ell)\right\}$. Exactly one of the following cases holds:*

1. *We have $\ell \notin \mathcal{W}^{(t)}(X)$ and $\mathtt{cContent}^{(t)}(X) \cap S_\ell(t) = \emptyset$.*
2. *There exists an $(\ell, u) \in \mathcal{W}^{(t)}(X)$, with u happening on step t_u, and two conditions are satisfied:*
   - *(a) $u \in \mathtt{cContent}^{(t)}(X) \cap S_\ell(t)$*
   - *(b) For any operation $v \in S_\ell(t)$ that happens on step t_v, with $t_u < t_v \leq t$, we have $v \in \mathtt{aContent}^{(t)}(X) \cup \mathtt{vContent}^{(t)}(X)$.*
3. *$X = \mathtt{root}(\mathcal{C})$, $(\ell, \bot) \in \mathcal{W}^{(t)}(X)$ and two conditions are satisfied:*
   - *(a) $\mathtt{cContent}^{(t)}(X) \cap S_\ell = \emptyset$.*
   - *(b) For all $v \in S_\ell(t)$, we have $v \in \mathtt{aContent}^{(t)}(X) \cup \mathtt{vContent}^{(t)}(X)$.*

Proof. This theorem can be proved by induction on the steps of the TCO model.

In the base case, at step $t = 0$, we begin with a computation tree $\mathcal{C}$ that has a single transaction $\mathtt{root}(\mathcal{C})$ with $(\ell, \bot) \in \mathtt{W}(\mathtt{root}(\mathcal{C}))$ for all $\ell \in \mathcal{M}$. Thus, on this step, all transactions $X \in \mathtt{xactions}(\mathcal{C})$ fall into Case 3.

For the inductive step, consider each instruction that the TCO model can issue, as described in Section 5.5.

The control-flow instructions (spawn, sync, sReturn) do not create or finish any transactions, nor do they change any transaction writesets. Thus, they do not affect the invariants in Theorem C.2. Similarly, a successful read does not affect the invariants because it only adds a new element (ℓ, u) into a readset of a transaction, but does not change any writesets.

Consider a successful write on step t that creates a memory operation u satisfying $W(u,\ell)$. Let $X = \texttt{xParent}(u)$. Then the write adds (ℓ, u) to $\texttt{W}(X)$. For all transactions $Z \in \texttt{xactions}^{(t)}(\mathcal{C})$, examine how u affects the invariants for Z.

1. Suppose that $Z = X$. Then, since write adds (ℓ, u) to $\texttt{W}^{(t)}(X)$, we need to check that Case 2 holds for X on step t. To check the first condition, we know that $u \in \texttt{cContent}^{(t)}(X)$ because $X = \texttt{xParent}(u)$. The second condition also holds trivially, because u happens on the current step t, and there are no other operations v such that $t_v > t_u$.

2. For any transaction $Z \neq X$ with $\ell \notin \texttt{W}^{(t)}(Z)$, we know by the inductive hypothesis and Case 1 that $\texttt{cContent}^{(t)}(Z) \cap S_\ell(t) = \emptyset$. After the step, we still have $\ell \notin \texttt{W}^{(t+1)}(Z)$ and $\texttt{cContent}^{(t+1)}(Z) \cap S_\ell(t+1) = \emptyset$, since u only changes the closed content set $\texttt{cContent}(X)$.

3. For any transaction $Z \neq X$ which has $(\ell, w) \in \texttt{W}^{(t)}(Z)$, we know that $Z \in \texttt{ances}(u)$. Otherwise, u would have caused a conflict with Z according to Definition 5.17.

   There are two subcases to consider: either $w \neq \perp$ or $w = \perp$ (which also implies $Z = \texttt{root}(\mathcal{C})$).

   - If $w \neq \perp$, then for Condition 2a, we know $w \in \texttt{cContent}(Z) \cap S_\ell$ before and after the step. Also, since X is issuing a write instruction, we must have $X \in \texttt{vTree}^{(t)}(\mathcal{C})$. Thus, u is added to $\texttt{vContent}^{(t+1)}(Z)$, and Condition 2b still holds.
   - If $w = \perp$, we have a similar subcase, except Z falls into Case 3 of Theorem C.2 instead of Case 2. Condition 3a is preserved because $Z \neq X$ and the write instruction does not change the set $\texttt{cContent}(Z)$. Also, Condition 3b is preserved because u is added to $\texttt{vContent}^{(t+1)}(Z)$.

Consider an xbegin that creates a transaction Z. Since Z begins with $\texttt{R}(Z) = \texttt{W}(Z) = \emptyset$, Z falls into Case 1, which is trivially satisfied because $\texttt{cContent}^{(t+1)}(Z) = \emptyset$.

Consider an xend that successfully commits a transaction Z. Let $Y = \texttt{xParent}(Z)$ be Z's parent transaction. Then, since the xend change $\texttt{status}(Z)$ from PENDING to COMMITTED, we know that

$$\texttt{cContent}^{(t+1)}(Y) = \texttt{cContent}^{(t)}(Y) \cup \texttt{cContent}^{(t)}(Z) ,$$

that is, the commit of Z merges its closed content into the closed content of its parent.

The writesets and content sets for all other transactions besides Y and Z are unchanged by the xend, and the commit conceptually clears writeset of Z since it is no longer active.

Thus, we only need to check that the xend preserves Theorem C.2 for Y. For any memory location ℓ, consider the possible cases for how the commit of Z can change $\texttt{W}(Y)$.

1. Suppose that $\ell \notin \texttt{W}^{(t)}(Z)$. By induction (Case 1), we know $\texttt{cContent}^{(t)}(Z) \cap S_\ell(t) = \emptyset$. We also know that the set $\texttt{cContent}(Y) \cap S_\ell$ is the same before and after the step. Thus, for Y, the step preserves either Case 1 or the first condition in Case 2 or Case 3.

   Since $\texttt{cContent}(Y) \cap S_\ell$ remains the same, the second condition of Case 2 or Case 3 is also preserved. The only way the xend instruction can contradict Condition 2b or Condition 3b is to remove a memory operation v from $\texttt{aContent}(Y) \cup \texttt{vContent}(Y)$. But any memory operation removed from $\texttt{aContent}(Y) \cup \texttt{vContent}(Y)$ would be added to the set $\texttt{cContent}(Y) \cap S_\ell$, which can not happen because this set remains the same.

2. Suppose that $(\ell, u) \in \texttt{W}^{(t)}(Z)$. After the commit of Z, we have $(\ell, u) \in \texttt{W}^{(t+1)}(Y)$, and we need to check Case 2 for Y.

   First, we can verify Condition 2a. By the inductive hypothesis (Case 2), we have $u \in \texttt{cContent}^{(t)}(Z)$, and thus after the xend, we have $u \in \texttt{cContent}^{(t+1)}(Y)$.

   Next, we can verify Condition 2b. By the inductive hypothesis, for all $v \in S_\ell(t)$ such that $t_v > t_u$, we have $v \in \texttt{aContent}^{(t)}(Z) \cup \texttt{vContent}^{(t)}(Z)$. When Z commits on step t, however, we must have $\texttt{vContent}^{(t)}(Z) = \emptyset$, since Z can only commit if all its nested transactions have completed. Thus any such v must be in $\texttt{aContent}^{(t)}(Z)$. But then, since $\texttt{aContent}^{(t)}(Z) \subseteq \texttt{aContent}^{(t)}(Y) = \texttt{aContent}^{(t+1)}(Y)$, v satisfies Condition 2b for Y.

Finally, the sigabort or xabort instructions preserve the invariants in Theorem C.2. The abort of a transaction Z conceptually invalidates the readset and writeset of Z and eliminates the need to check the invariants for Z in Theorem C.2. Also, an abort of a transaction Z only moves operations v from $\texttt{vContent}(X)$ to $\texttt{aContent}(X)$ for ancestor transactions $X \in \texttt{pAnces}(Z) \cap \texttt{xactions}(\mathcal{C})$. □

The intuition for Theorem C.2 lies mostly in Case 2. If at time t a pair (ℓ, u) is the writeset of a transaction X, then u is the last write to ℓ in X's subtree which is "committed with respect to" X. Any v which writes to ℓ after t_u (the step when u occurs) must belong to X's subtree, since otherwise, there will be a conflict. Furthermore, any v which happens after t_u must still be aborted or pending with respect to X (i.e., $v \in \texttt{aContent}^{(t)}(X) \cup \texttt{vContent}^{(t)}(X)$); otherwise, v should replace u in X's write set.

Case 1 states that the writeset of X does not contain a location ℓ if no memory operation in X's subtree commits ℓ to X. Case 3 of Theorem C.2 handles the special case of the root.

C.2 Proof of Sequential Consistency

Using the invariants in Section C.1, we can show that traces generated by the TCO model are sequentially consistent, i.e., $(\mathcal{C}, \Phi)$ satisfies Definition 5.10. Said differently, we can show that the observer function Φ is the transactional last writer function $\mathcal{X}_\mathcal{S}$ according to the sort order $\mathcal{S}$ generated by the TCO model.

Proof of Theorem 5.6. The first condition and second conditions of Definition 5.9 are true by construction, since the TCO model can only set $\Phi(v) = u$ if $u <_S v$, $W(u,\ell)$, and $R(v,\ell) \vee W(v,\ell)$.

To check the third condition, we require some setup. Suppose that at time t_v, v happens and the TCO model sets $\Phi(v) = u$. Let $A = \texttt{leaf}(\texttt{readers}^{(t_v)}(\ell) \cap \texttt{ances}(v))$. Since $\Phi(v) = u$, we have $(\ell,u) \in \texttt{R}^{(t_v)}(A)$. Let $Z = \texttt{leaf}(\texttt{xAnces}(A) \cap \texttt{writers}^{(t)}(\ell))$. By Theorem C.1, we know $(\ell,u) \in \texttt{W}^{(t_v)}(Z)$. By Theorem C.2, since $(\ell,u) \in \texttt{W}^{(t_v)}(Z)$, we know $u \in \texttt{cContent}^{(t_v)}(Z)$. Let $X = \texttt{xLCA}(u,v)$. We must have $Z \in \texttt{ances}(X)$, since otherwise, we could not have $\{u,v\} \subseteq \texttt{memOps}^{(t_v)}(Z)$. Since $u \in \texttt{cContent}^{(t_v)}(Z)$, we have $\neg(uHv)$, satisfying the third condition.

To check the fourth condition, suppose that at time t_v, the TCO model sets $\Phi(v) = u$. Assume for contradiction that there exists a w such that $W(w,\ell)$, and $u <_S w <_S v$. Then, since $\Phi(v) = u$, we know that $u \in \texttt{W}^{(t_v)}(X)$ for some transaction X. Therefore, by Theorem C.2 we have $w \in \texttt{memOps}^{(t_v)}(X)$, since $w \in \texttt{aContent}^{(t_v)}(X) \cup \texttt{vContent}^{(t_v)}(X)$. Let $Y = \texttt{xLCA}(w,v)$. Since $v,w \in \texttt{memOps}^{(t_v)}(X)$, we know that $X \in \texttt{ances}(Y)$. Consider the two cases for w:

1. Suppose that $w \in \texttt{aContent}^{(t_v)}(X)$. Then $w \in \texttt{cContent}^{(t_v)}(Y) \cup \texttt{aContent}^{(t_v)}(Y)$, since we know $X \in \texttt{ances}(Y)$. In fact, we can show that $w \in \texttt{aContent}^{(t_v)}(Y)$. Consider the two cases for X and Y:

   (a) Suppose that $Y = X$. Since we know $w \in \texttt{aContent}^{(t_v)}(X) \cup \texttt{vContent}^{(t_v)}(X)$ and $w \in \texttt{cContent}^{(t_v)}(Y) \cup \texttt{aContent}^{(t_v)}(Y)$, we have $w \in \texttt{aContent}^{(t_v)}(Y)$.

   (b) Suppose that $X \in \texttt{pAnces}(Y)$. Assume for contradiction that we have $w \in \texttt{cContent}^{(t_v)}(Y)$. Then by Theorem C.2, then there exists a $y \in \texttt{memOps}^{(t)}(Y)$ such that $(\ell,y) \in \texttt{W}^{(t_v)}(Y) \subseteq \texttt{R}^{(t_v)}(Y)$. This statement contradicts the fact that TCO model found (ℓ,u) from transaction X and set $\Phi(v) = u$, since a closer transaction Y had (ℓ,y) in its readset. Thus, we have $w \in \texttt{aContent}^{(t_v)}(Y)$.

   But then, since $w \in \texttt{aContent}^{(t_v)}(Y)$, we have wHv, which contradicts Condition 4 of Definition 5.9.

2. Suppose that $w \in \texttt{vContent}^{(t_v)}(X)$. Then $w \in \texttt{cContent}^{(t_v)}(Y) \cup \texttt{vContent}^{(t_v)}(Y)$. By the same logic as in the previous case, we can show $w \notin \texttt{cContent}^{(t_v)}(Y)$. Thus, $w \in \texttt{vContent}^{(t_v)}(Y)$.

   If $w \in \texttt{vContent}^{(t_v)}(Y)$, then by Definition 5.18, there must exist a transaction $Y' \in (\texttt{xDesc}(Y) - \{Y\})$ with $w \in \texttt{cContent}^{(t_v)}(Y')$. Then, by Theorem 5.5, there exists some transaction $Z \in \texttt{xDesc}(Y') \cap \texttt{vTree}^{(t_v)}(C)$ such that $\ell \in \texttt{W}^{(t_v)}(Z)$. In summary, since $W(w,\ell)$ and $w \in \texttt{vContent}^{(t_v)}(Y)$, we can find a transaction Z which is a proper descendant transaction of Y which has ℓ in its writeset at step t_v.

   There are two possibilities for Z: either $Z \in \texttt{ances}(v)$ or $Z \notin \texttt{ances}(v)$. If $Z \in \texttt{ances}(v)$, then since $Y \in \texttt{pAnces}(Z)$, we have a contradiction because the TCO model set $\Phi(v) = u$, getting u from Y, but Z had ℓ in its writeset and Z is a closer ancestor of v. If $Z \notin \texttt{ances}(v)$, then we have a conflict between v and Z on step t_v, contradicting the fact that v was a successful memory operation.

In both cases, we reach a contradiction. Thus, no such w can exist, and Condition 4 of Definition 5.9 must be satisfied. □

C.3 Proof for OAT Model

We can adapt the proof of sequential consistency for the TCO model and apply it to the OAT model. It turns out that all the necessary invariants for the TCO model (Theorems 5.5, C.1 and C.2) also hold for the OAT model if we modify the definition of content sets in Definitions 5.18 and 7.4 to account for memory locations that an open-nested transaction X commits directly to the root transaction.

Definition C.1. *On any step t, for any $X \in \mathtt{xactions}^{(t)}(C)$ and a memory operation $u \in \mathtt{memOps}^{(t)}(C)$, define the sets $\mathtt{cContent}^{(t)}(X)$, $\mathtt{aContent}^{(t)}(X)$, $\mathtt{oContent}^{(t)}(X)$, and $\mathtt{vContent}^{(t)}(X)$ according the $\mathtt{ContentType}(t,u,X)$ procedure:*

```
$\mathtt{ContentType}(t,u,X)$        // For any $u \in \mathtt{memOps}^{(t)}(t)$
$Z = \mathtt{xParent}(u)$
while $(Z \neq X)$
    if $Z \in \mathtt{vTree}^{(t)}(C)$ return $u \in \mathtt{vContent}^{(t)}(X)$
    if $Z \in \mathtt{ABORTED}^{(t)}(C)$ return $u \in \mathtt{aContent}^{(t)}(X)$
    if $(X = \mathtt{committer}(u))$ return $u \in \mathtt{oContent}^{(t)}(X)$
    $Z \leftarrow \mathtt{xParent}(Z)$
return $u \in \mathtt{cContent}^{(t)}(X)$
```

In addition to the closed content, aborted content, and active content sets, Definition C.1 adds an ***open content*** set $\mathtt{oContent}(X)$ for a transaction X.

With this generalized definition of content sets, we can prove Theorem 7.7.

Proof of Theorem 7.7. Intuitively, for a computation C and the execution order $\mathcal{S}$, when we are considering prefix races for a particular location ℓ, we can ignore all transactions X which can never have ℓ in $\mathtt{R}(X)$ or $\mathtt{W}(X)$. Then, the OAT model behaves exactly like the TCO model for all remaining transactions, and thus prefix-race freedom for the TCO model also implies the prefix-race freedom for the OAT model.

More formally, for a transaction X and any operation $u \in \mathtt{oContent}(X)$, u reads or writes a memory location ℓ that is committed to the root transaction by the ownership-aware commit mechanism before reaching X. By the ownership properties of OAT, we know that location ℓ can *never* be in the readset or writeset of X. Thus, the invariants on readsets and writesets from Theorems 5.5, C.1 and C.2 that we proved for the TCO model hold without any change for OAT. Thus, Theorem 5.6 also holds for OAT, and we can apply the same logic from the proof of Theorem 5.7 to prove Theorem 7.7. □

Bibliography

[1] 6.884. Lab 3: Stencil computing. Available from `http://courses.csail.mit.edu/6.884/spring10/labs/lab3.pdf`, 2010.

[2] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 26–37, June 2006.

[3] Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha. Nested parallelism in transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Salt Lake City, UT, USA, February 2008.

[4] Kunal Agrawal, I-Ting Angelina Lee, and Jim Sukha. Safe open-nested transactions through ownership. Technical Report MIT-CSAIL-TR-2008-038, Laboratory of Computer Science and Artificial Intelligence, Massachusetts Institute of Technology, June 2008. Available online at `http://supertech.csail.mit.edu/~angelee/safeTech.pdf`.

[5] Kunal Agrawal, I-Ting Angelina Lee, and Jim Sukha. Safe open-nested transactions through ownership. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Raleigh, NC, USA, February 2009.

[6] Kunal Agrawal, I-Ting Angelina Lee, and Jim Sukha. Brief announcement: Serial-parallel reciprocity in dynamic multithreaded languages. In *Proceedings of the 22nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 186–188, June 2010.

[7] Kunal Agrawal, Charles E. Leiserson, Yuxiong He, and Wen Jing Hsu. Adaptive work-stealing with parallelism feedback. *ACM Transactions on Computer Systems*, 26:7:1–7:32, September 2008.

[8] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Memory models for open-nested transactions. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, October 2006. In conjunction with *International Conference on Architectutal Support for Programming Languages and Operating Systems*.

[9] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Executing task graphs using work-stealing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2010.

[10] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Helper locks for fork-join parallel programming. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, January 2010.

[11] Emmanuel Agullo, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julie Langou, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Asim YarKhan. *PLASMA Users' Guide*. Version 2.0 edition, 2009. Available from `http://icl.cs.utk.edu/projectsfiles/plasma/pdf/users_guide.pdf`.

[12] Alfred V. Aho, Michael R. Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[13] Eric Allen, David Chase, Joe Hallett, Victor Luchango, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. *The Fortress Language Specification, Version 1.0.* ©Sun Microsystems, Inc., March 2008. `http://research.sun.com/projects/plrg/Publications/fortress.1.0.pdf`.

[14] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. *IEEE Micro*, 26(1):59–69, January 2006.

[15] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[16] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, Puerto Vallarta, Mexico, June 1998.

[17] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM TOPLAS*, 11(4):598–632, October 1989.

[18] Rosa M. Badia, José R. Herrero, Jesús Labarta, Josep M. Pérez, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation: Practice and Experience*, 21:2438–2456, December 2009.

[19] Woongki Baek, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. Implementing and evaluating nested parallel transactions in software transactional memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 253–262, New York, NY, USA, 2010. ACM.

[20] Woongki Baek, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. Making nested parallel transactions practical using lightweight hardware support. In *Proceedings of the 24th ACM International Conference on Supercomputing (ICS)*, pages 61–71, New York, NY, USA, 2010. ACM.

[21] Rajkishore Barik, Zoran Budimlic, Vincent Cavè, Sanjay Chatterjee, Yi Guo, David Peixotto, Raghavan Raman, Jun Shirako, Sağnak Taşirlar, Yonghong Yan, Yisheng Zhao, and Vivek Sarkar. The Habanero multicore software research project. In *Proceeding of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 735–736, New York, NY, USA, 2009. ACM.

[22] Greg Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 261–270. ACM Press, June 1993.

[23] João Barreto, Aleksandar Dragojević, Paulo Ferreira, Rachid Guerraoui, and Michal Kapalka. Leveraging parallel nesting in transactional memory. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 91–100, New York, NY, USA, 2010. ACM.

[24] M. A. Bender, R. Cole, E. Demaine, M. Farach-Colton, and J. Zito. Two simplified algorithms for maintaining order in a list. In *Proceedings of the European Symposium on Algorithms (ESA)*, pages 152–164, 2002.

[25] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 133–144, Barcelona, Spain, June 2004.

[26] Guy E. Blelloch and Margaret Reid-Miller. Pipelining with futures. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 249–259, New York, NY, USA, 1997. ACM.

[27] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1995. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-677.

[28] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[29] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[30] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[31] Robert D. Blumofe and Dionisios Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical Report, University of Texas at Austin, 1999.

[32] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), November 2006.

[33] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, New Orleans, Louisiana, January 2003.

[34] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. Technical Report 191, LAPACK Working Note, September 2007.

[35] Brian D. Carlstrom, Austen McDonald, Michael Carbin, Christos Kozyrakis, and Kunle Olukotun. Transactional collection classes. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, pages 56–67, New York, NY, USA, 2007. ACM Press.

[36] Aparna Chandramowlishwaran, Kathleen Knobe, and Richard Vuduc. Performance evaluation of Concurrent Collections on high-performance multicore computing systems. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Atlanta, GA, USA, April 2010.

[37] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 519–538, 2005.

[38] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[39] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.

[40] John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. Programming with exceptions in JCilk. *Science of Computer Programming (SCP)*, 63(2):147–171, December 2006.

[41] C.T. Davies. Recovery semantics for a DB/DC system. In *ACM National Conference*, pages 136–141, 1973.

[42] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the Symposium on Theory of Computing*, pages 365–372, New York City, May 1987.

[43] R. J. Duffin. Topology of series-parallel networks. *Journal of Mathematical Analysis and Applications*, 10:303–318, 1965.

[44] Kemal Ebcioglu, Vijay Saraswat, and Vivek Sarkar. X10: an experimental language for high productivity programming of scalable systems. In *Proceedings of the Second Workshop on Productivity and Performance in High-End Computing (PPHEC-05)*, February 2005. Held in conjunction with the *Eleventh Symposium on High Performance Computer Architecture*.

[45] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, Newport, Rhode Island, June 1997.

[46] David Ferry, Kunal Agrawal, and Jim Sukha. Implementing a data access centered design pattern. In *Workshop on Parallel Programming Patterns (ParaPLoP)*, May 2011.

[47] Jeremy T. Fineman. Provably good race detection that runs in parallel. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 2005.

[48] Matteo Frigo. The weakest reasonable memory model. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1998.

[49] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the Twenty-First Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, Calgary, Canada, August 2009.

[50] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, New York, October 17–19 1999.

[51] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[52] Matteo Frigo and Victor Luchangco. Computation-centric memory models. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 240–249, 1998.

[53] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, second edition, 2000.

[54] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34:12:1–12:25, May 2008.

[55] Jim Gray. The transaction concept: Virtues and limitations. In *Seventh International Conference of Very Large Data Bases*, pages 144–154, September 1981.

[56] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[57] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 9–17, Austin, Texas, August 1984.

[58] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, October 1985.

[59] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, page 102, Washington, DC, USA, 2004. IEEE Computer Society.

[60] Johnson M. Hart. *Windows System Programming*. Addison-Wesley, third edition, 2004.

[61] E. A. Hauck and B. A. Dent. Burroughs' B6500/B7500 stack mechanism. *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 245–251, 1968.

[62] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. Technical report, Brown University, July 2007. Also available at `http://www.cs.brown.edu/publications/techreports/reports/CS-07-08.html`.

[63] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, pages 207–216, New York, NY, USA, Feb 2008. ACM.

[64] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, New York, NY, USA, 2003. ACM Press.

[65] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

[66] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[67] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. Hopscotch hashing. In *DISC '08: Proceedings of the 22nd International Symposium on Distributed Computing*, pages 350–364, Berlin, Heidelberg, 2008. Springer-Verlag.

[68] Ralf Hoffmann, Matthias Korch, and Thomas Rauber. Performance evaluation of task pools based on hardware synchronization. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, page 44, Washington, DC, 2004. IEEE Computer Society.

[69] T.C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

[70] Institute of Electrical and Electronic Engineers. Information technology — Portable Operating System Interface (POSIX) — Part 1: System application program interface (API) [C language]. IEEE Standard 1003.1, 1996 Edition.

[71] Intel Corporation. *Intel® Threading Building Blocks*, March 2008. Available at `http://www.threadingbuildingblocks.org`.

[72] Intel Corporation. *Intel® Cilk++ SDK Programmer's Guide*, October 2009. Document Number: 322581-001US.

[73] Intel Corporation. *Intel® Cilk Plus Language Specification*, 2010. Document Number: 324396-001US. Available from `http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf`.

[74] Intel Corporation. *Intel® Math Kernel Library Reference Manual*, 2011. Document Number: 630813-041US. Available from `http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/index.htm`.

[75] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 151–160, New York, NY, USA, 1994. ACM.

[76] Theodore Johnson, Timothy A. Davis, and Steven M. Hadfield. A concurrent dynamic task graph. *Parallel Computing*, 22(2):327–333, 1996.

[77] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., second edition, 1988.

[79] C.W. Kessler and H. Seidl. Language support for synchronous parallel critical sections. In *Advances in Parallel and Distributed Computing, 1997. Proceedings*, pages 92 -99, March 1997.

[80] Kathleen Knobe. Ease of use with Concurrent Collections (CnC). In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism (HotPar)*, pages 17–17, Berkeley, CA, USA, 2009. USENIX Association.

[81] Matthias Korch and Thomas Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms. *Concurrency and Computation: Practice & Experience*, 16(1):1–47, 2003.

[82] David A. Kranz, Robert H. Halstead, Jr., and Eric Mohr. Mul-T: A high-performance parallel Lisp. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 81–90, Portland, Oregon, June 1989.

[83] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *POPL'81: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–218, New York, NY, USA, 1981. ACM Press.

[84] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, March 2006.

[85] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M. Badia. Scheduling linear algebra operations on multicore processors. Technical Report 213, LAPACK Working Note, February 2009.

[86] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[87] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[88] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[89] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5:308–323, September 1979.

[90] Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 36–43. ACM Press, 2000.

[91] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[92] Daan Leijen and Judd Hall. Optimize managed code for multi-core machines. *MSDN Magazine*, 2007. Available from `http://msdn.microsoft.com/magazine/`.

[93] Charles E. Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51(3):244–257, March 2010.

[94] Sung-Chae Lim, Joonseon Ahn, and Myoung Ho Kim. A concurrent B^{link}-tree algorithm using a cooperative locking protocol. In *Lecture Notes in Computer Science*, volume 2712, pages 253–260. Springer Berlin / Heidelberg, 2003.

[95] Malcolm Yoke Hean Low, Weiguo Liu, and Bertil Schmidt. A parallel BSP algorithm for irregular dynamic programming. In *7th International Symposium on Advanced Parallel Processing Technologies*, pages 151–160. Springer, 2007.

[96] Victor Luchangco. *Memory Consistency Models for High Performance Distributed Computing*. PhD thesis, MIT, 2001.

[97] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Proceedings of the Workshop of Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

[98] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2006.

[99] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, Feb 2006.

[100] J. Eliot B. Moss. Nested transactions and reliable distributed computing. In *SRDS*, pages 33–39, Pittsburgh, PA, July 1982.

[101] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, MA, USA, 1985.

[102] J. Eliot B Moss. Open nested transactions : Semantics and support. In *Proceedings of the Workshop on Memory Performance Issues (WMPI)*, Austin, Texas, Feb 2006.

[103] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, 63(2):186 – 201, 2006. Special issue on synchronization and concurrency in object-oriented languages.

[104] Rajeev Motwani, Steven Phillips, and Eric Torng. Non-clairvoyant scheduling. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 422–431, 1993.

[105] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, March 2007.

[106] OpenMP application program interface, version 3.0. Available from `http://www.openmp.org/mp-documents/spec30.pdf`, May 2008.

[107] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[108] R. Raman and D.S. Wise. Converting to and from dilated integers. *IEEE Transactions on Computers*, 57(4):567–573, April 2008.

[109] David P. Reed. *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[110] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[111] Michael L. Scott. Sequential specification of transactional memory semantics. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

[112] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[113] Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. Beyond nested parallelism: tight bounds on work-stealing overheads for parallel futures. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 91–100, New York, NY, USA, August 2009. ACM.

[114] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Boston, MA, third edition, 2000.

[115] Jim Sukha. Brief announcement: A lower bound for depth-restricted work stealing. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, August 2009.

[116] Texas Advanced Computing Center. *GotoBLAS2*, 2011. Available from `http://www.tacc.utexas.edu/tacc-projects/gotoblas2/`.

[117] I.L. Traiger. Trends in systems aspects of database management. In *International Conference on Databases*, pages 1–21. Wiley Heyden Ltd, 1983.

[118] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 212–222, New York, NY, USA, 1992. ACM.

[119] Jeffrey D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.

[120] Jacobo Valdes. *Parsing Flowcharts and Series-Parallel Graphs*. PhD thesis, Stanford University, December 1978. STAN-CS-78-682.

[121] Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. The recognition of series parallel digraphs. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–12, New York, NY, USA, 1979. ACM.

[122] Haris Volos, Adam Welc, Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, Xinmin Tian, and Ravi Narayanaswamy. NePalTM: design and implementation of nested parallelism for transactional memory systems. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 291–292, New York, NY, USA, 2009. ACM.

[123] David B. Wagner and Bradley G. Calder. Leapfrogging: a portable technique for implementing efficient futures. *SIGPLAN Notices*, 28(7):208–217, 1993.

[124] Gerhard Weikum. A theoretical foundation of multi-level concurrency control. In *Proceedings of the ACM SIGACT-SIGMOD symposium on Principles of Database systems (PODS)*, pages 31–43, New York, NY, USA, 1986. ACM Press.

[125] Jeannette M. Wing, Manuel Faehndrich, J. Gregory Morrisett, and Scott Nettles. Extensions to standard ML to support transactions. Technical Report CMU-CS-92-132, Carnegie Mellon, 1992.

[126] David S. Wise and Jeremy D. Frens. Morton-order matrices deserve compilers' support. Technical Report TR533, Indiana University, 1999.