Scheduling and Synchronization for Multicore Concurrency Platforms

by

Kunal Agrawal

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

PhD in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2009

Author .....
Department of Electrical Engineering and Computer Science
September 1, 2009

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

Accepted by ...
Terry P. Orlando
Chairman, Department Committee on Graduate Theses

Scheduling and Synchronization for Multicore Concurrency Platforms

by

Kunal Agrawal

Submitted to the Department of Electrical Engineering and Computer Science
on August 20, 2009, in partial fulfillment of the requirements
for the degree of
PhD in Computer Science and Engineering

Abstract

Developing correct and efficient parallel programs is difficult since programmers often have to manage low-level details like scheduling and synchronization explicitly. Recently, however, many hardware vendors have been shifting towards building multicore computers. This trend creates an enormous pressure to create ***concurrency platforms*** — platforms that provide an easier interface for parallel programming and enable ordinary programmers to write scalable, portable and efficient parallel programs. This thesis provides some provably-good practical solutions to problems that arise in the implementation of concurrency platforms, particularly in the domain of scheduling and synchronization.

The first part of this thesis describes work on scheduling of parallel programs written in ***dynamic multithreaded languages*** (such as Cilk, Hood etc.). These languages allow the programmer to express parallelism of their code in a natural manner, while an automatic scheduler in the concurrency platform is responsible for scheduling the program on the underlying parallel hardware. This thesis presents designs to increase the functionality of these concurrency platforms. The second part of the thesis presents work on transactional memory semantics and design. ***Transactional memory*** (TM), has been recently proposed as an alternative to locks. TM provides a transactional interface to memory. The programmers can specify their critical sections inside a transaction, and the TM concurrency platform guarantees that the region executes atomically. One of the purported advantages of TM over locks is that transactional code is *composable*. Most of the current TM concurrency platforms do not support full composability, however. This thesis addresses two of the composability problems in existing TM concurrency platforms.

Thesis Supervisor: Charles E. Leiserson
Title: Professor

Acknowledgments

First and foremost, I would like to express my deep gratitude to my wonderful adviser, Charles E. Leiserson. On being accepted to the MIT PhD program, when I first talked to Charles about working with him, he waved his hands around and talked to me about exciting ideas for about half an hour. I didn't understand 99% of what he said and was too awed to ask. However, I liked Charles and his ideas sounded grand. On the basis of this tenuous reasoning, I decided to work with him. It was the best decision I ever made. I often still don't understand what he says, though now I ask. His ideas still sound grand, though.

Charles' enthusiasm for research, teaching, and life in general, is boundless and infectious. I have enjoyed every moment of working with him and he has taught me a lot starting from how to come up with research ideas to how to get shapes to align in Powerpoint. As an adviser, he provides just the right mix of guidance and independence to allow me to come into my own as a researcher.

I would like to thank past and present members and visitors to the Supercomputing Technologies Group for their support and comments. In particular, I have had conversations about various topics both related and unrelated to research with Michael Bender, Sid Chatterjee, John Danaher, Jeremy Fineman, Zardosht Kasheff, Bradley Kuszmaul, Edya Ladan Mozes, Jelani Nelson, Tim Olsen, Gideon Stupp, Angelina Lee and Jim Sukha. Bradley has always been available with help with various issues such as which machine to buy, how to solve a particularly tricky implementation problem, etc. and his comments on presentations are invaluable. Michael Bender visited the group for a year, and has been a wonderful friend and mentor ever since. Yuxiong and I collaborated closely on my first project and I loved working with her. Jeremy Fineman, Angelina Lee and Jim Sukha have been friends and collaborators for all of my graduate career. Jim and I have come to understand each other's incomprehensible ramblings. I will miss walking over to his desk several times a day and talking about all of my half-baked ideas.

I would also like to thank Yves Robert and Anne Benoit from ENS-Lyon. We started an incredibly fruitful collaboration after talking for a couple of hours and my meetings with them are always amazingly productive. I had a wonderful time while visiting them at Lyon, and hope to go back, if only for the fondue.

I would like to express my gratitude towards all the professors I TA'd with: Manolis Kellis, Srini Devadas, Erik Demaine, Ron Rivest and Charles. I thoroughly enjoyed all of my TA experiences and learnt a lot about teaching from all of them. I am sure that I will be a better teacher because of them. In addition, I would like to thank Cynthia Skier for giving me the opportunity to teach at the MIT EECS Women's Technology Program. The summer I taught at WTP was the best summer I have ever had. My TA experiences and the WTP experience convinced me that I want teaching to be a part of my career.

I was fortunate to have great friends to spend time with. I'd like to thank Harr Chen, Pallavi Kaushik, Yuanzhen Li, Ali Mohammad, Artessa Saldivar-Sali, Lavanya Sharan, Neha Soni, Bill Thies, Katy Thorn. Special thanks to my roommates over the years who helped me remain sane. When life was difficult, as it invariably sometimes was, I knew that I could go home and talk to someone who would listen.

Finally, I am eternally grateful to my wonderful parents and my brother for loving me and supporting me. From my parents, I have always received encouragement without any pressure to succeed. My brother is also one of my best friends. Although I haven't always shown it over the last few years, I love them more than I can say.

Contents

List of Figures

Chapter 1

Introduction

Recently, due to the diminishing returns in uniprocessor performance, the hardware industry has shifted towards producing ***multicore chips***, where each chip contains multiple processing elements or ***cores***. The number of cores per chip has been increasing steadily (Figure 1.1), and most desktop and laptops now have more than one core. Since multicores are parallel machines, programmers must write parallel programs in order to utilize the full power of these machines.

Parallel programming is significantly more difficult than sequential programming, however. In order to get good performance from parallel programs, programmers have to carefully engineer their programs, often based on the particular characteristics of the target parallel machine. Programmers spend large amounts of time correcting and fine-tuning their programs. It is not incorrect to say that writing parallel code is often like writing assembly-level code. Therefore, parallel programming has been the exclusive domain of a small number of expert programmers. However, with the advent of multicores, it is more desirable than ever to simplify parallel programming so that ordinary programmers can write programs for multicore machines.

In order to write parallel programs using traditional methods such as pthreads, programmers have to often design their own scheduler and perform synchronization using fine-grained locks. Properly designed ***concurrency platforms*** can alleviate much of the programmers' burden, however. A concurrency platform is a software abstraction layer that coordinates, schedules and manages resources and provides an interface for programmers to write parallel programs. Figure 1.2 shows the abstract model of a parallel machine. A parallel machine may provide multiple concurrency platforms, each designed for a particular application domain. With the increasing adoption of multicore hardware, both the research community and the industry have realized that concurrency platforms are required to ensure that all programmers can write software that utilizes the full capability of these multicore computers. Various parallel programming languages and libraries such as MIT Cilk [BFJ+95], Cilkarts' Cilk++ [Art09], IBM's X10 [ESS05], Sun's Fortress [ACH+07], OpenMP [Boa08], and Intel's Thread Building Blocks [Rei07] are examples of concurrency platforms.

Ideally, a concurrency platform should free programmers from the drudgery of handling low-level implementation details of parallel programming, allowing them to concentrate on algorithm design. In addition, a concurrency platform should provide good performance. This thesis provides some provably good practical solutions to problems that arise in the implementation of concurrency platforms, primarily in the domain of scheduling and synchronization. In particular, this thesis presents work on improving concurrency platforms that provide the interface of "dynamic

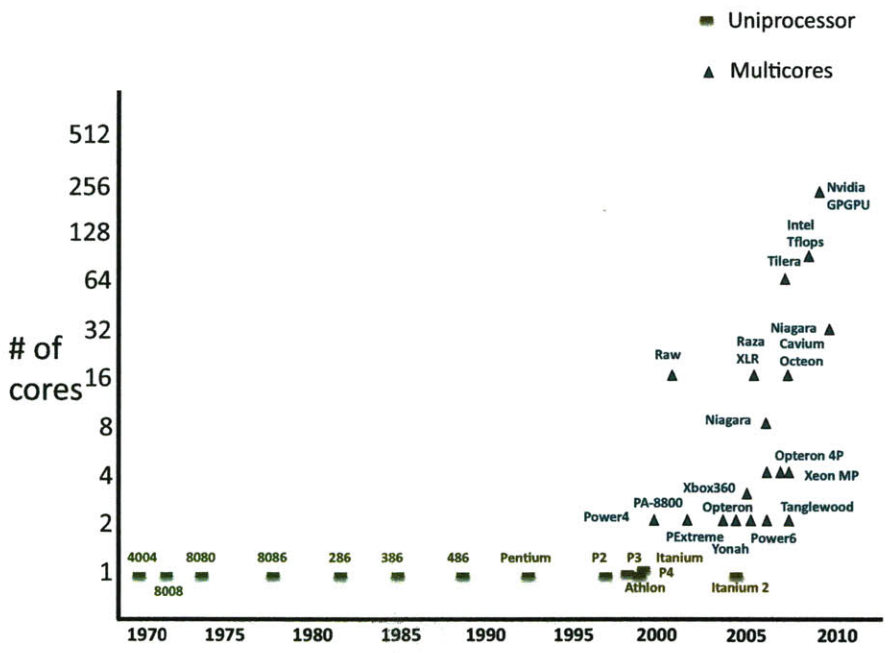

Figure 1.1: The squares are the uniprocessors and the triangles are multicores. Recently, the industry has been moving towards producing multicores. In addition, the number of cores in the multicores is increasing rapidly.

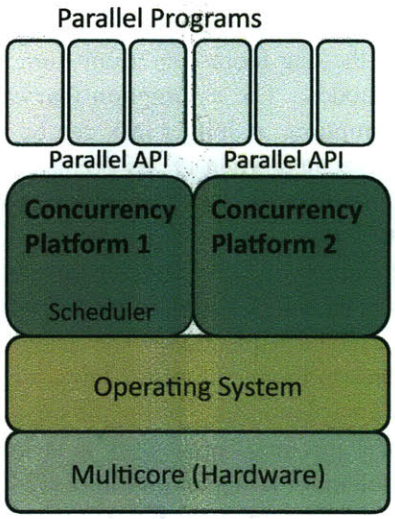

Figure 1.2: The logical view of a parallel machine with concurrency platforms. A machine may have more than one concurrency platform for different types of parallel applications. A concurrency platform provides a parallel API for parallel applications. In addition, it may provide services, such as automatic schedulers, in order to make parallel programming easier.

multithreading" and "transactional memory."

This chapter is organized as follows: Section 1.1 provides background and some definitions on dynamic multithreading and Section 1.2 provides background on transactional memory. Section 1.3 briefly describes the contributions and organization of this theses.

1.1 Dynamic Multithreading Background

Conventional concurrency platforms such as POSIX threads [Ins] or Java threads [GJSB00], provide a way to structure a large-scale computation into interacting ***persistent threads***. To obtain scalable performance using persistent threads can be difficult, however, because these programs are not adaptively parallel. If a programmer creates 10 threads, the program cannot effectively use 11 or more processors, even if those resources become available. Moreover, if the processor resources diminish to 9 or fewer, the multiplexing of threads onto the available resources can be dramatically inefficient [BP98a]. Thus, to obtain scalability, many pthreaded programs use the pthreads to implement a scheduler, such as a task-bag scheduler, complicating the direct expression of the programmer's desired algorithm.

On the other hand, new programming abstractions like ***dynamic multithreading*** — exemplified in languages like Cilk [BJK+95, FLR98], JCilk [DLL05], Hood [ABP98], NESL [BG96], Fortress [ACH+07], etc — provide concurrency platforms that allow the programmer to express the parallelism and the structure of the program in a more natural manner. The programmer is encouraged to express as much parallelism as she can, and the concurrency platform (including the compiler and the runtime system) is responsible for scheduling the application on the target machine.

A dynamic multithreaded jobs (also called a task-parallel job) are often modeled as dynamically unfolding directed acyclic graphs (dags) [BL98, BL99, Blu95, BG96, BGM99, FTYZ90, HS91, NB99, ST94] Each node in the dag represents a unit-time instruction, and an edge represents a serial dependence between nodes. The assumption that each node is a unit time instruction is primarily a simplifying assumption. A longer task can be modeled as a chain of short tasks. A node (task) is ***ready*** to be executed when all its predecessors have been executed. Scheduling of dynamic multithreaded jobs involves deciding which of the (potentially many) ready nodes to execute on the available processors.

We can define two parameters for such jobs. The ***work*** T_1 of the job corresponds to the total number of nodes in the dag. The work of a job is the amount of time it takes to execute this job on 1 processor, since the tasks execute sequentially. The second parameter is the ***span*** or ***critical-path length*** T_∞, which corresponds to the length of the longest chain of dependencies on the dag. The span of a job is equal to the completion time of the job on an infinite number of processors.

The completion time of a job on P processors is at least $\max T_1/P, T_\infty$. In this thesis, we consider two types of schedulers for dynamic multithreaded jobs. The first scheduler is the ***greedy scheduler*** [Gra69, Bre74], which completes a job in $T_1/P + T_\infty$ time. Therefore, a greedy scheduler's completion time is within a factor of 2 of the optimal completion time. The second scheduler we consider is a ***randomized work-stealing scheduler*** [BL98], which completes a job in $O(T_1/P + T_\infty)$ time (within constant factor of optimal). In spite of a worse completion time bound, work-stealing schedulers are often more desirable in practice than greedy scheduler since they have a better space bound and lower overheads. Many concurrency platforms such as Cilk, Cilk++, TBB, Fortress, X10, etc., employ randomized work-stealing schedulers.

1.2 Background on Transactional Memory

If two parallel threads access the same shared object concurrently, then these accesses must be properly synchronized in order to ensure correct performance. If these accesses are not properly synchronized, then the program is said to have a ***data race***. Data races lead to nondeterministic and unexpected program behavior. Conventionally, data races are prevented in parallel programs via ***mutual-exclusion locks***. Locks, however, introduce a host of difficulties. For example, to avoid deadlock when locking multiple objects, the locks must be acquired in a consistent linear order. This construct makes programming error-prone. In addition, every thread must grab a lock before accessing a shared object, regardless of whether another thread is actually accessing the object. Therefore, in the case where concurrent access to shared objects is rare, locks may introduce unnecessary overhead. Locks represent an example of "low-level" programming, since the programmer must manage the locks for all the shared objects in the system.

Transactional memory (TM) was proposed by Herlihy and Moss [HM] about 15 years ago as an alternative to locks. Recently, many software [HLM03, MSH+06, MSS05, SATH+06], hardware [HWC+04, AAK+05, MBM+06] and hybrid [DFL+06, KCJ+06] TM systems have been proposed and transactional memory has become an active area of research. The programmer simply declares the critical region to be `atomic`, and concurrency platform with a TM interface makes sure that all the instructions in the region appear to either have occurred atomically or not at all.

A TM system enforces atomicity by tracking the memory locations that each transaction in the system accesses. Most TM implementations maintain a transaction ***readset*** and ***writeset***, i.e., a

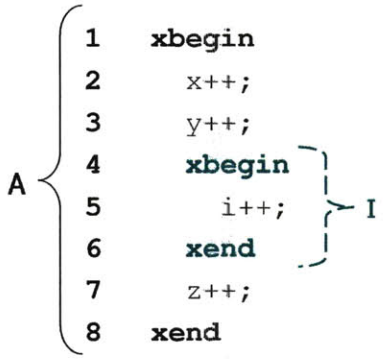

Figure 1.3: A code example where transaction I is nested inside A. The `xbegin` and `xend` delimiters mark the beginning and end of a transaction.

list of memory locations that a transaction has read from or written to, respectively. Typically, the system reports a conflict between two transactions A and B if both transactions access the same memory location and at least one of those accesses is a write. If A and B conflict, then TM aborts one of the transactions, rolls back any changes the aborted transaction made to global memory, and clears its readset and writeset. If a transaction completes without conflicting, then it is ***committed*** and its changes become visible.

Most of the work in this thesis concerns itself with ***nesting*** of transactions. Nested transactions arise when an outer transaction A in its body calls another transaction I. Figure 1.3 shows code for a transaction A within which another transaction I is nested. The database community has produced an extensive literature on nested transactions. Moss [Mos85] credits Davies [Dav73] with inventing nested transactions, and he credits Reed [Ree78] as providing the first implementation of what we now call closed transactions. Gray [Gra81] describes what we now call open transactions. The terms "open" and "closed" nesting were coined by Traiger [Tra83] in 1983.

The TM literature discusses three types of nesting: flat, closed, and open. The semantics and performance implications of each form of nesting can be understood through the example of Figure 1.3. If I is ***flat-nested*** inside A, then conceptually, A executes as if the code for I were inlined inside A. With flat-nesting, I's reads and writes are added directly to the readset and writeset of A. Thus, in Figure 1.3, if a concurrent transaction B tries to modify variable `i` while I is running, but before I has committed, then if I aborts, it also causes A to abort (since `i` conceptually belongs to the readset of A as well).

If I is ***closed-nested*** inside A (see, for example, [Mos85]), then conceptually, the operations of I only become part of A when I commits. In Figure 1.3, if B tries to modify `i` and causes I to abort, then the system only needs to abort and roll back I, but B need not abort A, because A has not accessed location `i` yet. Thus, closed nesting generally allows for a more efficient implementation compared with flat nesting, because closed nesting allows a nested transaction I to abort without forcibly aborting its parent transaction A, as with flat nesting. If I commits, however, I's readset and writeset are merged with A's readset and writeset. Thus, if B tries to modify `i` after I has committed but before A commits, the system may still abort A.

Finally, if I is ***open-nested*** inside A (see [WS92, MCC+06, MH05, Mos06]), then conceptually, the operations of I are not considered as part of A. When I commits, I's changes are made visible

to any other transaction B immediately, in the scheme of [MCC+06],[1] independent of whether A later commits or aborts. Thus, in Figure 1.3, B never aborts A, and B's access to variable `i` is never added to A's readset or writeset. Open nested transactions have fewer conflicts than closed-nested transactions, and allow for more concurrency. However, open-nested transactions break the strict serializability guarantees of TM.

1.3 Outline and Contributions

Dynamic multithreading and transactional memory provide useful abstractions for concurrency platforms. However, current concurrency platforms that provide these abstraction have certain limitations. This thesis addresses some of these limitations, and this section briefly describes both the limitations and the proposed solutions.

Adaptive scheduling

Most concurrency platforms for dynamic multithreaded languages do not address scheduling on ***multiprogrammed*** parallel machines — parallel machines with many parallel jobs running on them — very well. These concurrency platforms often use ***nonadaptive schedulers***, where the scheduler allots a fixed number of processors to the job for the job's lifetime. Nonadaptive schedulers are often ineffective for three reasons. First, when a job starts, the programmer must decide how many processors to allot to the job; this strategy burdens the programmers with analyzing the job's parallelism. Second, if the job's parallelism changes during execution, then nonadaptive schedulers' inflexibility can make them inefficient: jobs with small parallelism may waste processor cycles if they are allotted more processors than they can use. Third, new jobs may not be able to enter the system if most of the processors have already been allocated.

This thesis provides theoretical and empirical work on ***adaptive*** algorithms that continually adjust a job's allotment according to its parallelism and the parallelism of other jobs in the system. This work provides a basis for building concurrency platforms where programmers need not specify how many processors must be used to run their program, thereby unburdening programmers from analyzing the parallelism of the program.

For these multiprogrammed environments, my collaborators and I considered a two-level scheduling model: a system ***job scheduler*** is responsible for deciding how many processors each job is allotted, and each job has its own ***thread scheduler*** responsible for scheduling the individual threads (or tasks) of the job on the allotted processors effectively. Our scheduling algorithms use ***parallelism feedback*** — an estimate of the job's future parallelism — to request the "right" number of processors for the job. Periodically, the thread scheduler provides parallelism feedback to the job scheduler by requesting processors for the next interval. We designed thread schedulers that can provide provably good parallelism feedback for dynamic multithreaded programs and task parallel programs. In this thesis, we present two thread schedulers, **A-GREEDY** — based on greedy scheduling — and **A-STEAL** — based on work-stealing.

In order to ensure the robustness of these thread schedulers, we make three harsh assumptions in our theoretical analysis. First, we assume that the thread scheduler has no knowledge of the

[1] Several alternative policies for manipulating readsets and writesets were suggested in both [MH05, Mos06]. However, since then, the scheme described above has been used in most implementations, and therefore we do not discuss the alternatives here.

job's future parallelism. Second, we assume that the job's parallelism can change dramatically and frequently during execution and the job's future parallelism is not related to its past parallelism. Third, and most importantly, we assume that the scheduler operates under an omniscient adversarial environment. This environment knows all about the future and always makes decisions to hurt the thread scheduler's effectiveness. To analyze thread schedulers' performance under these adversarial conditions, we developed a new analytical technique called ***trim analysis***, which allows us to prove that our thread scheduler performs poorly on no more than a small number of time steps, exhibiting near-optimal behavior on the vast majority.

More precisely, suppose that a job has work T_1 and span T_∞. On a machine with P processors, both A-GREEDY and A-STEAL complete the job in expected $O(T_1/\widetilde{P} + T_\infty + L \lg P)$ time steps, where L is the length of a scheduling quantum and $\widetilde{P}$ denotes the $O(T_\infty + L \lg P)$-trimmed availability. This quantity is the average of the processor availability over all but the $O(T_\infty + L \lg P)$ time steps having the highest processor availability. When the job's parallelism dominates the trimmed availability, that is, $\widetilde{P} \ll T_1/T_\infty$, the job achieves nearly perfect linear speedup. Conversely, when the trimmed mean dominates the parallelism, the asymptotic running time of the job is nearly the length of its span, which is optimal.

We measured the performance of A-STEAL on a simulated multiprocessor system using synthetic workloads. For jobs with sufficient parallelism, our experiments confirm that A-STEAL provides almost perfect linear speedup across a variety of processor availability profiles. We compared A-STEAL with the ABP algorithm, an adaptive work-stealing thread scheduler developed by Arora, Blumofe, and Plaxton [ABP98] which does not employ parallelism feedback. On moderately to heavily loaded machines with large numbers of processors, A-STEAL typically completed jobs more than twice as quickly than ABP, despite being allotted the same or fewer processors on every step, while wasting only 10% of the processor cycles wasted by ABP.

This work was done in collaboration with Yuxiong He, Wenjing Hsu and Charles E. Leiserson, and appears in three conference papers [AHHL06, AHL06, AHL07] and one journal paper [AHHL08].

Dag evaluation library

Most languages (such as Cilk [BFJ+95], Cilk++ [Art09]) and libraries (such as Thread Building Blocks [Rei07]) that use work-stealing schedulers only allow programs which can be represented by fork-join (or series-parallel) dags. Some parallel programs may be most easily represented by non fork-join dags, however. These dags with arbitrary dependencies are tricky to express in fork-join languages such as Cilk++, since the programmer must maintain additional state to enforce dependencies that are not captured by the fork-join control flow of the program. My collaborators and I designed a Cilk++ library, called **DAGEVAL** , that allows programmers to evaluate arbitrary dags. In addition, the computation within each node of the dag can vary in load and can contain parallelism. Furthermore, DAGEVAL requires no modification to the Cilk++ runtime, making the techniques used applicable to other similar fork-join languages and libraries.

In DAGEVAL, we implement ***eager traversal*** of the dag. We prove that the eager traversal strategy is asymptotically optimal for dags with constant indegree and outdegree. That is, DAGEVAL completes a dag evaluation in $O(T_1/P + T_\infty)$ where T_1 is the work and T_∞ is the critical path of the computation. In addition, to evaluate the empirical performance of eager traversal, we implemented the dynamic program representing the Smith-Waterman algorithm [SW81],

an irregular dynamic program on a 2D grid which is used in computational biology. We find that when dag nodes are mapped to reasonably-sized blocks, our library's eager traversal exhibits low overhead and scales better than two other traversal strategies. In some cases, eager traversal even manages to outperform a divide-and-conquer implementation of the same dynamic program.

This work was done in collaboration with Charles Leiserson and Jim Sukha.

Helper locks in dynamic-multithreaded languages

As mentioned in Section 1.1, work-stealing schedulers guarantee linear speedup and many concurrency platforms employ randomized work-stealing schedulers. However, these guarantees on completion time of work-stealing schedulers do not hold if the programs contain synchronization such as locks. Therefore, these concurrency platforms can be inefficient when executing programs that contain synchronization. We introduce the notion of ***region helper locks*** for such concurrency platforms. A region helper lock protects a parallel subcomputation, called a ***parallel region***. Programmers can use region helper locks to express parallelism inside locked critical sections. Region helper locks allow programs with large critical sections to execute more efficiently, since they allow many processors to help complete these parallel regions. More specifically, say processor p tries to acquire a helper lock ℓ, and fails because some parallel region A protected by ℓ is already executing. Then, instead of blocking, p helps complete A.

We present a design of a work-stealing based concurrency platform, the ***parallel region lock (PRL)*** runtime, which can execute computations augmented with helper locks and parallel regions. The parallel region lock runtime allows *unbounded nesting* of parallel regions. We prove both completion time and space usage bounds for our PRL design. In particular, assuming that a program is "deadlock free," we show that the expected running time of a program on P processors is $O(W/P + \widetilde{T}_\infty + PN)$ where N is the number of parallel regions, W is the work of the computation and $\widetilde{T}_\infty$ is the "aggregate span", which is bounded by the sum of the spans of all the regions. For the space bounds, we prove that PRL completes a program using only $O(P\widetilde{S}_1)$ stack space, where $\widetilde{S}_1$ is the sum of serial stack space utilization over all parallel regions. Finally, we describe a prototype of helper locks implemented in MIT Cilk. To demonstrate the feasibility of implementing PRL, we use the prototype to implement a concurrent hash table with a resize operation protected by a region helper lock.

Transactional computation framework

Even though there has been a lot of research in TM recently, semantics of certain TM mechanisms are still poorly understood. Most TM designs are described using their implementation, and it is often difficult to unravel the semantic implications of design decision from these descriptions. This thesis presents a transactional computation framework inspired by Frigo and Luchanco's [FL98] computation centric framework. This framework allows us to define transactional semantics in an implementation independent manner. Our primary motivation for designing this framework was to precisely define the semantics of "open-nested transactions." We have found, however, that this framework is flexible enough to allows us to both perform *a posteriori* analysis of computations in order to understand the semantics of a TM design, and to define operational behavior of new TM designs.

Using this model, we define the traditional model of ***serializability*** and two new transactional-memory models, ***race freedom*** and ***prefix-race freedom***. We prove that these three memory models are equivalent for transactional-memory systems that support only closed nesting, as long as aborted transactions are "ignored." We prove that for systems that support open nesting, however, the models of serializability, race freedom, and prefix race freedom are distinct.

This work was done in collaboration with Charles E. Leiserson and Jim Sukha and appears in [ALS06].

Open nesting and ownership-aware nesting

Open nested transactions allow more concurrency in the TM system, but they also make transactions nonserializable. We use our transactional computation framework in order to show they support a much weaker memory model, called prefix-race freedom. As a consequence of this nonserializable behavior, if a transaction A has an open transaction B nested inside it, A may no longer see a transactionally consistent view of memory. This behavior can be avoided by carefully structuring A and B. This behavior also means, however, that if a function f containing an open transaction is called from within a transaction T, then T (and all other transactions that call T, and so on) must be aware of the fact that f has an open transaction so that T can be properly structured. Therefore, methods are no longer composable in a concurrency platform that supports open nesting.

The idea behind open nesting is to ignore "low-level" memory operations of an open-nested transaction when detecting conflicts for its parent transaction, and instead perform abstract concurrency control for the "high-level" operation that the nested transaction represents. Unfortunately, in a concurrency platform that supports open nesting, the TM runtime is unaware of the different levels of memory. Due to this, unconstrained use of open nesting leads to anomalous program behavior.

My collaborators and I designed an alternative called ***ownership-aware transactional memory*** which allows a more systematic and controlled form of open nesting. Ownership-aware transactional memory incorporates the notion of modules into the TM system and requires that transactions and data be associated with specific ***transactional modules*** or Xmodules. We propose a new ***ownership-aware commit mechanism***, a hybrid between an open-nested and closed-nested commit which commits a piece of data differently depending on which Xmodule owns the data. Moreover, we provide a set of precise constraints on interactions and sharing of data among the Xmodules based on familiar notions of abstraction. The ownership-aware commit mechanism and these restrictions on Xmodules allow us to prove that ownership-aware TM has clean memory-level semantics. In particular, it guarantees ***serializability by modules***, an adaptation of the definition of multilevel serializability from database systems. In addition, we describe how a programmer can specify Xmodules and ownership in a Java-like language. Our type system can enforce most of the constraints required by ownership-aware TM statically, and can enforce the remaining constraints dynamically. Finally, we prove that if transactions in the process of aborting obey restrictions on their memory footprint, then ownership-aware TM is free from ***semantic deadlock***. Therefore, ownership aware transactions offer the same concurrency as open-nesting, but provide it in a safe manner.

The research on semantics of open nested transactions was done jointly with Charles Leiserson and Jim Sukha[ALS06]. The research on ownership aware transactions was done jointly with

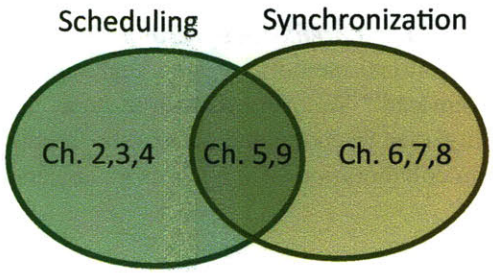

Figure 1.4: Thesis Organization.

Angelina I-Ting Lee and Jim Sukha [ALS09].

Nested parallelism in TM

Most implementations of transactional memory do not allow a transaction to call another method if the callee creates new parallelism. Therefore a function containing parallelism (and transactions to synchronize between its parallel threads) can not be called from within a transaction. Suppose that a function B calls a function A from within a transaction. With most current TM implementations, this code stops working correctly if a serial implementation of A is replaced by a correct parallel (and transactified) implementation of A with the same interface as the serial implementation. This behavior is clearly undesirable. Moreover, in programs using a dynamic-multithreaded language like Cilk, adding transactions in a natural manner generates code with parallelism inside transactions. It is unnatural to write code with no parallelism inside transactions when using such languages.

My collaborators and I designed a *provably efficient* software transactional memory system that allows parallelism inside transactions. This design is meant to add transactions to dynamic multithreaded languages that generate series-parallel programs. We designed XConflict, a data structure that facilitates conflict detection for a software transactional memory system which supports transactions with nested parallelism and unbounded nesting depth. For languages that use a Cilk-like work-stealing scheduler, XConflict answers concurrent conflict queries in $O(1)$ time and can be maintained efficiently. In particular, for a program with T_1 work and a span (or critical-path length) of T_∞, the running time on P processors of the program augmented with XConflict is only $O(T_1/P + PT_\infty)$.

Using XConflict, we describe CWSTM, a concurrency platform design for software transactional memory which supports transactions with nested parallelism and unbounded nesting depth of transactions. In the restricted case when no transactions abort and there are no concurrent readers, CWSTM executes a transactional computation on P processors also in time $O(T_1/p + PT_\infty)$. Although this bound holds only under rather optimistic assumptions, this result is the first theoretical performance bound on a TM system that supports transactions with nested parallelism which is independent of the maximum nesting depth of transactions.

This work was done in collaboration with Jeremy Fineman and Jim Sukha [AFS08].

Thesis organization

All of these contributions fall into the domain of scheduling and synchronization. Figure 1.4 shows the contributions of the thesis in pictorial form. Chapter 2 provides the algorithm and the theoretical results for adaptive scheduling, while Chapter 3 provides the experimental results. Chapter 4 provides work on dag evaluation. Chapter 5 provides work on parallel regions and helper locks. Chapter 6 explains the transactional computation framework, while Chapter 7 uses this framework to explain the semantics of open-nested transactions. Chapter 8 presents ownership-aware transactional memory, and Chapter 9 presents CWSTM.

Chapter 2

Adaptive Scheduling with Parallelism Feedback

In this chapter, we describe adaptive scheduling with parallelism feedback. The parallelism of programs written using task parallel or dynamic multithreaded languages such as Cilk [BFJ+95], Cilk++ [Art09], Intel's Thread Building Blocks [Rei07], etc. can change during the execution. Most concurrency platforms use scheduling algorithms that are ***nonadaptive***, however, where a fixed number of processors is allotted to the job for its lifetime. In a multiprogrammed environment, where a large number of jobs are executing on the same machine, nonadaptive schedulers are inflexible, and become inefficient when the parallelism of the jobs changes, or new jobs enter the system. This research concentrates on designing ***adaptive*** algorithms that use ***parallelism feedback***, an estimate of the job's future parallelism to request the "right" number of processors for the job so as to minimize both the *completion time* of the job, and the *waste* of processing resources by the job. These algorithms provide a basis for designing more effective concurrency platforms for task parallel and dynamic multithreaded languages. This chapter also describes the theoretical analysis of these algorithms. (Chapter 3 concentrates on the experimental evaluation.) In order to provide robust results, our theoretical analysis assumes harsh adversarial conditions and we introduce a new analytical technique called "trim analysis" in order to handle these conditions.

The chapter is organized as follows: Section 2.1 provides more background and motivation for adaptive scheduling; Section 2.2 describes our scheduling model and results, and introduces trim analysis; Sections 2.3 and 2.4 describe adaptive algorithms for greedy scheduling and work stealing respectively; Sections 2.5, 2.6, and 2.7 describe the theoretical analysis of these algorithms, and finally, Section 2.8 describes some related work.

2.1 Background and Motivation

The scheduling of a collection of parallel jobs onto a multiprocessor is an old and well-studied topic of research [TLW+94, BEGCS74, DGBL96, DD96, Gu95, MVZ93, Edm99, LV90, Squ95, TG89]. We consider so-called ***space-sharing*** [Fei97] for parallel jobs, where jobs occupy disjoint processor resources, as opposed to ***time-sharing***[Fei97], where different jobs may share the same processor resources at different times. Space-sharing schedulers can be implemented using a two-

This is joint work with Yuxiong He, Wenjing Hsu and Charles E. Leiserson [AHHL06, AHL07].

level strategy [Fei97]: a kernel-level ***job scheduler*** which allots processors to jobs, and a user-level ***thread scheduler*** which schedules the tasks belonging to a given job onto the allotted processors.

Most prior work on thread scheduling for multithreaded jobs deals with ***nonadaptive*** scheduling [BL99, BGM95, Gra69, Bre74, BG96, NB99], where the job scheduler allots a fixed number of processors to the job for its entire lifetime. For jobs whose parallelism is unknown in advance and which may change during execution, this strategy may waste processor cycles [Squ95], because a job with low parallelism may be allotted more processors than it can productively use. Moreover, in a multiprogrammed environment, nonadaptive scheduling may not allow a new job to start, because existing jobs may already be using most of the processors.

With ***adaptive*** scheduling [ABP98] (called "dynamic" scheduling in many papers), the job scheduler can change the number of processors allotted to a job while the job is executing. Thus, new jobs can enter the system, because the job scheduler can simply recruit processors from the already executing jobs and allot them to the new job. Unfortunately, as with a nonadaptive scheduler, this strategy may cause waste, because a job with low parallelism may still be allotted more processors than it can productively use.

Therefore, we require an adaptive scheduling strategy where the thread scheduler provides ***parallelism feedback*** to the job scheduler so that when a job cannot use many processors, those processors can be reallotted to jobs with ample need. Based on this parallelism feedback, the job scheduler can change the allotment of processors to each job according to the availability of processors in the current system environment and the job scheduler's administrative policy.

The question of how the job scheduler should partition the multiprocessor among the various jobs has been studied extensively [DGBL96, DD96, Gu95, MPT93, MVZ93, Edm99, LV90, RSSD95, RSD+94, YL01, MCN+00, ECBD03], but the administrative policy of the job scheduler is not the focus of this work. Instead, we study the problem of how the thread scheduler provides effective parallelism feedback to the job scheduler without knowing the future progress of the job, the future availability of processors, or the administrative priorities of the job scheduler.

Various researchers [DGBL96, DD96, Gu95, MVZ93, YL01] have used the notion of ***instantaneous parallelism***,[1] the number of processors the job can effectively use at the current moment, as the parallelism feedback to the job scheduler. Although using instantaneous parallelism for parallelism feedback is simple, it can cause gross misallocation of processor resources [Sen04]. For example, the parallelism of a job may change substantially during a scheduling quantum, alternating between parallel and serial phases. The sampling of instantaneous parallelism at a scheduling event between quanta may lead the thread scheduler to request either too many or too few processors depending on which phase is currently active, whereas the desirable request might be something in between. Consequently, the job may systematically waste processor cycles on the one hand or take too long to complete on the other.

Instead of using instantaneous parallelism, we use ***history-based strategies*** to provide parallelism feedback. Our strategy provides parallelism feedback to the job scheduler based on a single summary statistic and the job's behavior on the previous quantum. Even though our schedulers provide parallelism feedback using only the past behavior of the job, and we assume that the job's future parallelism can be completely uncorrelated with its history of parallelism, we prove shows that they schedule the job well with respect to both waste and completion time.

[1] These researchers actually use the general term "parallelism," but we prefer the more descriptive term.

2.2 Scheduling Model and Results

This section describes our scheduling model, and explains our results in detail. The scheduling model describes the mechanics of the communication between the thread scheduler and the job scheduler. In this section, we also explain our adversarial model of analysis and introduce a new analytical technique, called ***trim analysis*** in order to handle this adversarial model.

Our scheduling model is as follows: Each job has its own thread scheduler, and the thread scheduler operates in an online manner, oblivious to both the future characteristics of its job, and to the other jobs in the system. We assume that time is broken into a sequence of equal-size ***scheduling quanta*** $1, 2, \ldots$, each consisting of L time steps, and the job scheduler is free to reallocate processors between quanta. The thread scheduler operates as follows. Between quanta $q-1$ and q, it determines its job's ***desire*** d_q, which is the number of processors the job wants for quantum q. The thread scheduler provides the desire d_q to the job scheduler as its parallelism feedback. The job scheduler follows some processor allocation policy to determine the ***processor availability*** p_q — the number of processors to which the job is entitled for the quantum q. The number of processors the job receives for quantum q is the job's ***allotment*** $a_q = \min\{d_q, p_q\}$, the smaller of the job's desire and the processor availability. Once a job is allotted its processors, the allotment does not change during the quantum. Consequently, the thread scheduler must do a good job before a quantum of estimating how many processors it will need for all L time steps of the quantum, as well as do a good job of scheduling the job on the allotted processors.

This chapter describes two adaptive thread schedulers, A-GREEDY and A-STEAL, which provide parallelism feedback. A-GREEDY is a greedy thread scheduler suitable for centralized scheduling, where each job's thread scheduler can dispatch all the ready threads to the allotted processors in a centralized manner, such as the scheduling of data-parallel jobs. A-STEAL is a distributed thread scheduler, where each job is executed by decentralized work-stealing [BS81, Hal84, RSAU91, BL99]. These thread schedulers complete the job in near-optimal time while guaranteeing low waste.

Our theoretical analysis models the job scheduler as the thread scheduler's adversary, challenging the thread scheduler to be robust to the system environment and the job scheduler's administrative policies. As with completion time results for nonadaptive greedy and work-stealing schedulers', we provide results in terms of the work T_1 and the span T_∞ of the job. As mentioned in Chapter 1, nonadaptive greedy and randomized work-stealing schedulers guarantee that a job completes in $O(T_1/P + T_\infty)$ time, which is within constant factor of optimal. In an adaptive setting where the number of processors allotted to a job can change during execution, both $T_1/\overline{P}$ and T_∞ are lower bounds on the running time, where $\overline{P}$ is the mean of the processor availability during the computation. Therefore, one would like to prove the completion time of $O(T_1/\overline{P} + T_\infty)$ (which is asymptotically optimal and analogous to the nonadaptive results). In the worst case, however, an adversarial job scheduler can prevent any thread scheduler from providing good this completion time. For example, if the adversary chooses a huge number of processors for the job's processor availability just when the job has little instantaneous parallelism — the number of threads ready to run at a given moment — no adaptive scheduling algorithm can effectively utilize the available processors on that quantum. The adversary can therefore keep the availability low for all quanta with small parallelism and high for all quanta with low parallelism. With this availability profile, irrespective of parallelism feedback, the job will run slowly despite the mean parallelism being

high.[2]

We introduce a technique called ***trim analysis*** to analyze the time bound of adaptive thread schedulers under these adversarial conditions. From the field of statistics, trim analysis borrows the idea of ignoring a few "outliers." A ***trimmed mean***, for example, is calculated by discarding a certain number of lowest and highest values and then computing the mean of those that remain. For our purposes, it suffices to trim the availability from just the high side. For a given value R, we define the ***R-high-trimmed mean availability*** as the mean availability after ignoring the R steps with the highest availability, or just ***R-trimmed availability***, for short. A good thread scheduler should provide linear speedup with respect to an R-trimmed availability, where R is as small as possible.

We prove that both A-GREEDY and A-STEAL guarantee linear speedup with respect to $O(T_\infty + L \lg P)$-trimmed availability. [3] Specifically, consider a job with work T_1 and span T_∞ running on a machine with P processors and a scheduling quantum of length L. A-STEAL completes the job in expected $O(T_1/\widetilde{P} + T_\infty + L \lg P)$ time steps, where $\widetilde{P}$ denotes the $O(T_\infty + L \lg P)$-trimmed availability. Thus, the job achieves linear speed up with respect to the trimmed availability $\widetilde{P}$ when the parallelism T_1/T_∞ dominates $\widetilde{P}$. In addition, we prove that the total number of processor cycles wasted by the job is $O(T_1)$, representing at most a constant-factor overhead.

2.3 The Adaptive Greedy Algorithm

This section presents the adaptive greedy thread scheduler A-GREEDY. Before each quantum, A-GREEDY provides parallelism feedback to the job scheduler based on the job's history of utilization using a simple multiplicative-increase, multiplicative-decrease algorithm. A-GREEDY classifies quanta as "satisfied" versus "deprived" and "efficient" versus "inefficient." Of the four possibilities of classification, however, A-GREEDY only uses three: inefficient, efficient-and-satisfied, and efficient-and-deprived. Using this three-way classification and the job's desire for the previous quantum, it computes the desire for the next quantum. After the job scheduler allots a_q processors to the job for quantum q, A-GREEDY uses greedy scheduling [Gra69, Bre74] to schedule the ready tasks on the allotted processors.

To classify a quantum q as satisfied versus deprived, A-GREEDY compares the job's allotment a_q with its desire d_q. The quantum q is ***satisfied*** if $a_q = d_q$, that is, the job receives as many processors as A-GREEDY requested on its behalf from the job scheduler. Otherwise, if $a_q < d_q$, the quantum is ***deprived***, because the job did not receive as many processors as A-GREEDY requested.

Classifying a quantum as efficient versus inefficient is more complicated. We define the ***usage*** u_q of a quantum q as the amount of work completed by the job during the quantum, which is to say, the total number of unit-time tasks in the dag that were completed during the quantum. The maximum possible usage for a quantum q is La_q, where L is the length of quanta and a_q is the job's allotment for quantum q. A-GREEDY uses a ***utilization parameter*** δ, where $0 < \delta \leq 1$, as a threshold to differentiate between efficient and inefficient quanta. Typical values for δ might be 90–95%. We call a quantum q ***efficient*** if $u_q \geq \delta La_q$, that is, the usage is at least a δ fraction of the

[2]Using mean processor allotment instead of mean availability does not provide useful results. The trivial thread scheduler that always requests (and receives) 1 processor can achieve perfect linear speedup with respect to its mean allotment (which is 1) while wasting no processor cycles. By using a measure of availability, the thread scheduler must attempt to exploit parallelism.

[3]The constants in the bound are different for the two schedulers.

```
A-GREEDY(q, δ, ρ)
1  if q = 1
2     then d_q ← 1                          ▷ base case
3  elseif u_{q-1} < Lδa_{q-1}
4     then d_q ← d_{q-1}/ρ                  ▷ inefficient
5  elseif a_{q-1} = d_{q-1}
6     then d_q ← ρd_{q-1}                   ▷ efficient-and-satisfied
7  else d_q ← d_{q-1}                       ▷ efficient-and-deprived
8  Report desire d_q to the job scheduler.
9  Receive allotment a_q from the job scheduler.
10 Greedily schedule on a_q processors for L time steps.
```

Figure 2.1: Pseudocode for the adaptive greedy algorithm. A-GREEDY provides parallelism feedback to a job scheduler in the form of a desire for processors. Before quantum q, A-GREEDY uses the previous quantum's desire d_{q-1}, allotment a_{q-1}, and usage u_{q-1} to compute the current quantum's desire d_q based on the utilization parameter δ and the responsiveness parameter ρ.

maximum possible usage, in which case the job wastes few (at most $(1-\delta)La_q$) processor cycles. We call a quantum q ***inefficient*** otherwise.

A-GREEDY calculates the desire d_q of the current quantum q based on the previous desire d_{q-1} and the three-way classification of quantum $q-1$ as inefficient, efficient-and-satisfied, and efficient-and deprived. The initial desire is $d_1 = 1$. A-GREEDY uses a ***responsiveness parameter*** $\rho > 1$ to determine how quickly the scheduler responds to changes in parallelism. Typical values of ρ might range between 1.2 and 2.0. Figure 2.1 shows the pseudocode of A-GREEDY for one quantum. The algorithm takes as input the quantum q, the utilization parameter δ, and the responsiveness parameter ρ. Intuitively, it operates as follows:

- If quantum $q-1$ was inefficient, A-GREEDY overestimated the desire. In this case, A-GREEDY does not care whether the quantum is satisfied or deprived, and it decreases the desire (line 4) in quantum q.

- If quantum $q-1$ was efficient-and-satisfied, the job effectively utilized the processors that A-GREEDY requested on its behalf. Thus, A-GREEDY speculates that the job can use more processors and increases the desire (line 6) in quantum q.

- If quantum $q-1$ was efficient but deprived, the job used all the processors it was allotted, but A-GREEDY had requested more processors for the job than the job actually received from the job scheduler. Since A-GREEDY has no evidence whether the job could have used all the processors requested, it maintains the same desire (line 7) in quantum q.

Remarkably, this simple algorithm provides strong guarantees on waste and performance.

Greedy Scheduling

After the thread scheduler is allotted a_q processors for quantum q, it uses greedy scheduling to schedule the ready tasks on processors. Greedy scheduling operates as follows: At any time step, if more than a_q tasks are ready, then schedule any a_q of them. If at most a_q tasks are ready, then schedule all of them. Therefore greedy scheduling is a centralized scheduler since the thread scheduler must be aware of all the job's ready tasks.

2.4 Adaptive Work Stealing

This section presents the adaptive work-stealing thread scheduler A-STEAL. As with A-GREEDY, before the start of a quantum, A-STEAL estimates processor desire based on the job's history of utilization to provide parallelism feedback to the job scheduler. Instead of greedy scheduling, however, A-STEAL uses ***randomized work stealing*** [BL99, ABP98, MKHJ90] to schedule the job's ready tasks on the allotted processors. Unlike greedy scheduling, work stealing is a distributed strategy and therefore has lower synchronization overheads.

A-STEAL can use any provably good work-stealing algorithm, such as that of Blumofe and Leiserson [BL99] or the nonblocking one presented by Arora, Blumofe, and Plaxton [ABP98].[4] In these work-stealing thread schedulers, every processor allotted to the job maintains a double-ended queue, or ***deque***, of ready threads for the job. When the current thread spawns a new thread, the processor pushes the continuation of the current thread onto the top of the deque and begins working on the new thread. When the current thread completes or blocks, the processor pops the topmost thread off the deque to work on. If the deque of a processor is empty, however, the processor becomes a ***thief***, randomly picking a ***victim*** processor and ***stealing*** work from the bottom of the victim's deque. If the victim has no available work, then the steal is ***unsuccessful***, and the thief continues to steal at random from the other processors until it is ***successful*** and finds work. At all the time, every processor is either working or stealing.

Making work-stealing adaptive

This work-stealing algorithm must be modified to deal with dynamic changes in processor allotment to the job between quanta. Two simple modifications make the work-stealing algorithm adaptive.

Allotment gain: When the allotment increases from quantum $q-1$ to q, the job scheduler obtains $a_q - a_{q-1}$ additional processors. Since the deques of these new processors start out empty, all these processors immediately start stealing to get work from the other processors.

Allotment loss: When the allotment decreases from quantum $q-1$ to q, the job scheduler deallocates $a_{q-1} - a_q$ processors, whose deques may be nonempty. To deal with these deques, we use the concept of "mugging" [BLS]. When a processor runs out of work, instead of stealing immediately, it looks for a ***muggable*** deque, a nonempty deque that has no associated processor working on it. Upon finding a muggable deque, the thief ***mugs*** the deque by taking over

[4] These algorithms impose some additional restrictions on the job, for example, that each node has an out-degree of at most 2. Whatever restrictions assumed by the underlying work-stealing algorithm apply to A-STEAL as well.

the entire deque as its own. Thereafter, it works on the deque as if it were its own. If there are no muggable deques, the thief steals normally. Data structures can be set up between quanta so that stealing and mugging can be accomplished in $O(1)$ time [Sen04].

At all time steps during the execution of A-STEAL, every processor is either working, stealing, or mugging. We call the cycles that a processor spends on working, stealing, and mugging as ***work-cycles***, ***steal-cycles***, and ***mug-cycles***, respectively. We assume without loss of generality that work-cycles, steal-cycles, and mug-cycles all take single time steps. We bound time and waste in terms of these elementary processor cycles. Cycles spent stealing and mugging are ***wasted***, and the total waste is the sum of the number of steal-cycles and mug-cycles during the execution of the job.

A-STEAL*'s desire-estimation heuristic*

The desire estimation algorithm for A-STEAL is similar to A-GREEDY's desire estimation. Again, A-STEAL classifies the previous quantum as either "satisfied" or "deprived" and either "efficient" or "inefficient." The classification of satisfied versus deprived is identical to A-GREEDY. However, the classification of efficient versus inefficient is slightly different. Instead if usage (as with A-GREEDY), A-STEAL uses ***nonsteal usage***, which is the sum of the number of work-cycles and mug-cycles. Although it might seem counter intuitive for the definition of "efficient" to include mug-cycles, which, after all, are wasted, the rationale is that mug-cycles arise as a result of an allotment loss and do not generally indicate that the job has a surplus of processors. Apart from this difference, the desire estimation of A-STEAL is similar to that of A-GREEDY.

2.5 Trim Analysis of A-GREEDY for Unit Quanta

This section uses a trim analysis to analyze A-GREEDY for the special case where $L = 1$, that is, where each quantum is a ***unit*** quantum consisting of only a single time step. For unit quanta and for greedy schedulers, adaptive scheduling can be done efficiently using instantaneous parallelism as feedback, since the scheduler knows exactly how many tasks are ready for the next step and request exactly the right number of processors. However, this approach of using instantaneous parallelism does not generalize to longer quanta, or to work-stealing schedulers. Surprisingly, A-GREEDY's algorithm for desire estimation, which only uses historical information, provides nearly as good time bounds as the approach that uses instantaneous parallelism even for unit quanta. Moreover, these bounds can be extended to the case when $L \gg 1$ (Section 2.6) and to work-stealing schedulers (Section 2.7). The analysis for unit quanta given in this section gives intuition for the effectiveness of A-GREEDY's strategy for desire estimation.

For unit quanta, we shall prove that A-GREEDY with utilization parameter $\delta = 1$ completes a job with work T_1 and critical-path length T_∞ in at most $T \leq T_1/\widetilde{P} + 2T_\infty + \log_\rho P + 1$ time steps, where P is the number of processors in the machine and $\widetilde{P}$ is the $(2T_\infty + \log_\rho P + 1)$-trimmed availability. In contrast, a greedy thread scheduler that uses instantaneous parallelism as feedback completes the job in at most $T \leq T_1/\widetilde{P} + T_\infty$ time steps, where $\widetilde{P}$ is the T_∞-trimmed availability. Thus, even without up-to-date information on instantaneous parallelism, A-GREEDY operates nearly as efficiently. Moreover, the total number of processor cycles wasted by A-GREEDY in the course of the computation is bounded by ρT_1. (Instantaneous parallelism wastes none.)

To prove the completion-time bounds, we use a trim analysis. We label each quantum as either ***accounted*** or ***deductible***. Accounted quanta are those where $u_q = p_q$, that is, the usage equals the processor availability. The deductible quanta are those where $u_q < p_q$. Our trim analysis will show that when we ignore the relatively few deductible quanta, we obtain linear speedup on the more numerous accounted quanta.

We first relate the labeling of accounted and deductible to the three-way classification of quanta as inefficient, efficient-and-satisfied, and efficient-and-deprived.

Inefficient: In an inefficient quantum q, we have $u_q < a_q \leq p_q$, that is, the job uses fewer processors than it was allotted, and therefore it uses fewer processors than those available. Thus, inefficient quanta are deductible quanta, irrespective of whether they were satisfied or deprived.

Efficient-and-satisfied: On an efficient quantum q, we have $u_q = a_q$. Since $a_q = \min\{p_q, d_q\}$ by definition, on a satisfied quantum, we have $a_q = d_q \leq p_q$. Thus, we have $u_q \leq p_q$. Since we cannot guarantee that $u_q = p_q$, we assume pessimistically that quantum q is deductible.

Efficient-and-deprived: As before, on an efficient quantum q, we have $u_q = a_q$. On a deprived quantum, we have by definition that $a_q < d_q$, and since $a_q = \min\{p_q, d_q\}$, we have $a_q = p_q$. Thus, we have $u_q = a_q = p_q$, and quantum q is accounted.

Time Analysis

We prove the completion time bound of A-GREEDY by bounding the number of deductible and accounted quanta separately. We use a potential function argument to prove that the number of deductible quanta is at most $2T_\infty + \log_\rho P + 1$. We then show that the number of accounted quanta is at most T_1/P_A, where T_1 is the total work and $\widetilde{P}$ and P_A is the mean availability on accounted quanta. Thus, the total time to complete the job is at most $T_1/P_A + 2T_\infty + \log_\rho P + 1$, which is the sum of the number of accounted and deductible quanta, since each quantum consists of a single time step. Finally, we show that $P_A \geq \widetilde{P}$, where $\widetilde{P}$ is the $(2T_\infty + L\log_\rho P + 1)$-trimmed availability, which yields the desired result.

Our analysis uses a characterization of greedy scheduling based on whether the job uses all its allotted processors on a given step. We define a step to be ***complete*** if the job uses all the allotted processors in the step and ***incomplete*** if the job does not use all the available processors. In the special case of A-GREEDY with unit quanta, an inefficient quantum consists of a single incomplete step and an efficient quantum consists of a single complete step. The following lemma from the literature [Blu95, BL99, EZL89] shows that whenever a greedy scheduler (including A-GREEDY) schedules an incomplete step, the job makes progress on its critical path.

Lemma 2.1 *Any greedy scheduler reduces the length of a job's remaining critical path by* 1 *after every incomplete step.* □

We next bound the maximum desire during the course of the computation.

Lemma 2.2 *Suppose that* A-GREEDY *schedules a job on a machine with P processors. If ρ is* A-GREEDY*'s responsiveness parameter, then for every quantum q, the job's desire satisfies $d_q \leq \rho P$.*

PROOF. We use induction on the number of quanta. The base case $d_1 = 1$ holds trivially. If a given quantum $q-1$ was inefficient, the desire d_q decreases, and thus $d_q < d_{q-1} \leq \rho P$ by induction. If quantum $q-1$ was efficient-and-satisfied, then $d_q = \rho d_{q-1} = \rho a_{q-1} \leq \rho P$. If quantum $q-1$ was efficient-and-deprived, then $d_q = d_{q-1} \leq \rho P$ by induction. □

The deductible quanta for A-GREEDY are either inefficient or efficient and-satisfied. The next lemma bounds their number.

Lemma 2.3 *Suppose that* A-GREEDY *schedules a job with critical-path length T_∞ on a machine with P processors. If ρ is* A-GREEDY*'s responsiveness parameter, $\delta = 1$ is its utilization parameter, and $L = 1$ is the quantum length, then the schedule produces at most $2T_\infty + \log_\rho P + 1$ deductible quanta.*

PROOF. We use a potential-function argument based on the job's desire d_q before quantum q. Define the potential before quantum q to be

$$\Phi(q) = 2T_\infty^q - \log_\rho d_q \,,$$

where T_∞^q denotes the length of the remaining critical path before quantum q is executed, that is, the length of the longest path in the unexecuted dag. The initial potential is

$$\begin{aligned} \Phi(1) &= 2T_\infty - \log_\rho d_1 \\ &= 2T_\infty \,, \end{aligned}$$

since the desire in the first quantum is $d_1 = 1$. If the job executes for Q quanta, the final potential is

$$\begin{aligned} \Phi(Q+1) &= 2T_\infty^{Q+1} - \log_\rho d_{Q+1} \\ &\geq 0 - \log_\rho(\rho P) \\ &= -\log_\rho P - 1 \,, \end{aligned}$$

by Lemma 2.2. Since the potential starts at $2T_\infty$ and is at least $-\log_\rho P - 1$ at the end of the computation, the total decrease of the potential is $\Phi(1) - \Phi(Q+1) \leq 2T_\infty + \log_\rho P + 1$.

We now compute the decrease in potential during each quantum based on the three-way classification. Each case will use the fact that the decrease in potential during any quantum q is

$$\begin{aligned} \Delta\Phi &= \Phi(q) - \Phi(q+1) \\ &= (2T_\infty^q - \log_\rho d_q) - (2T_\infty^{q+1} - \log_\rho d_{q+1}) \\ &= 2(T_\infty^q - T_\infty^{q+1}) - (\log_\rho d_q - \log_\rho d_{q+1}) \,. \end{aligned}$$

Inefficient: An inefficient quantum q consists of a single incomplete step. After an incomplete step, the length of the remaining critical path reduces by 1 (Lemma 2.1). Moreover, we have $d_{q+1} = d_q/\rho$, since A-GREEDY reduces the desire after an inefficient quantum. Thus, the decrease in potential after an inefficient quantum is

$$\begin{aligned} \Delta\Phi &= 2(T_\infty^q - T_\infty^{q+1}) - (\log_\rho d_q - \log_\rho d_{q+1}) \\ &= 2\left(T_\infty^q - (T_\infty^q - 1)\right) - \left(\log_\rho d_q - \log_\rho(d_q/\rho)\right) \\ &= 2(1) - (1) \\ &= 1 \,. \end{aligned}$$

Efficient-and-satisfied: A-GREEDY increases the desire after every efficient-and-satisfied quantum ($d_{q+1} = \rho d_q$). The remaining critical-path length never increases. Thus, the decrease in potential is

$$\begin{aligned}\Delta\Phi &= 2(T_\infty^q - T_\infty^{q+1}) - (\log_\rho d_q - \log_\rho d_{q+1}) \\ &\geq 0 - \left(\log_\rho d_q - \log_\rho(\rho d_q)\right) \\ &= 1\,.\end{aligned}$$

Efficient-and-deprived: After efficient-and-satisfied quanta, A-GREEDY maintains the previous desire ($d_{q+1} = d_q$), and, as before, the critical-path length never increases. Thus, the decrease in potential is

$$\begin{aligned}\Delta\Phi &= 2(T_\infty^q - T_\infty^{q+1}) - (\log_\rho d_q - \log_\rho d_{q+1}) \\ &\geq 0\,.\end{aligned}$$

Thus, the potential never increases, and it decreases by at least 1 after every deductible quantum. Thus, the number of deductible quanta is at most $2T_\infty + \log_\rho P + 1$, the total decrease in potential. □

We now bound the number of accounted quanta.

Lemma 2.4 *Suppose that* A-GREEDY *schedules a job with work T_1. If $\delta = 1$ is* A-GREEDY*'s utilization parameter and $L = 1$ is the quantum length, then the schedule produces at most T_1/P_A accounted quanta, where P_A is the mean availability on accounted quanta.*

PROOF. Let A be the set of accounted quanta, and D be the set of deductible quanta. The mean availability on accounted quanta is $P_A = (1/|A|)\sum_{q\in A} p_q$. The total number of tasks executed over the course of the computation is $T_1 = \sum_{q\in A\cup D} u_q$, since each of the T_1 tasks is executed exactly once in either an accounted or a deductible quantum. Since accounted quanta are those for which $u_q = p_q$, we have

$$\begin{aligned}T_1 &= \sum_{q\in A\cup D} u_q \\ &\geq \sum_{q\in A} u_q \\ &= \sum_{q\in A} p_q \\ &= |A|\,P_A\end{aligned}$$

Thus, the number of accounted quanta is $|A| \leq T_1/P_A$. □

We can now bound the completion time of a job scheduled by A-GREEDY with unit quanta.

Theorem 2.5 *Suppose that* A-GREEDY *schedules a job with work T_1 and critical-path length T_∞ on a machine with P processors. If ρ is* A-GREEDY*'s responsiveness parameter, $\delta = 1$ is its utilization parameter, and $L = 1$ is the quantum length, then* A-GREEDY *completes the job in*

$$T \leq T_1/\widetilde{P} + 2T_\infty + \log_\rho P + 1$$

time steps, where $\widetilde{P}$ is the $(2T_\infty + \log_\rho P + 1)$-trimmed availability.

PROOF. The proof is a trim analysis. Let A be the set of accounted quanta, and D be the set of deductible quanta. Lemma 2.3 shows that there are $|D| \leq 2T_\infty + L \lg P + 1$ deductible time steps, since each quantum consists of a single time step. We have $P_A \geq \widetilde{P}$, since the mean availability on the accounted time steps (we trim the $|D|$ deductible steps) must be at least the $(2T_\infty + L \lg P + 1)$-trimmed availability (we trim the $2T_\infty + L \lg P + 1$ steps that have the highest availability). From Lemma 2.4, the number of accounted quanta is $|A| \leq T_1/P_A \leq T_1/\widetilde{P}$, and since $T = L(|A| + |D|)$, the desired time bound follows. □

Waste Analysis

We now prove the waste bound for A-GREEDY with unit quanta. Let $w_q = a_q - u_q$ be the waste of quantum q. In efficient quanta, the usage is $u_q = a_q$, and the waste is $w_q = 0$. Therefore, the job wastes processor cycles only on inefficient quanta. The next theorem shows that the waste on inefficient quanta can be amortized against the work done on efficient quanta.

Theorem 2.6 *Suppose that* A-GREEDY *schedules a job with work T_1 on a machine. If ρ is* A-GREEDY*'s responsiveness parameter, $\delta = 1$ is its utilization parameter, and $L = 1$ is the quantum length, then* A-GREEDY *wastes at most ρT_1 processor cycles in the course of its computation.*

PROOF. We use a potential-function argument based on the job's desire d_q before quantum q. Define the potential $\Psi(q)$ before quantum q as

$$\Psi(q) = \rho T_1^q + \frac{\rho}{\rho - 1} d_q \,,$$

where T_1^q is the total number of unexecuted tasks in the computation before quantum q. Thus, the initial potential is

$$\begin{aligned} \Psi(1) &= \rho T_1^1 + \frac{\rho}{\rho - 1} d_1 \\ &= \rho T_1 + \rho/(\rho - 1) \,, \end{aligned}$$

since $d_1 = 1$. If the job executes for Q quanta, the final potential is

$$\begin{aligned} \Psi(Q+1) &= \rho T_1^{Q+1} + \frac{\rho}{\rho - 1} d_{Q+1} \\ &\geq 0 + \rho/(\rho - 1) \,, \\ &= \rho/(\rho - 1) \,, \end{aligned}$$

since the desire d_q of any quantum q is at least 1. Therefore the total decrease in potential is $\Psi(1) - \Psi(Q+1) \leq \rho T_1$.

Based on the three-way classification, we shall show that if the waste on quantum q is $w_q = a_q - u_q$, then the potential decreases by at least w_q during the quantum. Each way will use the fact that the decrease in potential during any quantum q is

$$\begin{aligned} \Delta\Psi_q &= \Psi(q) - \Psi(q+1) \\ &= \left(\rho T_1^q + \frac{\rho}{\rho - 1} d_q\right) - \left(\rho T_1^{q+1} + \frac{\rho}{\rho - 1} d_{q+1}\right) \\ &= \rho(T_1^q - T_1^{q+1}) + \frac{\rho}{\rho - 1}(d_q - d_{q+1}) \,. \end{aligned}$$

Inefficient: For any quantum q, $w_q < a_q$, which is to say, the number of processor cycles wasted is less than the total number of processor cycles allotted. Since the allotment is $a_q \leq d_q$, we have $w_q < d_q$. After an inefficient quantum q, A-GREEDY reduces the desire to be $d_{q+1} = d_q/\rho$. Thus, the decrease in potential is

$$\begin{aligned}
\Delta\Psi_q &= \rho(T_1^q - T_1^{q+1}) + \frac{\rho}{\rho-1}(d_q - d_{q+1}) \\
&> \frac{\rho}{\rho-1}(d_q - d_q/\rho) \\
&= d_q \\
&> w_q \,.
\end{aligned}$$

Efficient-and-satisfied: Since no processor cycles are wasted on any efficient quantum q, we have $w_q = 0$ and the remaining work reduces by $u_q = a_q$. On an efficient-and-satisfied quantum q, the allotment is the same as the desire ($a_q = d_q$) and A-GREEDY increases the desire ($d_{q+1} = \rho d_q$) after the quantum. Thus, the decrease in potential is

$$\begin{aligned}
\Delta\Psi_q &= \rho(T_1^q - T_1^{q+1}) + \frac{\rho}{\rho-1}(d_q - d_{q+1}) \\
&= \rho a_q + \frac{\rho}{\rho-1}(d_q - \rho d_q) \\
&= \rho d_q - \rho d_q \\
&= 0 \\
&= w_q \,.
\end{aligned}$$

Efficient-and-deprived: On any efficient quantum q, we have $w_q = 0$ and the amount of remaining work reduces by $u_q = a_q$. Since the quantum q is efficient-and-deprived, we have $d_{q+1} = d_q$, because A-GREEDY maintains the previous desire. Therefore, the decrease in potential is

$$\begin{aligned}
\Delta\Psi &= \rho(T_1^q - T_1^{q+1}) + \frac{\rho}{\rho-1}(d_q - d_{q+1}) \\
&= \rho a_q + 0 \\
&> 0 \\
&= w_q \,.
\end{aligned}$$

In all three cases, if the job wastes w_q processors in quantum q, the potential decreases by at least w_q. Consequently, the total waste during the course of the computation is at most ρT_1, the total decrease in potential. □

2.6 Trim Analysis of A-GREEDY for the General Case

We now use a trim analysis to analyze the general case of A-GREEDY when each scheduling quantum has L time steps, δ is the utilization parameter, ρ is the responsiveness parameter, and P is the number of processors in the machine. For a job with work T_1 and critical-path length T_∞,

A-GREEDY achieves the following bounds on running time and waste, where $\widetilde{P}$ is the $(2T_\infty/(1-\delta) + L\log_\rho P + L)$-trimmed availability:

$$\begin{aligned} T &\leq \frac{T_1}{\delta\widetilde{P}} + \frac{2T_\infty}{1-\delta} + L\log_\rho P + L\,, \\ W &\leq \frac{1+\rho-\delta}{\delta} T_1\,. \end{aligned}$$

As in Section 2.5, we label each quantum as either accounted or deductible. Recall that a quantum q of length L and processor availability p_q has a total of Lp_q processor cycles available. Accounted quanta are those for which $u_q \geq \delta L p_q$, that is, the job uses at least a δ fraction of all available processor cycles. The deductible quanta are those for which $u_q < L\delta p_q$. By the same logic as in Section 2.5, inefficient quanta or efficient-and-satisfied quanta are labeled deductible. Efficient-and-deprived quanta, on the other hand, are labeled accounted.

Time Analysis

We bound the accounted and deductible quanta separately. We first show how inefficient quanta affect the remaining critical path of the job.

Lemma 2.7 A-GREEDY *reduces the length of a job's remaining critical path by at least* $(1-\delta)L$ *after every inefficient quantum, where* δ *is* A-GREEDY*'s utilization parameter and* L *is the quantum length.*

PROOF. The total number of tasks completed in an inefficient quantum q is less than $\delta L a_q$. Therefore, there can be at most δL complete steps in an inefficient quantum, since on a complete step, the job uses all the allotted processors, completing a_q tasks. Since there are L time steps in a quantum, there are at least $(1-\delta)L$ incomplete steps. Thus, the critical path reduces by at least $(1-\delta)L$, since Lemma 2.1 shows that every incomplete step reduces the critical path by 1. □

The next lemma bounds the number of deductible quanta.

Lemma 2.8 *Suppose that* A-GREEDY *schedules a job with critical-path length* T_∞ *on a machine with* P *processors. If* ρ *is* A-GREEDY*'s responsiveness parameter,* δ *is its utilization parameter, and* L *is the quantum length, then the schedule produces at most* $2T_\infty/((1-\delta)L) + \log_\rho P + 1$ *deductible quanta.*

PROOF. We use a potential-function argument as in Lemma 2.3. Define the potential before quantum q as

$$\Phi(q) = 2T_\infty^q/(1-\delta)L - \log_\rho d_q\,,$$

where T_∞^q is the remaining critical path before quantum q. If the job completes in Q quanta, the total decrease in potential is

$$\begin{aligned} \Phi_1 - \Phi_{Q+1} &= \frac{2T_\infty - 0}{(1-\delta)L} - (\log_\rho 1 - \log_\rho d_{Q+1}) \\ &\leq \frac{2T_\infty}{(1-\delta)L} + \log_\rho P + 1\,, \end{aligned}$$

since $d_{Q+1} \leq \rho P$ by Lemma 2.2.

We can compute the decrease in potential during each quantum based on the three-way classification. The decrease in potential in quantum q is

$$\begin{aligned}
\Delta\Phi &= \Phi(q) - \Phi(q+1) \\
&= 2T_\infty^q/(1-\delta)L - \log_\rho d_q - (2T_\infty^{q+1}/(1-\delta)L - \log_\rho d_{q+1}) \\
&= \frac{2}{(1-\delta)L}(T_\infty^q - T_\infty^{q+1}) - (\log_\rho d_q - \log_\rho d_{q+1}) \,.
\end{aligned}$$

Inefficient: By Lemma 2.7, the critical path reduces by at least $(1-\delta)L$ during an inefficient quantum. Moreover, we have $d_{q+1} = d_q/\rho$, since A-GREEDY reduces the desire after an inefficient quantum. Thus, the decrease in potential after an inefficient quantum is

$$\begin{aligned}
\Delta\Phi &= \frac{2}{(1-\delta)L}(T_\infty^q - T_\infty^{q+1}) - (\log_\rho d_q - \log_\rho d_{q+1}) \\
&= \frac{2}{(1-\delta)L}\left(T_\infty^q - (T_\infty^q - (1-\delta)L)\right) - \left(\log_\rho d_q - \log_\rho(d_q/\rho)\right) \\
&= 2(1) - (1) \\
&= 1 \,.
\end{aligned}$$

Efficient-and-satisfied: A-GREEDY increases the desire after every efficient-and-satisfied quantum ($d_{q+1} = \rho d_q$). The remaining critical-path length never increases. Thus, the decrease in potential is at least 1.

Efficient-and-deprived: After efficient-and-satisfied quanta, A-GREEDY maintains the previous desire ($d_{q+1} = d_q$), and, as before, the critical-path length never increases. Thus, the potential does not increase.

Thus, the potential never increases, and it decreases by at least 1 after every deductible quantum. Thus, the number of deductible quanta is at most $2T_\infty/((1-\delta)L) + \log_\rho P + 1$, the total decrease in potential. □

We now bound the number of accounted quanta.

Lemma 2.9 *Suppose that* A-GREEDY *schedules a job with work T_1. If δ is* A-GREEDY*'s utilization parameter and L is the quantum length, then the schedule produces at most $T_1/\delta L P_A$ accounted quanta, where P_A is the mean availability on accounted quanta.*

PROOF. Let A be the set of accounted quanta, and let D be the set of deductible quanta. The mean availability on accounted quanta is $P_A = (1/|A|)\sum_{q\in A} p_q$. The total number of tasks executed in the course of the computation is $T_1 = \sum_{q\in A\cup D} u_q$. Since the accounted quanta are those for which

$u_q \geq \delta L p_q$, we have

$$\begin{aligned} T_1 &= \sum_{q \in A \cup D} u_q \\ &\geq \sum_{q \in A} u_q \\ &\geq \sum_{q \in A} \delta L p_q \\ &= \delta L \, |A| \, P_A \, . \end{aligned}$$

Therefore, the total number of accounted quanta is at most $|A| \leq T_1 / \delta L P_A$. □

The next theorem provides the time bound for A-GREEDY.

Theorem 2.10 *Suppose that* A-GREEDY *schedules a job with work T_1 and critical-path length T_∞ on a machine with P processors. If ρ is* A-GREEDY*'s responsiveness parameter, δ is its utilization parameter, and L is the quantum length, then* A-GREEDY *completes the job in*

$$T \leq T_1 / \delta \widetilde{P} + 2T_\infty / (1 - \delta) + L \log_\rho P + L$$

time steps, where $\widetilde{P}$ is the $(2T_\infty/(1-\delta) + L \log_\rho P + L)$-trimmed availability.

PROOF. The proof is a trim analysis. Let A be the set of accounted quanta, and D be the set of deductible quanta. Lemma 2.8 shows that there are $|D| \leq 2T_\infty/(1-\delta)L + \log_\rho P + 1$ deductible quanta, and hence at most $L\,|D| = 2T_\infty/(1-\delta) + L \log_\rho P + L$ time steps belong to deductible quanta. We have that $P_A \geq \widetilde{P}$, since the mean availability on the accounted time steps (we trim the $L\,|D|$ deductible steps) must be at least the $(2T_\infty/(1-\delta) + L \log_\rho P + L)$-trimmed availability (we trim the $2T_\infty/(1-\delta) + L \log_\rho P + L$ steps that have the highest availability). From Lemma 2.9, the number of accounted quanta is $|A| \leq T_1/\delta P_A \leq T_1/\delta \widetilde{P}$, and since $T = L(|A| + |D|)$, the desired time bound follows. □

Waste Analysis

We now prove the waste bound for A-GREEDY.

Theorem 2.11 *Suppose that* A-GREEDY *schedules a job with work T_1 on a machine. If ρ is* A-GREEDY*'s responsiveness parameter, δ is its utilization parameter, and L is the quantum length, then* A-GREEDY *wastes at most $(1 + \rho - \delta) T_1 / \delta$ processor cycles in the course of its computation.*

PROOF. We can prove the bound using a potential-function argument similar to the one presented in Theorem 2.6. In this case the potential before quantum q is defined as

$$\Psi(q) = \frac{1 + \rho - \delta}{\delta} T_1^q + \frac{\rho}{\rho - 1} L d_q \, ,$$

where T_1^q is the total number of unexecuted tasks in the computation before quantum q and d_q is the desire for quantum q.

Instead, here we use an accounting argument for better intuition about A-GREEDY's waste. We amortize the waste different quanta according to whether they are efficient or inefficient.

Inefficient: Based on Lemma 2.19, every inefficient quantum q maps to a unique efficient-and-satisfied quantum r such that $d_r = d_q/\rho$. Therefore, the waste on the inefficient quantum q can be amortized against the work done on efficient-and-satisfied quantum r. During the inefficient quantum q, the waste is at most $w_q < La_q$. The work done on the corresponding efficient-and-satisfied quantum is $u_r \geq \delta La_r = \delta Ld_r = \delta Ld_q/\rho$. Thus, the waste (less than Ld_q) during the inefficient quantum q is at most ρ/δ times of the work (at least $\delta Ld_q/\rho$) on its corresponding efficient-and-satisfied quantum r. Thus, the total waste over all inefficient quanta is at most $\rho T_1/\delta$, which is at most ρ/δ times the work (at most T_1) on efficient-and-satisfied quanta.

Efficient: The job completes at least $L\delta a_q$ work on an efficient quantum with allotment a_q, and wastes at most $(1-\delta)La_q$ processor cycles. Therefore the total waste on efficient quanta is at most $((1-\delta)/\delta)T_1$, which is $((1-\delta)/\delta$ times the work done in efficient quanta (at most T_1).

Since the total waste is the sum of the waste on efficient and inefficient time steps, it is at most $((1+\rho-\delta)/\delta)T_1$. □

We can decompose the bounds of Theorems 2.10 and 2.11 into separate bounds for accounted and deductible quanta.

Corollary 2.12 *Suppose that* A-GREEDY *schedules a job with work T_1 and critical-path length T_∞ on a machine with P processors, and suppose that ρ is* A-GREEDY*'s responsiveness parameter, δ is its utilization parameter, and L is the quantum length. Let T_a and T_d be the number of time steps in accounted and deductible quanta, respectively, and let W_a and W_d be the waste on accounted and deductible quanta, respectively. Then,* A-GREEDY *achieves the following bounds:*

$$\begin{aligned} T_a &\leq (1/\delta)T_1/\widetilde{P}\,, \\ T_d &\leq (2\min\{L, 1/(1-\delta)\})T_\infty + L\log_\rho P + L\,, \\ W_a &\leq (1/\delta - 1)T_1\,, \\ W_d &\leq (\rho/\delta)T_1\,. \end{aligned}$$

□

As can be seen from these inequalities, the bounds for accounted quanta are stronger than those for deductible quanta. The reason is that the job scheduler in our model is adversarial. In practice, however, it seems unlikely that the job scheduler would actually act as an adversary. Thus, A-GREEDY's behavior on the deductible quanta is likely to be much better than these worst-case bounds predict. Moreover, since the adversary's bad behavior is limited to relatively few deductible quanta, we conjecture that in practice the overall time and waste of a real scheduler based on A-GREEDY more closely follows the bounds for accounted quanta.

2.7 Trim Analysis of A-STEAL

This section uses a trim analysis to analyze A-STEAL with respect to both time and waste. Suppose that A-STEAL schedules a job with work T_1 and span T_∞ on a machine with P processors. Let ρ denote A-STEAL's responsiveness parameter, δ the utilization parameter, and L the quantum length. We will show that A-STEAL completes the job in time $T = O\left(T_1/\widetilde{P} + T_\infty + L\lg P + L\ln(1/\epsilon)\right)$

with probability at least $1 - \epsilon$, where $\widetilde{P}$ denotes the $O(T_\infty + L \lg P + L \ln(1/\epsilon))$-trimmed availability. This bound implies that A-STEAL achieves linear speed-up on all the time steps excluding at most $O(T_\infty + L \lg P + L \ln(1/\epsilon))$ time steps with highest processor availability. Moreover, A-STEAL guarantees that the total number of processor cycles wasted during the job's execution is $W = O(T_1)$.

We prove these bounds using a trim analysis. We label each quantum as either ***accounted*** or ***deductible***. Accounted quanta are those with $n_q \geq L\delta p_q$, where n_q denotes the nonsteal usage. That is, the job works or mugs for at least a δ fraction of the Lp_q processor cycles possibly available during the quantum. Conversely, the deductible quanta are those where $n_q < L\delta p_q$. Our trim analysis will show that when we ignore the relatively few deductible quanta, we obtain linear speedup on the more numerous accounted quanta. We can relate this labeling to a three-way classification of quanta as inefficient, efficient-and-satisfied, and efficient-and-deprived:

- **Inefficient:** In an inefficient quantum q, we have $n_q < L\delta a_q \leq L\delta p_q$, since the allotment a_q never exceeds the availability p_q. Thus, we label all inefficient quanta as deductible, irrespective of whether they are satisfied or deprived.

- **Efficient-and-satisfied:** On an efficient quantum q, we have $n_q \geq L\delta a_q$. Since we have $a_q = \min\{p_q, d_q\}$, for a satisfied quantum it follows that $a_q = d_q \leq p_q$. Despite these two bounds, we may nevertheless have $n_q < L\delta p_q$. Since we cannot guarantee that $n_q \geq L\delta p_q$, we pessimistically label the quantum q as deductible.

- **Efficient-and-deprived:** As before, on an efficient quantum q, we have $n_q \geq L\delta a_q$. On a deprived quantum, we have $a_q < d_q$ by definition. Since $a_q = \min\{p_q, d_q\}$, we must have $a_q = p_q$. Hence, it follows that $n_q \geq L\delta a_q = L\delta p_q$, and we label quantum q as accounted.

Time analysis

We now analyze the execution time of A-STEAL by separately bounding the number of deductible and accounted quanta. Two observations provide intuition for the proof. First, each inefficient quantum contains a large number of steal-cycles, which we can expect to reduce the length of the remaining span. This observation will help us to bound the number of deductible quanta. Second, most of the processor cycles in an efficient quantum are spent either working or mugging. We will show that there cannot be too many mug-cycles during the job's execution, and thus most of the processor cycles on efficient quanta are spent doing useful work. This observation will help us to bound the number of accounted quanta.

The following lemma, proved in Lemma 11 of [BL99], shows how steal-cycles reduce the length of the job's span.

Lemma 2.13 *If a job has r deques of ready threads, then $3r$ steal-cycles suffice to reduce the length of the job's remaining span by at least 1 with probability at least $1 - 1/e$, where e is the base of the natural logarithm.* □

The next lemma shows that an inefficient quantum reduces the length of the job's span, which we will later use to bound the total number of inefficient quanta.

Lemma 2.14 *Let δ denote* A-STEAL*'s utilization parameter, and L the quantum length. With probability greater than $1/4$,* A-STEAL *reduces the length of a job's remaining span in an inefficient quantum by at least $(1-\delta)L/6$.*

PROOF. Let q be an inefficient quantum. A processor with an empty deque steals only when it cannot mug a deque, and hence, all the steal-cycles in quantum q occur when the number of nonempty deques is at most the allotment a_q. Therefore, by Lemma 2.13, $3a_q$ steal-cycles suffice to reduce the span by 1 with probability at least $1-1/e$. Since the quantum q is inefficient, it contains at least $(1-\delta)La_q$ steal-cycles. Divide the time steps of the quantum into ***rounds*** such that each round contains $3a_q$ steal-cycles, except for possibly the last. Thus, there are at least $m = (1-\delta)La_q/3a_q = (1-\delta)L/3$ rounds.[5] We call a round ***good*** if it reduces the length of the span by at least 1; otherwise, the round is ***bad***. For each round i in quantum q, we define the indicator random variable X_i to be 1 if round i is a bad round and 0 otherwise, and let $X = \sum_{i=1}^{m} X_i$. Since we have $\Pr\{X_i = 1\} < 1/e$, linearity of expectation dictates that $\mathrm{E}[X] < m/e$. We now apply Markov's inequality [CLRS01, p. 1111], which says that for a nonnegative random variable X, we have $\Pr\{X \geq t\} \leq \mathrm{E}[X]/t$ for all $t > 0$. Substituting $t = m/2$, we obtain $\Pr\{X > m/2\} \leq \mathrm{E}[X]/(m/2) \leq (m/e)/(m/2) = 2/e < 3/4$. Thus, the probability exceeds $1/4$ that quantum q contains at least $m/2$ good rounds. Since each good round reduces the span by at least 1, with probability greater than $1/4$, the span is reduced during quantum q by at least $m/2 = ((1-\delta)L/3)/2 = (1-\delta)L/6$. □

Lemma 2.15 *Suppose that* A-STEAL *schedules a job with span T_∞ on a machine. Let ρ denote* A-STEAL*'s responsiveness parameter, δ the utilization parameter, and L the quantum length. Then, for any $\epsilon > 0$, with probability at least $1-\epsilon$, the schedule produces at most $48T_\infty/(L(1-\delta)) + 16\ln(1/\epsilon)$ inefficient quanta.*

PROOF. Let I be the set of inefficient quanta. Define an inefficient quantum q as ***productive*** if it reduces the span by at least $(1-\delta)L/6$, and ***unproductive*** otherwise. For each quantum $q \in I$, define the indicator random variable Y_q to be 1 if q is productive and 0 otherwise. By Lemma 2.14, we have $\Pr\{Y_q = 1\} > 1/4$. Let the total number of productive quanta be $Y = \sum_{q \in I} Y_q$. For simplicity in notation, let $A = 6T_\infty/((1-\delta)L)$. If the job's execution contains $|I| \geq 48T_\infty/((1-\delta)L) + 16\ln(1/\epsilon)$ inefficient quanta, then we have $\mathrm{E}[Y] > |I|/4 \geq 12T_\infty/((1-\delta)L) + 4\ln(1/\epsilon) = 2A + 4\ln(1/\epsilon)$. Using the Chernoff bound $\Pr\{Y < (1-\lambda)E[Y]\} < \exp(-\lambda^2 E[Y]/2)$ [MR95,

[5] Actually, the number of rounds is $m = \lfloor (1-\delta)L/3 \rfloor$, but we will ignore the roundoff for simplicity. A more detailed analysis can nevertheless produce the same constants in the bounds for Lemmas 2.15 and 2.18.

p. 70] and choosing $\lambda = (A + 4\ln(1/\epsilon)) / (2A + 4\ln(1/\epsilon))$, we obtain

$$\begin{aligned}
& \Pr\{Y < A\} \\
= \; & \Pr\left\{Y < \left(1 - \frac{A + 4\ln(1/\epsilon)}{2A + 4\ln(1/\epsilon)}\right)(2A + 4\ln(1/\epsilon))\right\} \\
= \; & \Pr\{Y < (1-\lambda)(2A + 4\ln(1/\epsilon))\} \\
< \; & \exp\left(-\frac{\lambda^2}{2}(2A + 4\ln(1/\epsilon))\right) \\
= \; & \exp\left(-\frac{1}{2} \cdot \frac{(A + 4\ln(1/\epsilon))^2}{2A + 4\ln(1/\epsilon)}\right) \\
< \; & \exp\left(-\frac{1}{2} \cdot 4\ln(1/\epsilon) \cdot \frac{1}{2}\right) \\
= \; & \epsilon \,.
\end{aligned}$$

Therefore, if the number of inefficient quanta is $|I| = 48T_\infty/((1-\delta)L) + 16\ln(1/\epsilon)$, the number of productive quanta is at least $A = 6T_\infty/((1-\delta)L)$ with probability at least $1-\epsilon$. By Lemma 2.14 each productive quantum reduces the span by at least $(1-\delta)L/6$, the total span reduced is no less than T_∞ with probability at least $1-\epsilon$ when $|I| = 48T_\infty/((1-\delta)L) + 16\ln(1/\epsilon)$. In order words, with probability at least $1-\epsilon$, the job encounters no more than $48T_\infty/((1-\delta)L) + 16\ln(1/\epsilon)$ inefficient quanta before it completes. Therefore, the number of inefficient quanta is $|I| \leq 48T_\infty/((1-\delta)L) + 16\ln(1/\epsilon)$ with probability at least $1-\epsilon$. □

The following technical lemma bounds the maximum value of desire.

Lemma 2.16 *Suppose that* A-STEAL *schedules a job on a machine with P processors. Let ρ denote* A-STEAL*'s responsiveness parameter. Before any quantum q, the desire d_q of the job is at most ρP.*

PROOF. We use induction on the number of quanta. The base case $d_1 = 1$ holds trivially. If a given quantum $q-1$ was inefficient, the desire d_q decreases, and thus $d_q < d_{q-1} \leq \rho P$ by induction. If quantum $q-1$ was efficient-and-satisfied, then $d_q = \rho d_{q-1} = \rho a_{q-1} \leq \rho P$. If quantum $q-1$ was efficient-and-deprived, then $d_q = d_{q-1} \leq \rho P$ by induction. □

The next lemma reveals a relationship between inefficient quanta and efficient-and-satisfied quanta.

Lemma 2.17 *Suppose that* A-STEAL *schedules a job on a machine with P processors. If ρ is* A-STEAL*'s responsiveness parameter, and the schedule produces m inefficient quanta, then it produces at most $m + \log_\rho P + 1$ efficient-and-satisfied quanta.*

PROOF. Assume for the purpose of contradiction that a job's execution produces $k > m + \log_\rho P + 1$ efficient-and-satisfied quanta. Recall that the desire increases by ρ after every efficient-and-satisfied quantum, decreases by ρ after every inefficient quantum, and does not change otherwise. Thus, the total increase in desire is ρ^k, and the total decrease in desire is ρ^m. Since the desire starts at 1, the desire at the end of the job is $\rho^{k-m} > \rho^{\log_\rho P + 1} > \rho P$, contradicting Lemma 2.16. □

The following lemma bounds the number of efficient-and-satisfied quanta.

Lemma 2.18 *Suppose that* A-STEAL *schedules a job with span T_∞ on a machine with P processors. Let ρ denote* A-STEAL*'s responsiveness parameter, δ the utilization parameter, and L the quantum length. Then, for any $\epsilon > 0$, with probability at least $1 - \epsilon$, the schedule produces at most $48T_\infty/((1-\delta)L) + \log_\rho P + 16\ln(1/\epsilon)$ efficient-and-satisfied quanta.*

PROOF. The lemma follows directly from Lemmas 2.15 and 2.17. □

The next lemma shows that for each inefficient quantum there exists a corresponding efficient-and-satisfied quantum.

Lemma 2.19 *Suppose that* A-STEAL *schedules a job on a machine. Let I and C denote the set of inefficient quanta and the set of efficient-and-satisfied quanta produced by the schedule. If ρ is* A-STEAL*'s responsiveness parameter, then there exists an injective mapping $f : I \to C$ such that for all $q \in I$, we have $f(q) < q$ and $d_{f(q)} = d_q/\rho$.*

PROOF. For every inefficient quantum $q \in I$, define $r = f(q)$ to be the latest efficient-and-satisfied quantum such that $r < q$ and $d_r = d_q/\rho$. Such a quantum always exists, because the initial desire is 1 and the desire increases only after an efficient-and-satisfied quantum. We must prove that f does not map two inefficient quanta to the same efficient-and-satisfied quantum. Assume for the sake of contradiction that there exist two inefficient quanta $q < q'$ such that $f(q) = f(q') = r$. By definition of f, the quantum r is efficient-and-satisfied, $r < q < q'$, and $d_q = d_{q'} = \rho d_r$. After the inefficient quantum q, A-STEAL reduced the desire to d_q/ρ. Since the desire later increased again to $d_{q'} = d_q$ and the desire increases only after efficient-and-satisfied quanta, there must be an efficient-and-satisfied quantum r' in the range $q < r' < q'$ such that $d(r') = d(q')/\rho$. But then, by the definition of f, we would have $f(q') = r'$. Contradiction. □

We can now bound the total number of mug-cycles executed by processors.

Lemma 2.20 *Suppose that* A-STEAL *schedules a job with work T_1 on a machine with P processors. Let ρ denote* A-STEAL*'s responsiveness parameter, δ the utilization parameter, and L the quantum length. Then, the schedule produces at most $((1+\rho)/(L\delta - 1 - \rho))T_1$ mug-cycles.*

PROOF. When the allotment decreases, some processors are deallocated and their deques are declared muggable. The total number M of mug-cycles is at most the number of muggable deques during the job's execution. Since the allotment reduces by at most $a_q - 1$ from quantum q to quantum $q+1$, there are $M \le \sum_q (a_q - 1) < \sum_q a_q$ mug-cycles during the execution of the job.

By Lemma 2.19, for each inefficient quantum q, there is a distinct corresponding efficient-and-satisfied quantum $r = f(q)$ that satisfies $d_q = \rho d_r$. By definition, each efficient-and-satisfied quantum r has a nonsteal usage $n_r \ge L\delta a_r$ and allotment $a_r = d_r$. Thus, we have $n_r + n_q \ge L\delta a_r = ((L\delta)/(1+\rho))(a_r + \rho a_r) = ((L\delta)/(1+\rho))(a_r + \rho d_r) \ge ((L\delta)/(1+\rho))(a_r + a_q)$, since $a_q \le d_q$ and $d_q = \rho d_r$. Except for these inefficient quanta and their corresponding efficient-and-satisfied quanta, any other quantum q is efficient, and hence $n_q \ge L\delta a_q$ for these quanta. Let $N = \sum_q n_q$ be the total number of nonsteal-cycles during the job's execution. We have $N = \sum_q n_q \ge ((L\delta)/(1+\rho)) \sum_q a_q \ge ((L\delta)/(1+\rho))M$. Since the total number of nonsteal-cycles is the sum of work-cycles and mug-cycles and the total number of work-cycles is T_1, we have $N = T_1 + M$, and hence, $T_1 = N - M \ge ((L\delta)/(1+\rho))M - M = ((L\delta - 1 - \rho)/(1+\rho))M$, which yields $M \le ((1+\rho)(L\delta - 1 - \rho))T_1$. □

Lemma 2.21 *Suppose that* A-STEAL *schedules a job with work T_1 on a machine with P processors. Let ρ denote* A-STEAL*'s responsiveness parameter, δ the utilization parameter, and L the quantum length. Then, the schedule produces at most $(T_1/(L\delta P_A))(1 + (1 + \rho)/(L\delta - 1 - \rho))$ accounted quanta, where P_A is mean availability on accounted quanta.*

PROOF. Let A and D denote the set of accounted and deductible quanta, respectively. The mean availability on accounted quanta is $P_A = (1/|A|)\sum_{q\in A} p_q$. Let N be the total number of nonsteal-cycles. By definition of accounted quanta, the nonsteal usage satisfies $n_q \geq L\delta a_q$. Thus, we have $N = \sum_{q\in A\cup D} n_q \geq \sum_{q\in A} n_q \geq \sum_{q\in A} \delta L p_q = \delta L |A| P_A$, and hence, we obtain

$$|A| \leq N/(L\delta P_A) . \tag{2.1}$$

The total number of nonsteal-cycles is the sum of the number of work-cycles and mug-cycles. Since there are at most T_1 work-cycles on accounted quanta and, by Lemma 2.20, there are at most $M \leq ((1+\rho)(L\delta - 1 - \rho))T_1$ mug-cycles, we have $N \leq T_1 + M < T_1(1 + (1+\rho)/(L\delta - 1 - \rho))$. Substituting this bound on N into Inequality (2.1) completes the proof. □

We are now ready to bound the running time of jobs scheduled with A-STEAL.

Theorem 2.22 *Suppose that* A-STEAL *schedules a job with work T_1 and span T_∞ on a machine with P processors. Let ρ denote* A-STEAL*'s responsiveness parameter, δ the utilization parameter, and L the quantum length. For any $\epsilon > 0$, with probability at least $1 - \epsilon$,* A-STEAL *completes the job in*

$$T = \frac{T_1}{\delta \widetilde{P}} \left(1 + \frac{1+\rho}{L\delta - 1 - \rho}\right) + O\left(\frac{T_\infty}{1-\delta} + L\log_\rho P + L\ln(1/\epsilon)\right) \tag{2.2}$$

time steps, where $\widetilde{P}$ is the $O(T_\infty/(1-\delta) + L\log_\rho P + L\ln(1/\epsilon))$-trimmed availability.

PROOF. The proof is a trim analysis. Let A be the set of accounted quanta, and let D be the set of deductible quanta. The overall number of time steps is thus at most $L(|A| + |D|)$. Lemmas 2.15 and 2.18 show that there are at most $|D| = O(T_\infty/((1-\delta)L) + \log_\rho P + \ln(1/\epsilon))$ deductible quanta with high probability, since efficient-and-satisfied quanta and inefficient quanta are deductible. Hence there are at most $L|D| = O(T_\infty/(1-\delta) + L\log_\rho P + L\ln(1/\epsilon))$ time steps in deductible quanta with high probability. We have that $P_A \geq \widetilde{P}$, since the mean availability on the accounted time steps (we trim the $L|D|$ deductible steps) must be at least the $O(T_\infty/(1-\delta) + L\log_\rho P + L\ln(1/\epsilon))$-trimmed availability (we trim the $O(T_\infty/(1-\delta) + L\log_\rho P + L\ln(1/\epsilon))$ steps that have the highest availability). From Lemma 2.21, the number of accounted quanta is bounded by $|A| = (T_1/(L\delta P_A))(1 + (1+\rho)/(L\delta - 1 - \rho))$. Combining the two parts, the desired time bound follows. □

Corollary 2.23 *Suppose that* A-STEAL *schedules a job with work T_1 and span T_∞ on a machine with P processors. Let ρ denote* A-STEAL*'s responsiveness parameter, δ the utilization parameter, and L the quantum length. Then,* A-STEAL *completes the job in expected time $\mathrm{E}[T] = O(T_1/\widetilde{P} + T_\infty + L\lg P)$ where $\widetilde{P}$ is the $O(T_\infty + L\lg P)$-trimmed availability.*

PROOF. With probability $(1-\epsilon)$, A-GREEDY completes the job in time

$$T_\epsilon = \frac{T_1}{\delta\widetilde{P}}\left(1+\frac{1+\rho}{L\delta-1-\rho}\right)+O\left(\frac{T_\infty}{1-\delta}+L\log_\rho P+L\ln(1/\epsilon)\right) \tag{2.3}$$

$$\tag{2.4}$$

for some constant C. In order to get the expectation, we assume that the maximum completion time is T_1 if the job runs entirely sequentially.

$$\begin{aligned} E[T] &= (1-\epsilon)T_\epsilon+\epsilon T_1 \\ &= (1-\epsilon)\frac{T_1}{\delta\widetilde{P}}\left(1+\frac{1+\rho}{L\delta-1-\rho}\right)+O\left(\frac{T_\infty}{1-\delta}+L\log_\rho P+L\ln(1/\epsilon)\right)+\epsilon T_1 \end{aligned}$$

Substitute $\epsilon = 1/P$ and assume that δ and ρ are constants.

$$\begin{aligned} E[T] &= (1-1/P)O\left(\frac{T_1}{\widetilde{P}}+T_\infty+L\ln P+L\ln(P)\right)+1/PT_1 \\ &= O\left(\frac{T_1}{\widetilde{P}}+T_\infty+L\ln P+\right) \end{aligned}$$

□

The analysis leading to Theorem 2.22 and its corollary makes two assumptions. First, we assume that the scheduler knows exactly how many steal-cycles have occurred in the quantum. Second, we assume that the processors can find the muggable deques instantaneously. We now relax these assumptions and show that they do not adversely affect the asymptotic running time of A-STEAL.

A scheduling system can implement the counting of steal-cycles in several ways that impact our theoretical bounds only minimally. For example, if the number of processors in the machine P is smaller than the quantum length L, then the system can designate one processor to collect all the information from the other processors at the end of each quantum. Collecting this information increases the time bound by a multiplicative factor of only $1 + P/L$. As a practical matter, one would expect that $P \ll L$, since scheduling quanta tend to be measured in tens of milliseconds and processor cycle times in nanoseconds or less, and thus the slowdown would be negligible. Alternatively, one might organize the processors for the job into a tree structure so that it takes $O(\lg P)$ time to collect the total number of steal-cycles at the end of each quantum. The tree implementation introduces a multiplicative factor of $1 + (\lg P)/L$ to the job's execution time, an even less-significant overhead.

The second assumption, that it takes constant time to find a muggable deque, can be relaxed in a similar manner. One option is to mug serially, that is, while there is a muggable deque, all processors try to mug according to a fixed linear order. This strategy could increase the number of mug-cycles by a factor of P in the worst case. If $P \ll L$, however, this change again does not affect the running time bound by much. Alternatively, to obtain a better theoretical bound, we could use a counting network [AHS94] with width P to implement the list of muggable deques, in which case each mugging operation would consume $O(\lg^2 P)$ processor cycles. The number of accounted steps in the time bound from Lemma 2.21 would increase slightly to $(T_1/(\delta\widetilde{P}))/\left(1+(1+\rho)\lg^2 P/(L\delta-1-\rho)\right)$, but the number of deductible steps would not change.

Waste analysis

The next theorem bounds the waste, which is the total number of mug- and steal-cycles.

Theorem 2.24 *Suppose that* A-STEAL *schedules a job with work T_1 on a machine with P processors. Let ρ denote* A-STEAL*'s responsiveness parameter, δ the utilization parameter, and L the quantum length. Then,* A-STEAL *wastes at most*

$$W \leq \left(\frac{1+\rho-\delta}{\delta} + \frac{(1+\rho)^2}{\delta(L\delta - 1 - \rho)}\right) T_1 \tag{2.5}$$

processor cycles in the course of computation.

PROOF. Let M be the total number of mug-cycles, and let S be the total number of steal-cycles. Hence, we have $W = S + M$. Since Lemma 2.20 bounds M, we only need to bound S, which we do using an accounting argument based on whether a quantum is inefficient or efficient. Let S_{ineff} and S_{eff}, where $S = S_{\text{ineff}} + S_{\text{eff}}$, be the numbers of steal-cycles on inefficient and efficient quanta, respectively.

Inefficient quanta Lemma 2.19 shows that every inefficient quantum q with desire d_q has a distinct corresponding efficient-and-satisfied quantum $r = f(q)$ with desire $d_r = d_q/\rho$. Thus, the steal-cycles on quantum q can be amortized against the nonsteal-cycles on quantum r. Since quantum r is efficient-and-satisfied, its nonsteal usage satisfies $n_r \geq L\delta a_q/\rho$ and its allocation is $a_r = d_r$. Therefore, we have $n_r \geq L\delta a_r = L\delta d_r = L\delta d_q/\rho \geq L\delta a_q/\rho$. Let s_q be the number of steal-cycles in quantum q. Since there are La_q processor cycles in the quantum, we have $s_q \leq La_q \leq \rho n_r/\delta$, that is, the number of steal-cycles in the inefficient quantum q is at most a ρ/δ fraction of the nonsteal-cycles in its corresponding efficient-and-satisfied quantum r. Therefore, the total number of steal-cycles in all inefficient quanta satisfies $S_{\text{ineff}} \leq (\rho/\delta)(T_1 + M)$.

Efficient quanta On any efficient quantum q, the job has at least $L\delta a_q$ work- and mug-cycles and at most $L(1-\delta)a_q$ steal-cycles. Summing over all efficient quanta, the number of steal-cycles on efficient quanta is $S_{\text{eff}} \leq ((1-\delta)/\delta)(T_1 + M)$.

The total waste is therefore $W = S + M = S_{\text{ineff}} + S_{\text{eff}} + M \leq (T_1 + M)(1 + \rho - \delta)/\delta + M$. Since Lemma 2.20 provides $M < T_1(1+\rho)/(L\delta - 1 - \rho)$, the theorem follows. □

Interpretation of the bounds

If the utilization parameter δ and responsiveness parameter ρ are constants, the bounds in Equation (2.2) and Inequality (2.5) can be simplified somewhat as follows:

$$\begin{aligned} T &= \frac{T_1}{\delta\widetilde{P}}(1 + O(1/L)) + O\left(\frac{T_\infty}{1-\delta} + L\log_\rho P + L\ln(1/\epsilon)\right) , \\ W &= \left(\frac{1+\rho-\delta}{\delta} + O(1/L)\right) T_1 . \end{aligned} \tag{2.6}$$

This reformulation allows us to more easily see the trade-offs due to the setting of the δ and ρ parameters.

In the time bound, as δ increases toward 1, the coefficient of $T_1/\widetilde{P}$ decreases toward 1, and the job comes closer to perfect linear speedup on accounted steps. The number of deductible steps increases at the same time, however. Moreover, as δ increases and ρ decreases, the completion time increases and the waste decreases. The utilization parameter δ may lie between 80% and 95%, and the responsiveness parameter ρ can be set between 1.2 and 2.0. The quantum length L is a system configuration parameter, which might have values in the range 10^3 to 10^5.

To see how these settings affect the waste bound, consider the waste bound as comprising two parts, where the waste due to steal-cycles is $S \leq ((1+\rho-\delta)/\delta)T_1$ and the waste due to mug-cycles is $M = O(1/L)T_1$. We can see that the waste due to mug-cycles is just a tiny fraction compared to the work T_1. Thus, these bounds indicate that adaptive scheduling with parallelism feedback can be achieved without imposing much overhead when adding to or removing processors from jobs.

The major part of waste comes from steal-cycles, where S is generally less than $2T_1$ for typical parameter values. The analysis of Theorem 2.24 shows, however, that the number of steal-cycles in efficient steps is bounded by $((1-\delta)/\delta)T_1$, which is a small fraction of S. Thus, most of the waste comes from the steal-cycles in inefficient quanta. Our analysis assumes that the job scheduler is an adversary, creating as many inefficient quanta as possible. Of course, job schedulers are generally not adversarial. Thus, in practice, we expect the waste to be a much smaller fraction of T_1 than our bounds. Chapter 3 describes experiments that confirm this intuition.

2.8 Related Work

This section discusses related work on adaptive scheduling of parallel jobs. There has been a large amount of work on nonadaptive thread scheduling, both using greedy scheduling and work stealing. Work in the area of adaptive scheduling has generally centered on job schedulers, some of which use "dynamic equipartitioning" as a strategy for allotting processors to jobs.

Greedy scheduling is an old scheduling strategy, first introduced by Graham [Gra69], and later independently invented by Brent [Bre74]. Some version of it has since been implemented in many data parallel languages such as Fortran (HPF) [For93], NESL [BCH+94, BG96], and ZPL [CCL+00]. Of particular interest is the work by Blelloch and his coauthors [BG96, NB99, BGM99] which provides various nonadaptive task schedulers for a generalized class of data-parallel jobs, called ***nested*** data-parallel jobs. Specifically, their "prioritized" task schedulers are provably efficient with respect to both time and space. A-GREEDY can be combined with prioritized task schedulers to produce adaptive task schedulers that are provably efficient with respect to time, space, and waste.

Work-stealing has been used as a heuristic since Burton and Sleep's research [BS81] and Halstead's implementation of Multilisp [Hal84]. Many variants have been implemented since then [MKHJ90, HZJ94, FM87], and it has been analyzed in the context of load balancing [RSAU91], backtrack search [KZ88], etc. Blumofe and Leiserson [BL99] proved that the work-stealing algorithm is efficient with respect to time, space, and communication for the class of "fully strict" multithreaded computations. Arora, Blumofe and Plaxton [ABP98] extended the time bound result to arbitrary multithreaded computations. Various researchers [HLMS06, HS02, CL05] have since simplified and improved the memory allocation for deques for ABP. In addition, Acar, Blelloch, and Blumofe [ABB00] showed that work-stealing schedulers are efficient with respect to cache misses for jobs with "nested parallelism." Variants of work-stealing algorithms have been imple-

mented in many systems [BJK+95, FLR98, BP99], and empirical studies show that work-stealing schedulers are scalable and practical [FLR98, BP98b].

Adaptive thread scheduling without parallelism feedback has been studied in the context of multithreading, primarily by Blumofe and his coauthors [BL97, BP94, ABP98]. In this work, the thread scheduler schedules threads using a "work-stealing" [MKHJ90, BL99] strategy, but it does not provide the feedback about the job's parallelism to the job scheduler. The work in [BL97, BP94] addresses networks of workstations where processors may fail or join and leave a computation while the job is running, showing that work-stealing provides a good foundation for adaptive thread scheduling. In theoretical work, Arora, Blumofe, and Plaxton [ABP98] exhibit a work-stealing thread scheduler that provably completes a job in $O(T_1/\overline{P} + PT_\infty/\overline{P})$ expected time, where $\overline{P}$ is the average number of processor allotted to the job by the job scheduler. Although they provide no bounds on waste, one can prove that their algorithm may waste $\Omega(PT_\infty)$ processor cycles in an adversarial setting.

Adaptive job schedulers have been studied empirically [MVZ93, YL01, TG89, LV90, MEB88, NSS93] and theoretically [Gu95, DD96, MPT93, Edm99, ECBD03, BDKS04]. McCann, Vaswani, and Zahorjan [MVZ93] studied many different job schedulers and evaluated them on a set of benchmarks. They also introduced the notion of dynamic equipartitioning, which gives each job a fair allotment of processors, while allowing processors that cannot be used by a job to be reallocated to other jobs. Their studies indicate that dynamic equipartitioning may be an effective strategy for adaptive job scheduling. Gu [Gu95] proved that dynamic equipartitioning with instantaneous parallelism feedback is 4-competitive with respect to makespan for batched jobs with multiple phases, where the parallelism of the job remains constant during the phase and the phases are relatively long compared with the length of a scheduling quantum. Deng and Dymond [DD96] proved a similar result for mean response time for multiphase jobs regardless of their arrival times. Song [Son98] proves that a randomized distributed strategy can implement dynamic equipartitioning.

Subsequent to the work described in this chapter, my collaborators extended this work [HHL06, HHL07] analyzing the performance of A-GREEDY or A-STEAL when combined with dynamic equipartitioning and roundrobin job schedulers. They find that a combination of A-GREEDY and A-STEAL with these "nice" job schedulers yields makespan that is constant competitive with the optimal. In addition, if all the jobs arrive at the same time, then these combinations also lead to constant-competitive mean completion time.

Chapter 3

Experimental Evaluation of Adaptive Work Stealing with Parallelism Feedback

In the previous chapter, we described A-STEAL, a work stealing algorithm for adaptive scheduling with parallelism feedback. In this chapter, we shall evaluate A-STEAL experimentally. Our studies monitored the behavior of A-STEAL on a simulated multiprocessor system using synthetic workloads. Our experiments indicate that A-STEAL provides good performance on moderately to heavily loaded large machines. In addition, we compared A-STEAL with an adaptive scheduler that does not provide parallelism feedback[ABP98], and A-STEAL compares favorably in performance.

This chapter is organized as follows: Section 3.1 provides the summary of our experiments, Section 3.2 explains the experimental setup, and Sections 3.3, 3.4, 3.5, and 3.6 provide details about four sets of experiments.

3.1 Summary of Experiments

To evaluate the performance of A-STEAL empirically, we built a discrete-time simulator using DESMO-J [DES99]. Some of our experiments benchmarked A-STEAL against ABP [ABP98], an adaptive thread scheduler by Arora et al. that does not supply parallelism feedback to the job scheduler. This section describes our simulation setup and the results of the experiments.

We conducted four sets of experiments on the simulator with synthetic jobs. Our results are summarized below:

- The ***time experiments*** investigated the performance of A-STEAL on over 2300 job runs. A linear-regression analysis of the results provides evidence that the coefficients on the number of accounted and deductible steps are considerably smaller than the upper bounds provided by our theoretical bounds. A second linear-regression analysis indicates that A-STEAL completes jobs on average for at most twice the optimal number of time steps, which is the same bound provided by offline greedy scheduling [Gra69, Bre74].

- The ***waste experiments*** are designed to measure the waste incurred by A-STEAL in practice and compare the observed waste to the theoretical upper bounds. Our experiments indicate

This is joint work with Yuxiong He and Charles E. Leiserson [AHL06].

that the waste is almost insensitive to the parameter settings and is a tiny fraction (less than 10%) of the work for jobs with high parallelism.

- The ***time-waste experiments*** compare the completion time and waste of A-STEAL with ABP [ABP98] by running single jobs with predetermined availability profiles. These experiments indicate that on large machines, when the mean availability $\overline{P}$ is considerably smaller than the number P of processors in the machine, A-STEAL completes jobs faster than ABP while wasting fewer processor cycles than ABP. On medium-sized machines, when $\overline{P}$ is of the same order as P, ABP completes jobs slightly faster than A-STEAL, but it still wastes many more processor cycles than A-STEAL.

- The ***utilization experiments*** compare the utilization of A-STEAL and ABP when many jobs with varying characteristics are using the same multiprocessor resource. The experiments provide evidence that on moderately to heavily loaded large machines, A-STEAL consistently provides a higher utilization than ABP for a variety of job mixes.

3.2 Simulation Setup

Our Java-based discrete-time simulator, which was implemented using DESMO-J [DES99], implements four major entities — processors, jobs, thread schedulers, and job schedulers. The simulator tracks their interactions in a two-level scheduling environment. We modeled jobs as dags, which are executed by the thread scheduler. When a job is submitted to the simulated multiprocessor system, an instance of a thread scheduler is created for the job. The job scheduler allots processors to the job, and the thread scheduler simulates the execution of the job using work-stealing. The simulator operates in discrete time steps: a processor can complete either a work-cycle, steal-cycle, or mug-cycle during each time step. We ignored the overheads due to the reallocation of processors in the simulation.

We tested synthetic multithreaded jobs with the parallelism profile shown in Figure 3.1. Each job alternates between a serial phase of length w_1 and a parallel phase (with h-way parallelism) of length w_2. The average parallelism of the job is approximately $(w_1 + hw_2)/(w_1 + w_2)$. By varying the values of w_1, w_2, h, and the number of iterations, we can generate jobs with different work, span, and frequency of the change of the parallelism.

In the time-waste experiments and the utilization experiments, we compared the performance of A-STEAL with that of another thread scheduler, ABP [ABP98], an adaptive thread scheduler that does not provide parallelism feedback to the job scheduler. In these experiments, ABP is always allotted all the processors available to the job. ABP uses a nonblocking implementation of work-stealing and always maintains P deques. When the job scheduler allots $a_q = p_q$ processors in quantum q, ABP selects a_q deques uniformly at random from the P deques, and the allotted processors start working on them. Arora, Blumofe, and Plaxton [ABP98] prove that ABP completes a job in expected time

$$T = O(T_1/\overline{P} + PT_\infty/\overline{P}) , \tag{3.1}$$

where $\overline{P}$ is the average number of processors allotted to the job by the job scheduler.

Although Arora *et al.* provide no bounds on waste, one can prove that ABP may waste $\Omega(PT_\infty)$ processor cycles in an adversarial setting. For a job which is completely sequential, we have $T_1 =$

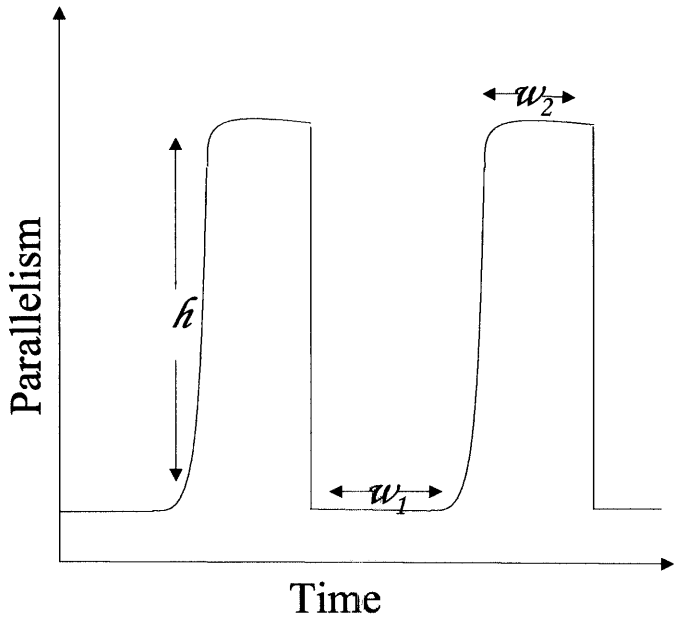

Figure 3.1: The parallelism profile (for 2 iterations) of the jobs used in the simulation.

$T_\infty \leq PT_\infty$. A job scheduler may allot all P processor cycles to this job. Therefore, the total number of processor cycles allotted to the job is PT_∞, since the job completes in T_∞ time. The job uses only $T_1 = T_\infty$ processor cycles. Therefore, ABP wastes $O(PT_\infty)$ processor cycles.

We implemented three kinds of job schedulers: profile-based, equipartitioning [MVZ93], and dynamic equipartitioning [MVZ93]. A profile-based job scheduler was used in the first four sets of experiments, and both equipartitioning and dynamic equipartitioning job schedulers were used in the utilization experiment. An ***equipartitioning*** (EQ) job scheduler simply allots the same number of processors to all the active jobs in the system. Since ABP provides no parallelism feedback, EQ is a suitable job scheduler for ABP's scheduling model. ***Dynamic equipartitioning*** (DEQ) is a dynamic version of the equipartitioning policy, but it requires parallelism feedback. A DEQ job scheduler maintains an equal allotment of processors to all jobs with the constraint that no job is allotted more processors than it requests. DEQ is compatible with A-STEAL's scheduling model, since it can use the feedback provided by A-STEAL to decide the allotment.

For the first three experiments — time, waste, and time-waste — we ran a single job with a predetermined ***availability profile***: the sequence of processor availabilities p_q for all the quanta while the job is executing. For the ***profile-based*** job scheduler, we precomputed the availability profile, and during the simulation, the job scheduler simply used the precomputed availability for each quantum. We generated three kinds of profiles:

- **Uniform profiles:** The processor availabilities in these profiles follow the uniform distribution in the range from 1 to the maximum number P of processors in the system. These profiles represent near-adversarial conditions for A-STEAL, because the availability for one quantum is unrelated to the availability for the previous quantum.

- **Smooth profiles:** In these profiles, the change of processor availabilities from one scheduling quantum to the next follows a standard normal distribution. Thus, the processor availability is unlikely to change significantly over two consecutive quanta. These profiles attempt to model situations where new arrivals of jobs are rare, and the availability changes significantly only when a new job arrives.

- **Practical profiles:** These availability profiles were generated from the workload archives [Fei] of various computer clusters. We computed the availability at every quantum by subtracting the number of processors that were being used at the start of the quantum from the number of processors in the machine. These profiles are meant to capture the processor availability in practical systems.

A-STEAL requires certain parameters as input. The responsiveness parameter is $\rho = 1.5$ for all the experiments. For all experiments except the waste experiments, the utilization parameter is $\delta = 0.8$. We varied δ in the waste experiments. The quantum length L represents the time between successive reallocations of processors by the job scheduler and is selected to amortize the overheads due to communication between the job scheduler and the thread scheduler and to the reallocation of processors. In conventional computer systems, a scheduling quantum is typically between 10 and 20 milliseconds. Our experience with the Cilk runtime system [Sup03] indicates that a steal/mug-cycle takes approximately 0.5 to 5 microseconds, suggesting that the quantum length L should be set to values between 10^3 and 10^5 time steps. Our theoretical bounds indicate that as long as $T_\infty \gg L \log P$, the length of L should have little effect on our results. Due to the performance limitations of our simulation environment, however, we were unable to run very long jobs: most have span in the order of only a few thousand time steps. Therefore, to satisfy the condition that $T_\infty \gg L \log P$, we set $L = 200$.

3.3 Time Experiments

The running-time bounds proved in Chapter 2, Section 2.7, though asymptotically strong, have weak constants. The time experiments were designed to investigate what constants would occur in practice and how A-STEAL performs compared to an optimal scheduler. We performed linear-regression analysis on the results of 2331 job runs using many availability profiles as decided earlier to answer these questions.

Our first time experiment uses the bounds in Equation (2.2) as a simple model, as in the study [BJK+96]. Assuming that equality holds and disregarding smaller terms, the model estimates performance as

$$T \approx c_1 T_1/\widetilde{P} + c_\infty T_\infty \,, \tag{3.2}$$

where $c_1 > 0$ is the ***work overhead*** and $c_\infty > 0$ is the ***span overhead***. When $\delta = 0.8$, $\rho = 1.5$, and $L = 200$, the coefficients for the asymptotic bounds in Equation (2.2) turn out to be $1.26 < c_1 < 1.27$ and $c_\infty = 480$, but a direct analysis of expectation can improve the bound on span overhead to $c_\infty = 60$. Since the span overhead c_∞ is large, the bound indicates that A-STEAL may not provide linear speedup except when $T_1/T_\infty \gg 60\widetilde{P}$. Moreover, on accounted time steps, A-STEAL might not provide perfect linear speedup, since the work overhead is $1.26 > 1$.

In practice, however, we should not expect these large overheads to materialize. First, our analysis is focused on asymptotic bounds and use bounding techniques such as Markov's inequality and

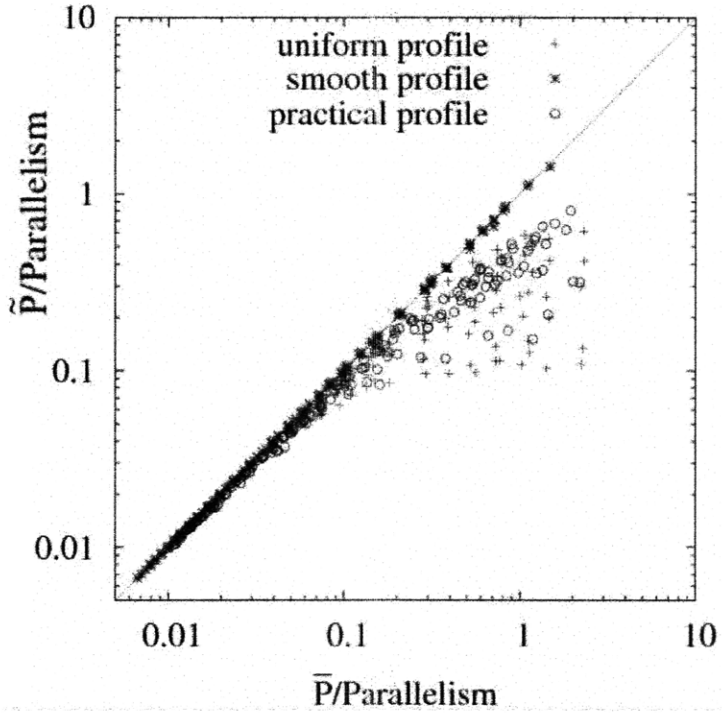

Figure 3.2: Comparing the (true) mean availability $\overline{P}$ with the trimmed availability $\widetilde{P}$ using three availability profiles. Each data point represents a job execution for which the mean availability and trimmed availability were measured. These values were normalized by dividing by the parallelism T_1/T_∞ of the job. When the parallelism satisfies $T_1/T_\infty > 5\overline{P}$, the experiments indicate that for all profiles, the trimmed availability is a good approximation of the mean availability. All these experiments used $\delta = 0.8$ and $\rho = 1.5$.

Chernoff bounds, which are not necessarily tight. Second, our analysis assumes that the job completes only the minimum number of work-cycles in each quantum, specifically, 0 on a deductible quantum and $\delta L a_q$ on an accounted quantum with allotment a_q.

Our first linear-regression analysis fits the running time of the 2331 job runs to Equation (3.2). The trimmed mean $\widetilde{P}$ of a job run is computed as the average processor availability of all accounted steps during the execution of the job. The least-squares fit to the data to minimize relative error yields $c_1 = 0.960 \pm 0.003$ and $c_\infty = 0.812 \pm 0.009$ with 95% confidence. The R^2 correlation coefficient of the fit is 99.4%. Since $c_\infty = 0.812 \pm 0.009$, on average the jobs achieved linear speedup when $T_1/T_\infty \gg \widetilde{P}$. In addition, since we have $c_1 = 0.960 \pm 0.003$, A-STEAL achieves almost perfect linear speedup on the accounted steps. The fact that $c_1 < 1$ stems from the fact that jobs performed some work during the deductible steps.

We performed a second set of regression tests on the same set of jobs to compare the performance of A-STEAL with an optimal scheduler. We fit the job data to the curve

$$T = \hat{c}_1 T_1/\overline{P} + \hat{c}_\infty T_\infty \,. \tag{3.3}$$

The analysis yields $\hat{c}_1 = 0.992 \pm 0.003$ and $\hat{c}_\infty = 0.911 \pm 0.008$ with an R^2 correlation coefficient of 99.4%. Both $T_1/\overline{P}$ and T_∞ are lower bounds on the job's running time, and thus an optimal scheduler requires at least $\max\{T_1/\overline{P}, T_\infty\} \geq (T_1/\overline{P} + T_\infty)/2 \geq (\hat{c}_1 T_1/\overline{P} + \hat{c}_\infty T_\infty)/2$ time steps, since $\hat{c}_1 < 1$ and $\hat{c}_\infty < 1$. Consequently, on average A-STEAL completed the jobs within at most twice the time of an optimal scheduler.

The Equations (3.2) and (3.3) both predict performance with high accuracy, and yet $\widetilde{P}$ and $\overline{P}$ can diverge significantly. To resolve this paradox, we compared $\widetilde{P}$ and $\overline{P}$ on the job runs. Figure 3.2 shows a graph of the results, where $\widetilde{P}$ and $\overline{P}$ are each normalized by dividing by the parallelism T_1/T_∞ of the job. The diagonal line in the figure is the curve $\widetilde{P} = \overline{P}$.

If a job has parallelism $T_1/T_\infty > 5\overline{P}$ (data points on the left), the experiment indicates that for all three kinds of availability profiles, we have $\widetilde{P} \approx \overline{P}$. In this case, we have $T_1/\widetilde{P} \approx T_1/\overline{P}$ and $T_1/\overline{P} \gg T_\infty$, which implies that the first terms in Equations (3.2) and (3.3) are nearly identical and dominate the running time. On the other hand, if a job has small parallelism (data points on the right), the values of $\widetilde{P}$ and $\overline{P}$ diverge and the divergence depends on the availability profile used. In this region, however, the running time is dominated by the span T_∞, and thus, the divergence of $\widetilde{P}$ and $\overline{P}$ has little influence on the running time.

3.4 Waste Experiments

Our theoretical analysis shows that the waste exhibited by A-STEAL is at most $O(T_1)$. The constant hidden in the O-notation depends on the parameter settings. In our first waste experiment, we varied the value of the utilization parameter δ to determine the relationship between the waste and the setting of δ. For our second experiment, we investigated whether the waste incurred by a job depends on the job's parallelism.

The proof of Theorem 2.6 shows that the number of processor cycles wasted by a job is $((1-\delta)/\delta)T_1$ on efficient quanta and approximately $(\rho/\delta)T_1$ on inefficient quanta. Substituting $\delta = 0.8$ and $\rho = 1.5$, A-STEAL could waste as many as $0.25T_1$ processor cycles on efficient quanta and as many as $1.875T_1$ processor cycles on inefficient quanta. Since this analysis assumes that the job

scheduler is an adversary and that the job completes the minimum number of work-cycles in each quantum, we did not expect these constants to materialize in practice.

We measured the waste for 300 jobs, most of which had parallelism $T_1/T_\infty > 5\overline{P}$, for $\delta = 0.5, 0.6, \ldots, 1.0$. The job runs used many availability profiles drawn equally from the three kinds. Figure 3.3 shows the average of waste normalized by the work T_1 of the job. For comparison we plotted the normalized theoretical bound Inequality (2.6) for the total waste and the normalized bound $((1-\delta)/\delta)T_1$ for the waste on efficient quanta. As the figure shows (although the curve is barely distinguishable from the x-axis), the observed waste is less than 10% of the work T_1 for most values of δ and is considerably less than what the theoretical bounds predicted. Moreover, the waste seems to be quite insensitive to the particular value of δ.

We also ran an experiment to determine whether parallelism has an effect on waste. The bound in Inequality (2.6) does not depend on the parallelism T_1/T_∞ of the job, but only on the work T_1. For the 2331 job runs used in the time experiments, we measured the waste versus parallelism. Since waste is insensitive to δ, all jobs used the value $\delta = 0.8$. Figure 3.4 graphs the results. As can be seen in the figure, the higher the parallelism, the lower the waste-to-work ratio. The reason is that when the parallelism is high, the job can usually use most of the available processors without readjusting its desire. When the parallelism is low, however, the job's desire must track its parallelism closely to avoid waste. This situation is where A-STEAL is most effective, as the job pushes the theoretical waste bounds to their limit.

3.5 Time-Waste Experiments

The time-waste experiments were designed to compare A-STEAL with ABP, an adaptive thread scheduler with no parallelism feedback. For our first experiment, we ran A-STEAL and ABP to execute 756 job runs on a simulated machine with $P = 512$ processors. Each head-to-head run used one of two practical availability profiles, one with $\overline{P} = 30$ and one with $\overline{P} = 60$. We measured the time and waste of A-STEAL and ABP for each run. Our second experiment was similar, but it used only $P = 128$ processors in the simulated machine over 330 job runs. Whenever the availability exceeded 128, which was not often, we chopped the availability to 128.

Figure 3.5 shows the ratio of ABP to A-STEAL with respect to both time and waste as a function of the mean availability $\overline{P}$, normalized by dividing by the parallelism T_1/T_∞. This experiment shows that A-STEAL completed jobs about twice as fast as ABP while wasting only about 10% of the processor cycles wasted by ABP. Not surprisingly, A-STEAL wastes fewer processor cycles than ABP, since A-STEAL uses parallelism feedback to limit possible excessive allotment. Paradoxically, however, A-STEAL completes jobs faster than ABP, even though A-STEAL's allotment in every quantum is at most that of ABP, which is always allotted all the available processors.

ABP's slow completion is due to how ABP manages its ready deques. In particular, ABP has no mechanism for increasing and decreasing the number r of ready deques, and it maintains $r = P$ deques throughout the execution. Randomized work-stealing algorithms require $\Theta(r)$ steal-cycles to reduce the length of the span by 1 in expectation. Consequently, if r is large, each steal-cycle becomes less effective, and the job's progress along its span slows. Thus, if the job has small or moderate parallelism (data points on the right), the span dominates the running time. If the job has large parallelism (data points on the left), however, the impact is less. In contrast, A-STEAL continues to make good progress along the span, regardless of parallelism, by reducing the number

of deques according to its allotment.

This paradox can also be understood by using the model from Equation (3.2) for A-STEAL and an analogous model based on Equation (3.1) for ABP. Let us consider three cases:

- $T_1/T_\infty < \overline{P} \ll P$ (data points on the right): Whereas A-STEAL completes the job in $\Theta(T_\infty)$ time, ABP requires $\Theta(PT_\infty/\overline{P})$ time.

- $\overline{P} < T_1/T_\infty \ll P$ (data points in the middle): A-STEAL provides linear speedup since $T_1/T_\infty > \widetilde{P}$, but ABP does not, since $T_1/T_\infty \ll P$.

- $P < T_1/T_\infty$ (data points on the left): Both provide linear speedup in this range.

Since ABP performed relatively poorly when P is large compared to $\overline{P}$, our second experiment investigated the case when P is closer to $\overline{P}$. Figure 3.6 shows the results on 330 job runs on a simulated machine with $P = 128$. In this case, when jobs' parallelism is large compared to $\overline{P}$, both ABP and A-STEAL perform about the same with respect to both time and waste. As the parallelism gets closer to $\overline{P}$, ABP performs slightly better than A-STEAL with respect to time and slightly worse with respect to waste. Since $\overline{P} \approx P$, the two models coincide, and ABP and A-STEAL perform comparably. Therefore, on small machines, where the disparity between $\overline{P}$ and P cannot be very great, the advantage of parallelism feedback is diminished, and ABP may yet be an effective thread-scheduling algorithm.

3.6 Utilization Experiments

The utilization experiments compared A-STEAL with ABP on a large server where many jobs are running simultaneously and jobs arrive and leave dynamically. We implemented job schedulers to allocate processors among various jobs: dynamic equipartitioning [MVZ93] for A-STEAL and equipartitioning [TG89] for ABP. We simulated a 1000-processor machine for about 10^6 time steps, where jobs had a mean interarrival time of 1000 time steps. We compared the utilization provided by A-STEAL and ABP over time.

It was unclear to us what distribution the parallelism and the span should follow. Although many workload models for parallel jobs have been studied [Sev94, Fei96, Dow98, CB01, LF03], none appears to apply directly to multithreaded jobs. Some studies [LO86, HBD97, HB99] claim that the sizes of Unix jobs follow a heavy-tailed distribution. Lacking a well-recognized guideline, we decided to try various distributions, and as it turned out, our results were fairly insensitive to which we chose.

We considered 9 sets of jobs using three distributions on each of the parallelism and the span. The means of the distributions were chosen so that jobs arrive faster than they complete and the load on the machine progressively increases. Thus, we were able to measure the utilization of the machine under various loads. The three distributions we explored were the following:

- **Uniform distribution (U):** The span is picked uniformly from the range $1,000$ to $99,000$. The parallelism is generated uniformly in the range $[1, 80]$.

- **Heavy-tailed distribution 1 (HT1):** We used a Zipf's-like [Zip49] heavy-tailed distribution where the probability of generating x is proportional to $1/x$. In our experiments, the

distribution for parallelism has mean value 36, and the distribution for span has mean value 50,000.

- **Heavy-tailed distribution 2 (HT2):** In this distribution, the probability of generating x is proportional to $1/\sqrt{x}$. In our experiments, the distribution for parallelism has mean value 36, and the distribution for span has mean value 50,000.

Of the 9 possible sets of jobs, we ran 6 experiments using parallelism and span drawn from U/U, U/HT1, HT1/U, HT1/HT1, HT2/U, and HT2/HT2. For all these experiments, the comparison between A-STEAL+DEQ and ABP+EQ followed the same qualitative trends. We broke time into intervals of 2000 time steps and measured the utilization — the fraction of processor cycles spent working — for each interval. Figure 3.7 shows the utilization as a function of time (log-scale) for the U/U experiment at the top and for HT1/HT1 on the bottom. As can be seen in both figures, ABP+EQ starts out with a higher utilization, since A-STEAL+DEQ initially requests just one processor. Before 10% of the simulation has elapsed, however, A-STEAL+DEQ overtakes ABP+EQ with respect to the utilization and then consistently provides a higher utilization. Although the figure does not show it, the mean completion time of jobs under ABP+EQ is nearly 50% less than those under A-STEAL+DEQ for both these distributions.

3.7 Related Work

In this section, we mention some of the related work not already touched upon in Chapter 2.

The paper most related to this work is one by Arora, Blumofe and Plaxton [ABP98] which compared A-STEAL against. Adaptive task scheduling without parallelism feedback has also been studied empirically in the context of data-parallel languages [EAS+95, EASS94]. This work focuses on compiler and runtime support for environments where the number of processors changes while the program executes. Adaptive task scheduling with parallelism feedback has been studied empirically in [TBB96, Son98, Sen04]. These researchers use a job's history of processor utilization to provide feedback to dynamic-equipartitioning job schedulers. These studies use different strategies for parallelism feedback, and all report better system performance with parallelism feedback than without, but it is not apparent which strategy is superior.

As we mentioned in Chapter 2, my collaborators extended this work [HHL06, HHL07] analyzing the performance of A-GREEDY or A-STEAL when combined with dynamic equipartitioning and roundrobin job schedulers. In addition to the theoretical results, they also performed simulation studies using the same simulation environment described in this chapter. They find that the performance of a concurrency platform that combines a thread scheduler using A-STEAL with a job scheduler that uses dynamic equipartioning performs very well on a large variety of job mixes.

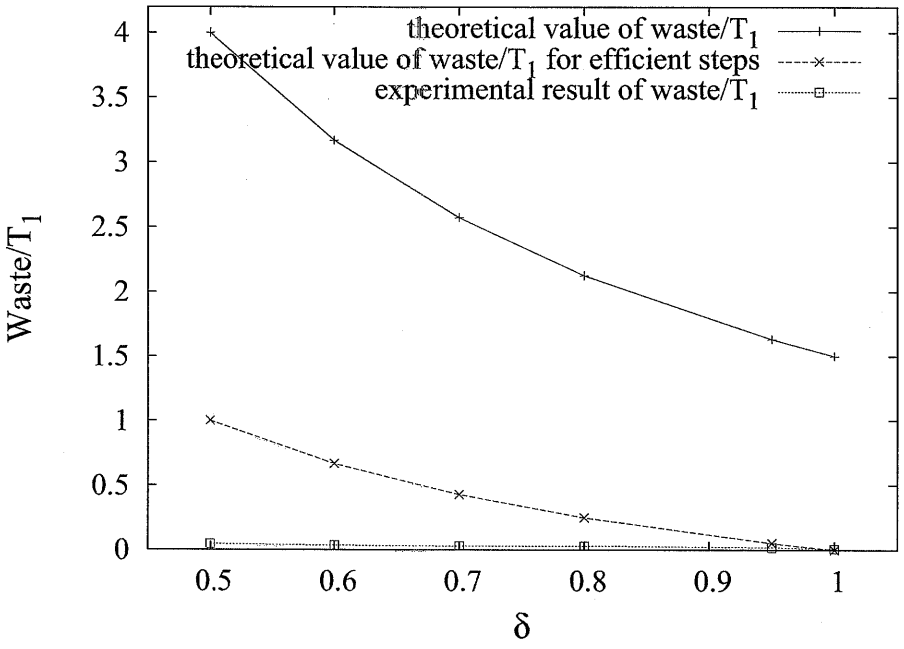

Figure 3.3: Comparing the theoretical and practical waste (normalized by T_1) using A-STEAL for various values of the utilization parameter δ. The top line shows the total theoretical waste, the next line shows the theoretical waste on efficient quanta, and the bottom line shows the observed waste. The observed waste appears to be almost insensitive to the value of δ and is much smaller than the theoretical waste.

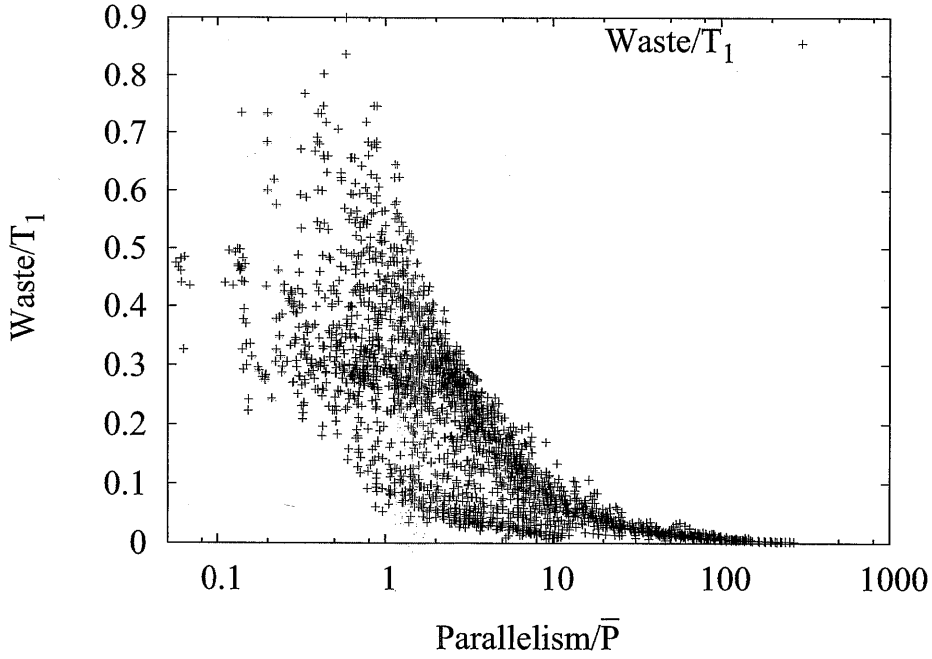

Figure 3.4: How waste varies with parallelism. When $T_1/T_\infty > 10\overline{P}$, that is, the job's parallelism significantly exceeds the average availability, the observed waste is only a tiny fraction of the work T_1. For jobs with small parallelism, the waste showed a large variance but never exceeded the work T_1 in any of our runs. The utilization parameter was $\delta = 0.8$ for all job runs.

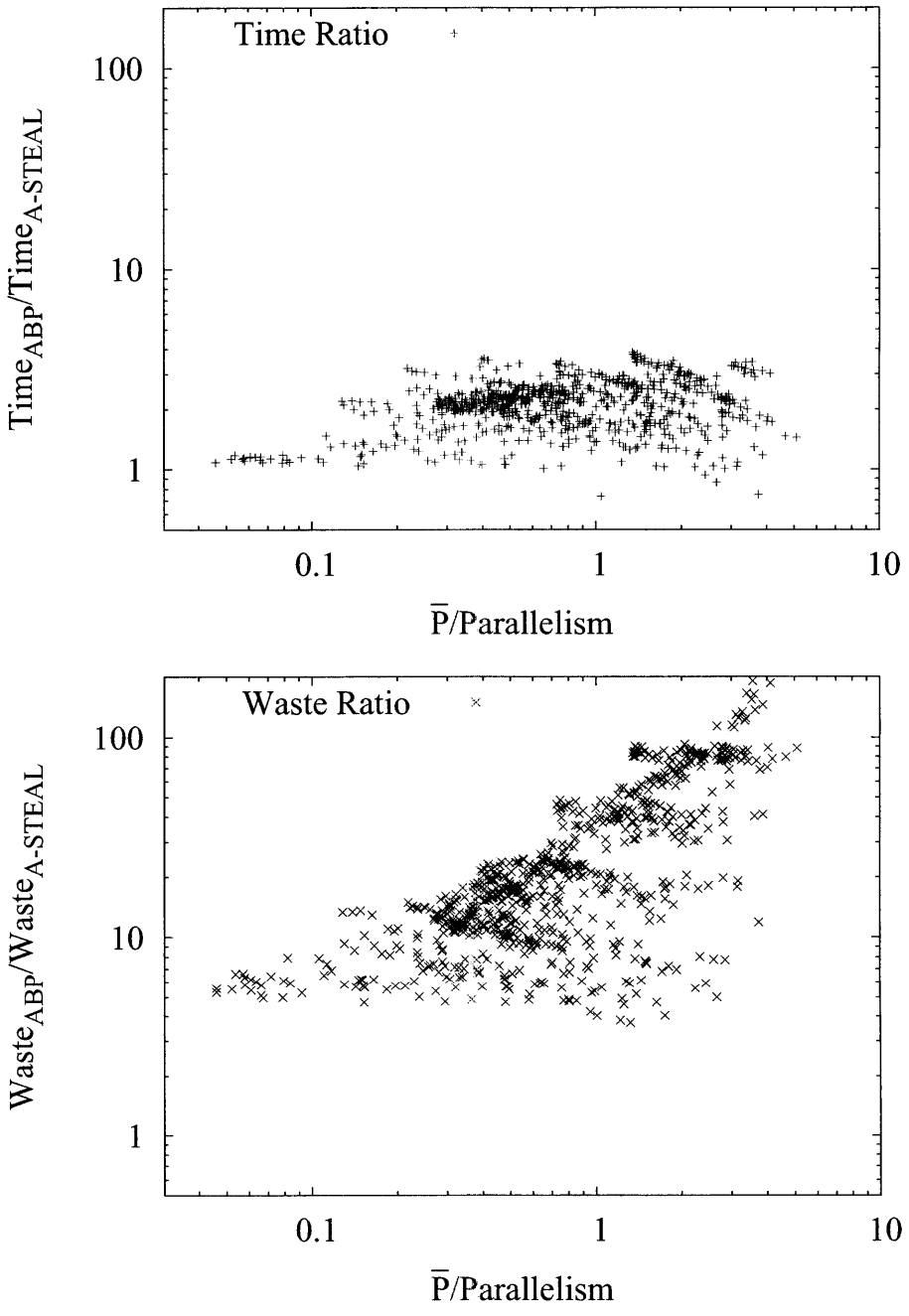

Figure 3.5: Comparing the time and waste of A-STEAL against ABP when $P = 512$ and $\overline{P} = 30, 60$. In this experiment, where P exceeds $\overline{P}$ by a significant margin, A-STEAL completes jobs about twice as fast as ABP while wasting less than 10% of the processor cycles wasted by ABP.

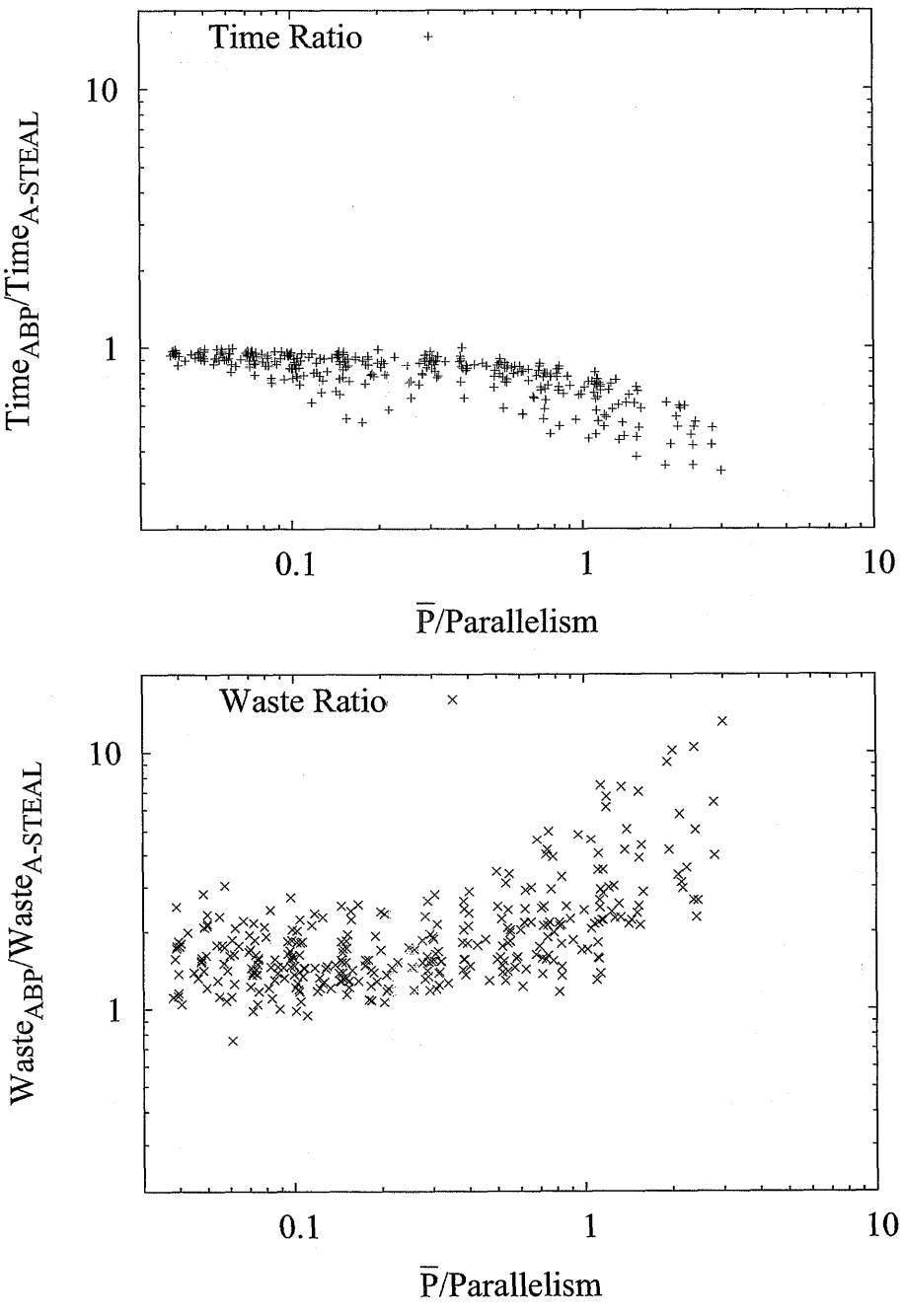

Figure 3.6: Comparing the time and waste of A-STEAL against ABP when $P = 128$ and $\overline{P} = 30, 60$. In this experiment, where P and $\overline{P}$ are closer in magnitude, A-STEAL runs slightly slower than ABP, but it still tends to waste fewer processor cycles than ABP.

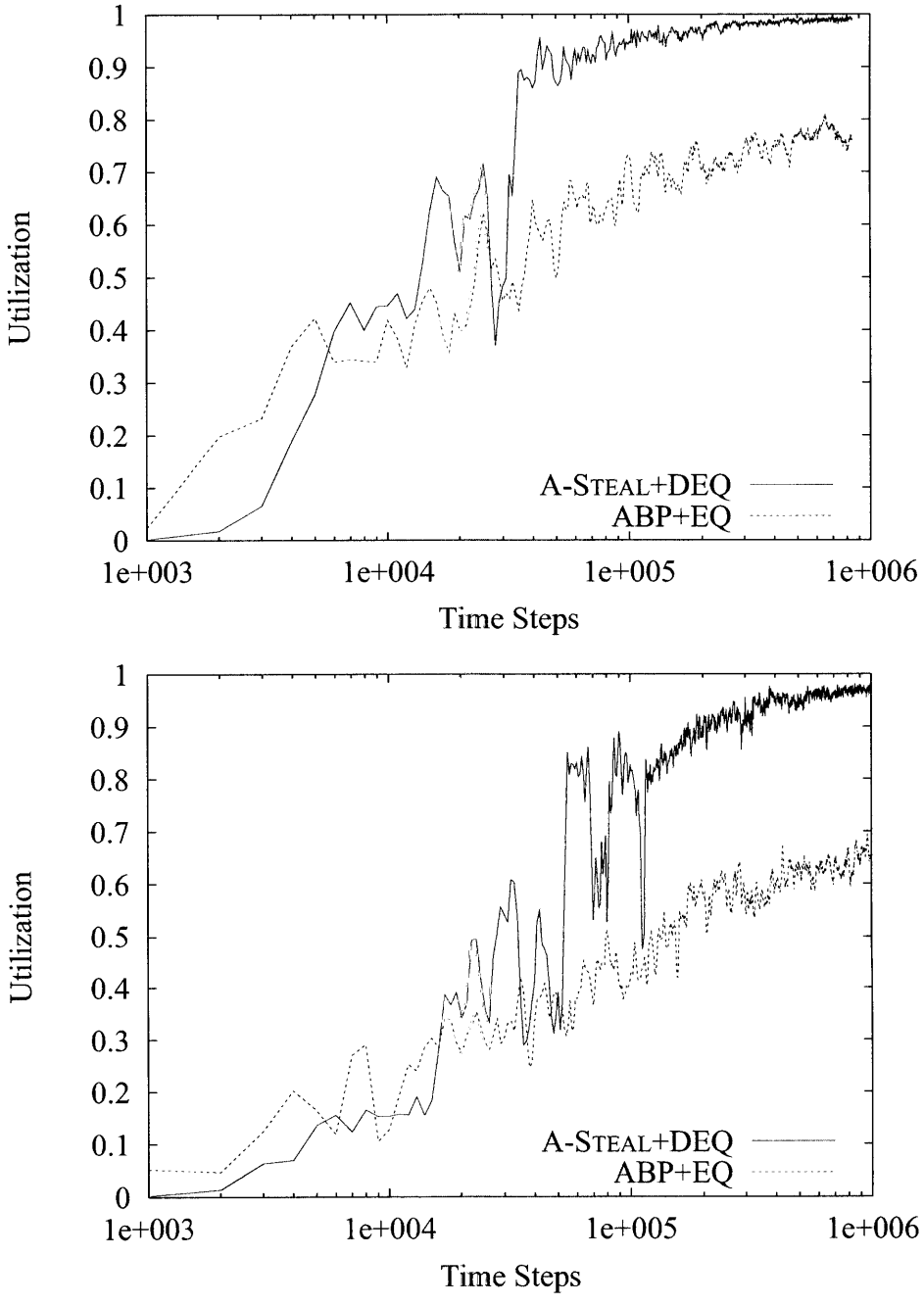

Figure 3.7: Comparing the utilization over time of A-STEAL+DEQ and ABP+EQ. In the top figure, both the span and the parallelism follow the uniform distribution, and in the bottom figure, they follow the HT1 distribution.

Chapter 4

Library for Dag Evaluations in Cilk++

Many programming problems can be formulated as computations on a directed acyclic graph, where every node represents some computation, and a directed edge represents the constraint that the predecessor's computation depends on the result of successor's computation. We say that an ***evaluation*** of a dag computation $\mathcal{D}$, or a ***dag evaluation of*** $\mathcal{D}$ for short, is the result from performing the computations for all nodes in the dag $\mathcal{D}$. Often, in order to evaluate these dag evaluations efficiently, programmers have to write their own task pool schedulers. This chapter describes a concurrency platform, called DAGEVAL, for evaluating such computations efficiently. This concurrency platform is a Cilk++ library and it allows programmers to exploit parallelism in a dag computation without writing their own task-pool schedulers.

The outline of the chapter is as follows: Section 4.1 presents some motivation for solving the problem, Section 4.2 describes the interface and the implementation of the dag evaluation library, and Section 4.3 shows the theoretical analysis of performance of this library. Sections 4.4, 4.5, and 4.6 describe the experimental setup, applications and the experiments for evaluating the performance of the library.

4.1 Motivation and Results

Many parallel computations are best described in the form of a dag $\mathcal{D} = (\mathcal{D}_V, \mathcal{D}_E)$. Each node $B \in \mathcal{D}_V$ in the dag represents some computation that depends on all the successor of the node in the dag. The result of a node can be computed when the results of all its children are available. The goal is to compute all the results. Note that this dag is slightly different from the execution dag we described in Chapter 1. For one, the dependencies go in the opposite direction: In an execution dag, each node depends on its predecessors, while in $\mathcal{D}$, each node depends on its successors. Second, in an execution dag, each node has exactly one unit of work, while in $\mathcal{D}$, each node can contain an arbitrary amount of work. The most important difference, however, is that the execution dag is a model of a parallel computation, while a dag evaluation is a programming methodology. An execution dag is an a posteriori model of a parallel program and the programmer may know nothing about it. On the other hand, a dag evalution is explicitly programmed as a dag by the programmer.

Writing sequential code for dag computations is straightforward; as shown in Figure 4.1, one can traverse the dag nodes in a depth-first manner, computing each node after all its descendants have been computed. dag evaluations often contain inherent parallelism, however. For example,

This work was done in collaboration with Charles E. Leiserson and Jim Sukha.

```
SEQ-EVALUATE(B)
1  for (D ∈ children(B))
2       do if status[D] ≠ COMPUTED
3            then SEQ-EVALUATE(D) ▷ Recursively compute the child
4  COMPUTE(B)
5  status[B] ← COMPUTED
```

Figure 4.1: Pseudocode for sequential evaluation of a dag rooted at A. COMPUTE(B) represents the computation to be done at a node B.

two nodes that are not connected by any directed path in the dag can be computed in parallel. Furthermore, a node's COMPUTE method may contain additional parallelism. The challenging task, is to write a parallel program that effectively exploits both the dag's inherent parallelism and the possible parallelism in the COMPUTE method.

In order to parallelize a dag evaluation, programmers may design their own scheduler, possibly based on a task pool [KR03], to manage threads and to execute dag nodes. This approach is tedious and error-prone, however, since the programmer must explicitly track whether all the dependencies have been satisfied in addition to managing synchronization and load balancing. Data structures from existing concurrent libraries may simplify a scheduler for dag evaluations, but may introduce unnecessary overheads due to their generic interface. For example, one might use a concurrent queue from Intel Thread Building Blocks (TBB) library [Rei07] to maintain the list of all nodes that are ready to execute. Unfortunately, the queue's FIFO ordering introduces unnecessary dependencies between nodes. Finally, as mentioned above, nodes' COMPUTE functions may also contain parallelism, and the custom scheduler for dag evaluation must be designed to inter-operate effectively with this other form of parallelism.

On the other hand, programmers might try to code dag evaluations using a concurrency platforms, such as Cilk++ [Art09], which provide their own runtime scheduler. In Cilk++ code, programmers can ***spawn*** a function `foo()`, thereby specifying that `foo()` can potentially execute in parallel with the code immediately after the spawn of `foo()`. Within a function, programmers can ***sync*** to wait for all prior spawns in the function to complete. Cilk++, like its predecessor Cilk [BJK+96, FLR98], relies on a work-stealing scheduler for efficient execution. Programmers can easily express any fork-join parallelism inside the compute functions of individual nodes using these mechanisms.

However, since the Cilk++ language supports *only* fork-join parallelism, it is impossible encode the dependencies for an arbitrary dag within the control flow of a Cilk++ program without introducing additional synchronization. For example, one straightforward approach for expressing arbitrary dag dependencies in a fork-join program is to use locks, as shown in Figure 4.2. This program resembles the sequential dag evaluation shown in Figure 4.1. Unfortunately, using locks in this manner limits the scalability of a Cilk++ program in practice. These locks also invalidate the theoretical bounds on a program's completion time using a Cilk-like work-stealing scheduler. Ideally, one would like to be able to easily express dag evaluations in a language such as Cilk++, and take advantage of the efficient runtime schedulers provided by the language.

```
LOCK-EVALUATE(B)
1  acquire (lock(B)) ▷ Prevent other workers from interfering
2  for (D ∈ children(B))
3       do if status[D] ≠ COMPUTED
4            then spawn LOCK-EVALUATE(D) ▷ Recursively compute children
5  COMPUTE(B)
6  status[B] ← COMPUTED
7  release (lock(p))
```

Figure 4.2: dag Evaluation in a fork-join parallel language, using locks.

We have designed and implemented a library, called DAGEVAL, for parallel dag evaluations in Cilk++. Our interface is based on C++ objects and template classes; to represent dag nodes, the programmer creates an object which is a subtype of a base `DAGNode` class and overrides the object's `Compute` method. We support our interface for dag evaluations purely as a Cilk++ template library, without any modifications to the Cilk++ runtime system. As a result, DAGEVAL easily supports nested `spawn` statements inside the `Compute` function of each node. Thus, using DAGEVAL, it is easy to exploit both the inherent parallelism between the different dag nodes, and the parallelism inside the COMPUTE function of each dag node. In addition, a similar approach can be used to create libraries for dag evaluations for other fork-join parallel languages.

In DAGEVAL, we implement a parallel algorithm for dag evaluation using what we call ***eager traversal*** of the dag. In an eager traversal, each worker thread greedily attempts to visit the dag in a depth-first fashion, recursively trying to visit the children of each node it encounters. If any children are already being visited when a worker gets to it, the worker does some bookkeeping to record the fact, and moves on. Intuitively, in an eager traversal, the first worker to reach a node B ***expands*** B — spawning visits to all its children — and the worker who computes the value of the last uncomputed child of B ***enables*** the ***compute*** for B.

We evaluate DAGEVAL both theoretically and experimentally. Theoretically, for evaluations whose dags have nodes with only constant indegree and outdegree, we show the runtime for eager traversal is an asymptotically optimal $O(T_1/P + T_\infty)$, where T_1 is the sum of the work of the computes of all the nodes in the dag, and T_∞ is maximum over all paths p through the dag of the sum of the spans of the nodes along p. Irregular dynamic programs, i.e., dynamic programs which require a variable amount of time to compute every cell, represent one class of applications which fit this paradigm of dag evaluations. Such dynamic programs appear frequently in algorithms for computational biology (e.g., the Smith-Waterman algorithm (e.g., [SW81]). To evaluate the performance of DAGEVAL, we implemented an irregular dynamic program on an 2D grid using dag evaluations. Our empirical results indicate that when dag nodes are mapped to reasonably-sized blocks of cells, DAGEVAL is competitive with divide-and-conquer implementations of the same dynamic program. For example, for an N by N grid of cells with $N = 4000$, and blocks of 16 by 16 cells, DAGEVAL was less than 4% slower on one processor than a straightforward divide-and-conquer implementation, but actually more than 9% faster than the divide-and-conquer

approach on 8 processors.

4.2 DAGEVAL: A dag Evaluation Library

We implemented DAGEVAL for dag evaluation in Cilk++. This section describes the library interface and the implementation of the eager traversal strategy.

Interface

In DAGEVAL, programmers specify dag computations by creating nodes which extend from a base `DAGNode` object, specifying the children of each node, and evaluating the dag by invoking the `evaluate` method on the root of the dag.

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

$$M(i,j) = \max \begin{cases} M(i-1,j) + s(i-1,j) \\ M(i,j-1) + s(i,j-1) \end{cases}. \tag{4.1}$$

Figure 4.3 illustrates how one can express this computation as a dag evaluation. The code constructs a dag node for every cell $M(i,j)$ by extending from a dag node class. DAGEVAL supports different types of traversals of the dag, with the type specified as a template parameter of the `DAGNode` class. The programmer uses two methods of the base `DAGNode` class: `init_node` initializes each node with a specified key and a pointer to a structure wrapping the computation's global parameters, and `add_child` specifies children for a node.

Implementation

Now we describe the implementation details of DAGEVAL. Once a computation has been expressed as a dag computation, DAGEVAL provides several traversal options for performing the evaluation. In this section, we describe our primary strategy, called an ***eager traversal***. Other traversal options are described in Section 4.4.

Abstractly, an algorithm evaluates a node B by first visiting all of B's children, waiting for those children to complete, and then calling B's compute method. To evaluate a dag eagerly, DAGEVAL maintains the following fields for each node:

- **Key**: A unique 64-bit integer identifier for the node.
- **List of Children**: Each node has a pointer to an array for the list of children of the node.
- **Status**: Each node has a ***status*** field which changes monotonically, from `UNVISITED` to `VISITED`, then to `COMPUTED`, and finally to `COMPLETED`.
- **Waiting Parent List** Each node D maintains a list of parent nodes B which must be notified when D has been computed.
- **Join Counter**: Each node B maintains a ***join counter***, whose value is equal to the number of child nodes B is waiting on.

```
class DPDag {
 int n; int* s; MNode* g;
 DPDag(int n_, int* s_): n(n_), s(s_) {
  g = new MNode[n*n];
  for (int i = 0; i < n; ++i) {
   for (int j = 0; j < n; ++j) {
    int k = n*i+j;
    g[k].init_node(k, (void*)this);
    if (i > 0) {g[k].add_child(&MNode[k-n])};
    if (j > 0) {g[k].add_child(&MNode[k-1])};
   } } }
 int evaluate() { g[n*n - 1]->compute(); }
};

template <int TraversalType> ;
class MNode: public DAGNode<TraversalType> {
 int res;
 void Compute() {
  this->res = 0;
  for (int i = 0; i < children.size(); i++) {
   MNode* child = children.get(i);
   int child_val = child->res + s[child->key];
   res = MAX(child_val, res);
  } } };
```

Figure 4.3: Code using the dag evaluation library to solve the dynamic program in Equation (4.1). This code constructs a dag node for every cell $M(i, j)$.

For some dag evaluation strategies, some of these fields can be omitted. For example, in a serial depth-first traversal, nodes do not need to store the waiting parent list or the join counter.

Abstractly, the eager traversal strategy visits all nodes recursively. When a worker is visiting node A, if A's status is UNVISITED, then the worker tries to atomically changes the status to VISITED. It then tries to visit all of A's children recursively. If the status of A's child, say B is at least VISITED (due to a visit via some other parent of B), but not yet COMPUTED, then it increments A's join counter and adds A's to the list of B's waiting parents. After the worker has spawned all of A's children, it returns, even though A's value may not have been computed yet. Later, when another worker finishes computing node B, that worker decrements the join counter of all the nodes in B's waiting parent's list. If the join counter for any node B becomes 0, that worker enables A and recursively spawns the compute for A. A node's status changes to COMPLETED when it has notified all its parents in this fashion.

A more precise implementation is shown in Figure 4.4. In our code, a node B is initialized when join counter equal to the number of its children.[1] DAGEVAL attempts to expand each node B by spawning TRYVISITCHILD on all its children. The TRYVISITCHILD(B, D) method attempts to atomically change the status of a child D from UNVISITED to VISITED (line 3); if this change succeeds, we say that B is the ***expanding parent*** of D. Then, TRYVISITCHILD checks whether D is COMPUTED already; if not, then B must be added to D's waiting parent list.[2] Both the addition of a node to D's waiting parent list (line 9 in TRYVISITCHILD) and the change of D's status to COMPLETED (line 15 in COMPUTEANDNOTIFY) must be done while holding D's lock in order to prevent potential race conditions.[3]

The code in COMPUTEANDNOTIFY starting on line 3 handles the notification of parents. While D is notifying its parents, more nodes B which are parents of D might append themselves to D's waiting parent list. Each D maintains a ***notification counter***, tracking which elements on D's waiting parent list the runtime has already spawned notifications for. When D's notification counter is equal to the current length of its waiting parent list, then the worker changes status$[D]$ to COMPLETED.

4.3 Analysis of Performance

In this section, we provide a theoretical analysis of the runtime on P processors of parallel dag evaluation using an eager traversal strategy.

Definitions

Consider a dag evaluation computation expressed by the dag $\mathcal{D}$, with nodes $\mathcal{D}_V$ and edges $\mathcal{D}_E$. Conceptually, each node B in $\mathcal{D}_V$ has a list of children children(B), and a list of parents par(B). Let out(B) and in(B) be the out and in-degrees of the node B in $\mathcal{D}$. For simplicity in stating

[1] The interface we describe here permits this initialization because B's children are known before traversal of the dag begins. DAGEVAL also supports an interface where B's children are discovered when B is first visited; this interface requires increments of B's join counter as children are discovered.

[2] The waiting parent list is implemented as a concurrent dynamic array, optimized for atomic insertions at the end of the array.

[3] We can optimize slightly and potentially avoid a lock acquire (not shown in the code) by first checking if status$[D]$ is already COMPUTED before trying line 7 of TRYVISITCHILD.

```
TRYVISITCHILD(B, D)
 1  enabled ← false
 2  if status[D] = UNVISITED
 3     then enabled ← CAS status[D] from
                       UNVISITED to VISITED.
 4          if enabled
 5            then spawn EXPAND(D)
 6  finished ← true
 7  lock(D)
 8  if status[D] < COMPUTED
 9     then add B to waitingParents(D)
10          finished = false
11  unlock(D)
12  if finished
13     then val ← ATOMDECANDFETCH(join(B))
14          if val = 0
15            then spawn COMPUTEANDNOTIFY(B)

EXPAND(B)
1  assert(status[B] = VISITED)
2  assert(join(B) = |children(B)|)
3  for (c ∈ children(B))
4       do spawn TRYVISITCHILD(B, D)

COMPUTEANDNOTIFY(D)
 1  COMPUTE(D)
 2  status[D] = COMPUTED
 3  n ← SIZE(waitingParents(D))
 4  notified(D) ← 0
 5  while notified(D) < n do
 6     for i ∈ [notified(D), n)
 7       do B ← elmt i of waitingParents(D)
 8          val ← ATOMDECANDFETCH(join(B))
 9          if val = 0
10            then spawn COMPUTEANDNOTIFY(B)
11     notified(D) ← n
12     lock(D)
13     n ← SIZE(waitingParents(D))
14     if notified(D) = n
15          then status[D] ← COMPLETED
16     unlock(D)
```

Figure 4.4: Pseudocode for dag evaluation using an eager traversal.

the results, we assume that the dag has a unique node with no incoming edges (represented by $\texttt{root}(\mathcal{D})$), and a unique COMPLETED node with no outgoing edges (represented by $\texttt{final}(\mathcal{D})$). In addition, we define $\mathit{paths}(B, D)$ as the set of all paths in $\mathcal{D}$ from node B to node D.

To analyze the runtime of dag evaluation using eager traversals, we analyze executions of the program in Figure 4.4. For every node B in $\mathcal{D}$, an eager traversal invokes EXPAND(B) and COMPUTEANDNOTIFY(B) exactly once; however, the exact work performed by these functions varies, depending on a particular execution. Each possible execution can be represented as an execution graph (more specifically, dag) $\mathcal{E}$.

We define several notations for subgraphs of an execution graph $\mathcal{E}$. For a particular execution graph $\mathcal{E}$ and a dag node B, $\texttt{exp}^{\mathcal{E}}(B)$ denotes the subgraph of $\mathcal{E}$ corresponding to the call EXPAND(B), $\texttt{comNot}^{\mathcal{E}}(B)$ is the subgraph corresponding to the call COMPUTEANDNOTIFY(B), and $\texttt{com}^{\mathcal{E}}(B)$ is the subgraph corresponding to COMPUTE(B). For any subgraph $\mathcal{E}'$ of an computation dag, we denote the work of the subgraph as $W(\mathcal{E}')$, and the critical path length (span) as $S(\mathcal{E}')$. We overload notation, and when the superscript $\mathcal{E}$ is omitted, we mean the maximum of the quantity over all execution graphs $\mathcal{E}$; for example, $W(\texttt{com}(B))$ denotes the maximum work for COMPUTE(B) over all possible executions $\mathcal{E}$.

To account for costs due to locking and synchronization separately in an execution, for an execution $\mathcal{E}$, we define two quantities. Let $L_T(B, D)$ be the time spent inside the TRYVISIT(B, D) function holding locks or performing the CAS operation and atomic decrements. Let $L_C(B)$ be the time spent inside COMPUTEANDNOTIFY(B) holding locks or on atomic decrements. Again, both quantities represent maximum values over all executions.

Work of dag evaluation

To calculate the work of a dag evaluation, we first construct (pessimistic) bounds on the time an eager traversal spends waiting at synchronization operations.

Lemma 4.1 *For an eager traversal of $\mathcal{D}$,*

$$L_T(B, D) = O(\min\{\texttt{out}(B) + \texttt{in}(D), P\})$$

$$L_C(D) = O\left(\texttt{in}(D) + \sum_{(B,D)\in\mathcal{D}_E} \min\{\texttt{out}(B), P\}\right)$$

PROOF. In TRYVISITCHILD(B, D), we may have contention on the atomic decrement of B's join counter and on the insertion into $\texttt{waitingParents}(D)$. Each decrement of the join counter can wait at most $\min\{\texttt{out}(B), P\}$ time and the insertion can wait at most $\min\{\texttt{in}(D), P\}$ time.

Similarly, in COMPUTEANDNOTIFY(D), we may perform an atomic decrement for every edge $(B, D) \in \mathcal{D}_E$ and this decrement may wait at most $\min\{\texttt{out}(B), P\}$ time. When checking the size field of D's waiting parent list, we may wait for at most all of D's parents. □

Lemma 4.2 *The work of an eager traversal of $\mathcal{D}$ is*

$$\left(\sum_{B \in \mathcal{D}_V} W(\texttt{com}(B))\right) + O\left(|\mathcal{D}_E|\right) + O\left(\sum_{(B,B) \in \mathcal{D}_E} \min\{\texttt{out}(B) + \texttt{in}(B), P\}\right).$$

PROOF. The first term arises from the work of the compute functions. The second term bounds the work of traversing $\mathcal{D}$, assuming no contention. The third term covers the contention cost (Lemma 4.1). □

Span of dag evaluation

The nondeterministic nature of the computation complicates a direct calculation of $S(\exp(B))$. Instead, we construct a new, deterministic execution dag $\mathcal{E}^*$, whose span is an upper bound on the span of EXPAND(root). We define the methods EXPAND*(B), TRYVISITCHILD*(B, D) and COMPUTEANDNOTIFY*(B) to be the same as the original methods, except that all possible recursive calls always occur. In other words, EXPAND*(B) always recursively expands every child of B and COMPUTEANDNOTIFY*(B) always recursively computes every parent of B. Let $\texttt{exp}^*(B)$ and $\texttt{comNot}^*(B)$ be the execution subgraphs corresponding to these modified method calls for B, and let $\mathcal{E}^*$ be the execution dag for EXPAND*(root). Since any execution $\mathcal{E}$ forms a prefix of $\mathcal{E}^*$, we know $S(\texttt{exp}^*(B)) \geq S(\texttt{exp}^{\mathcal{E}}(B)$ and $S(\texttt{comNot}^*(B)) \geq S(\texttt{comNot}^{\mathcal{E}}(B))$. Figures 4.5 and 4.6 show execution dags for these modified methods.

Lemma 4.3 bounds $S(\texttt{comNot}^*(B))$ and Lemma 4.4 bounds $S(\texttt{exp}^*(B))$.

Lemma 4.3 *$S(\texttt{comNot}^*(B))$ is at most*

$$\max_{p \in \mathit{paths}(\texttt{root}, B)} \left\{\sum_{X \in p} (O(\texttt{in}(X)) + L_C(X) + S(\texttt{com}(X)))\right\}.$$

PROOF. From the COMPUTEANDNOTIFY code in Figure 4.4, we see that $\texttt{comNot}^*(X)$ will enable all parents of a node X, with each recursive COMPUTEANDNOTIFY* happening in parallel (e.g., see Figure 4.5). It is an upper bound to put all synchronization operations on the span of X.

The proof is by induction on the distance of a node X from root in $\mathcal{D}$. In the base case, $S(\texttt{comNot}^*(\texttt{root}))$ is bounded by $f_C(\texttt{root}, c)$ since computation at root makes no recursive calls.

In the inductive step, assume the lemma holds for all nodes Y which are at a distance at most $k-1$ from root, and consider a node X at a distance k from root.

From Figure 4.5, we can see that for some constant c,

$$\begin{aligned} S(\texttt{comNot}^*(X)) \; &\leq S(\texttt{com}(X)) + c \cdot \texttt{in}(X) + L_C(X) \\ &\quad + \max_{A \in \texttt{par}(X)} \{S(\texttt{comNot}(B))\} \end{aligned}$$

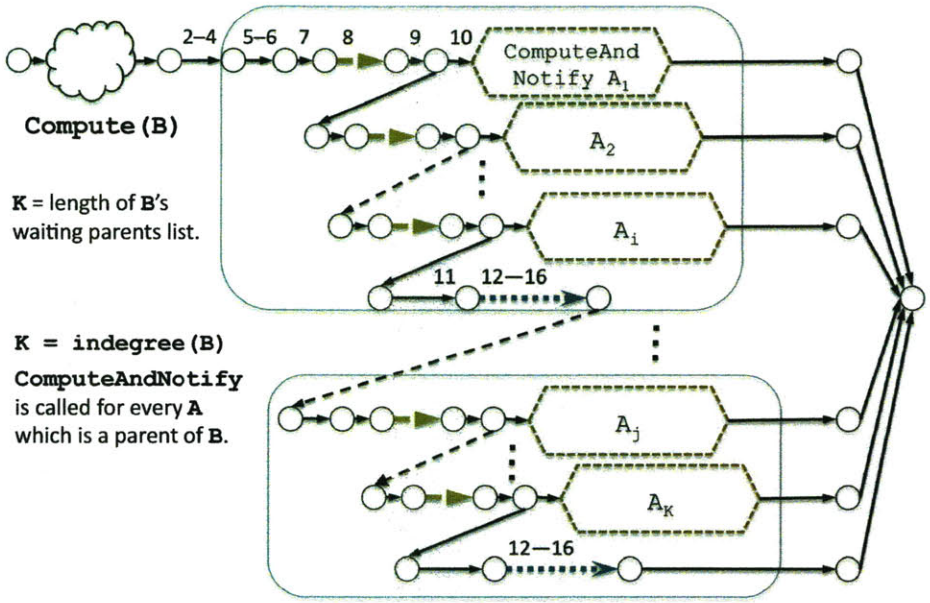

Figure 4.5: An execution dag for COMPUTEANDNOTIFY*(D). Numbers correspond to line numbers from Figure 4.4. Dashed arrows correspond to operations which require synchronization; either an atomic decrement, or a locked section. In the worst-case for span, all $\texttt{in}(D)$'s parents are added to the waiting parent list, and the last parent B_K added has the maximum span of all of B's parents.

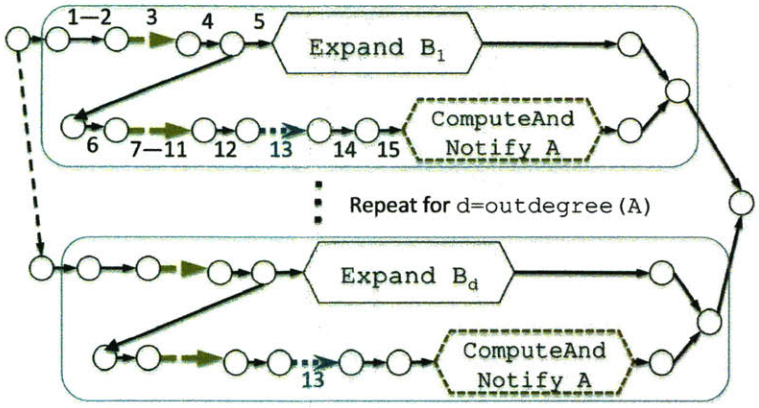

Figure 4.6: Execution dag for EXPAND*(B). Dashed arrows correspond to synchronization operations. Every child D_i of B is recursively expanded, and COMPUTEANDNOTIFY*(B) is called for every child D_i

The inductive step follows by considering all paths p from `root` to X, and applying the inductive hypothesis to the prefix of p without X. □

Lemma 4.4 *$S(\texttt{exp}^*(B))$ is at most*

$$S(\texttt{comNot}^*(\texttt{final}))+ \max_{p \in paths(B,\texttt{final})} \left\{ \sum_{X \in p} O(\texttt{out}(X)) + \sum_{(X,Y) \in p} L_T(X,Y) \right\}.$$

PROOF. The proof is by induction on the distance of a node X from `final`. In the base case, we know *procExpand**(`final`) only calls COMPUTEANDNOTIFY*(`final`).

In the inductive step, suppose for all nodes $Y \in \mathcal{D}$ at distance at most $k-1$ from `final`, the lemma holds. Consider a node X at distance k from `final`.

From Figure 4.6, we see that for some constant c, $S(\texttt{exp}(X))$ satisfies the recurrence

$$S(\texttt{exp}^*(X)) \leq c \cdot \texttt{out}(X) + \max_{Y \in \texttt{children}(X)} \left\{ \begin{array}{c} S(\texttt{exp}^*(Y)) + L_T(X,Y) \\ S(\texttt{comNot}^*(X)) + L_T(X,Y) \end{array} \right\}. \quad (4.2)$$

Using Lemma 4.3, we know that $S(\texttt{comNot}^*(\texttt{final})) \geq S(\texttt{comNot}^*(X))$ for any $X \in \mathcal{D}$. Then, using the inductive hypothesis, we can substitute the inductive hypothesis for $S(\texttt{exp}^*(Y))$, bound all COMPUTEANDNOTIFY terms by $S(\texttt{comNot}^*(\texttt{final}))$, and get

$$S(\texttt{exp}^*(X)) \leq S(\texttt{comNot}^*(\texttt{final}))+ + \max_{Y \in \texttt{children}(X)} \left\{ \max_{p \in paths(Y,\texttt{final})} f_E(p,c) + c \cdot \texttt{out}(X) + L_T(X,Y) \right\}.$$

Adding X onto the path p and combining terms completes the proof of the inductive hypothesis. □

Completion time bounds

Using Lemmas 4.2 and 4.4, and the analysis of a Cilk-like work-stealing scheduler [BL99], we obtain the following bound for the completion time for eager traversal.

Theorem 4.5 *Consider a dag $\mathcal{D}$ with maximum degree d and span $S(\mathcal{D})$. With probability at least $1-\epsilon$, an eager traversal of $\mathcal{D}$ on P completes in time*

$$O\left(T_1/P + T_\infty + \lg(P) + \lg(1/\epsilon) + L(\mathcal{D})\right),$$

where

$$T_1 = \left(\sum_{B \in \mathcal{D}_V} W(com(B)) \right) + O\left(|\mathcal{D}_E|\right)$$

$$T_\infty = \max_{p \in paths(root,final)} \left\{ \sum_{X \in p} S(com(X)) \right\} + O(dS(\mathcal{D}))$$

and

$$L(\mathcal{D}) = O\left(\left(\frac{|\mathcal{D}_E|}{P} + dS(\mathcal{D}) \right) \min\{d, P\} \right)$$

PROOF. The proof follows from Lemmas 4.1, 4.2 and 4.4; we bound the indegrees and outdegrees of nodes by d, and bound expressions which compute maximum over paths p in terms $S(\mathcal{D})$. □

In Theorem 4.5, the terms T_1 and T_∞ represent the natural generalizations of work and span for a dag, taking into account the subcomputations within each node. For example, the first term in T_∞ is the sum of the spans of all the compute nodes along the longest path in $\mathcal{D}$. The T_∞ term contains $dS(\mathcal{D})$ instead of just $S(\mathcal{D})$ because each node along the longest path of nodes in $\mathcal{D}$ may scan through its lists of neighbors serially. The $L(\mathcal{D})$ term gives a bound on the contention due to synchronization during the dag evaluation. The extra factor of $\min\{d, P\}$ appears because we assume worst-case contention, which hopefully is unlikely to occur in practice. As an important special case, note that for dags where every node has constant degree, i.e., $d = O(1)$, the term $L(\mathcal{D})$ is absorbed by $T_1/P + T_\infty$.

4.4 Experimental Setup

In this section, we describe our experimental setup for comparing eager traversal with other traversals for dag evaluation. First we describe the other dag traversals and then describe other details.

Traversals for dag evaluation

DAGEVAL, described in Section 4.2, supports three different parallel dag traversals — eager, blocking and helping traversals. We compare these three traversals for our benchmarks.

An ***eager traversal*** is the strategy discussed in Section 4.2, and is the default option for DAGEVAL.

A ***blocking traversal*** visits each node B by recursively spawning visits to B's children. When a worker successfully marks B as `VISITED`, it first tries to recursively visit any of B's children which are `UNVISITED`. Then, if all of B's children are at least `VISITED`, then the worker blocks until B's status is `COMPUTED`. Blocking traversal is similar to LOCK-EVALUATE shown in Figure 4.2.

A ***helping traversal*** is similar to a blocking traversal; however, when a worker encounters an B whose children are at least `VISITED`, then it picks one of B's children which is not yet `COMPUTED`, and recursively tries to help visit that children. In this approach, a node may be "visited" by multiple workers, but additional synchronization guarantees that the compute of every node happens exactly once.

Blocking and helping traversals were implemented in a straightforward fashion, using atomic updates and spin-waiting on the status fields for synchronization. Note that for blocking and helping traversals, a visit of a node B never returns until the value of B has been computed. If the programmer mistakenly creates graph $\mathcal{D}$ with a cycle, blocking traversal may deadlock, helping traversal may enter an infinite loop, and an eager traversal may terminate without computing all the nodes.

Machine configuration

We ran our experiments on two multicore machines, with 8 and 16 total cores, respectively. Our first machine is a two-socket machine, quad-core (3.16 GHz Intel Xeon X5460) machine with 8 GB RAM. Each processor has 6 MB of cache, shared among the four cores, and a 1333 MHz FSB. The second machine is a four-socket machine, quad-core (2.40 Ghz Intel E7340) machine with 16 GB RAM. Each processor has 4 MB of shared cache, and a 1066Mhz FSB. Both machines run 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 (based on GCC 4.2.4) with optimization flag `-O2`.

4.5 Dynamic Programming Application

Our primary test application is a dynamic programming computation on a 2D grid. In particular, we consider the dynamic program which computes a value $M(i,j)$ based on the following set of recursive equations:

$$\begin{array}{rcl} E(i,j) & = & \max_{k\in\{0,1,\ldots i-1\}} M(k,j) + \gamma(i-k) \\ F(i,j) & = & \max_{k\in\{0,1,\ldots j-1\}} M(i,k) + \gamma(j-k) \\ M(i,j) & = & \max \left\{ \begin{array}{l} M(i-1,j-1) + s(i,j) \\ E(i,j) \\ F(i,j) \end{array} \right. \end{array} \tag{4.3}$$

The functions $s(i,j)$ and $\gamma(z)$ can be computed in constant time; in our actual implementation, we lookup values for s and γ from tables in memory. This dynamic program is irregular because the work for computing the cells is not the same for each cell; $O(i+j)$ work must be done to compute $M(i,j)$. Therefore, in total, computing $M(m,n)$ using Equation (4.3) requires $\Theta(mn(m+n))$ work ($\Theta(n^3)$, when $m=n$). As described in [LLS07], this particular dynamic program models the computation used for the Smith-Waterman [SW81] algorithm with a general penalty gap function γ.

4.5.1 Parallel Algorithms

We explored two types of parallel algorithms for this dynamic program. One type is based on dag evaluations, and the other uses divide-and-conquer techniques.

DP as a dag evaluation

We can express the dynamic program in Equation (4.3) as a dag evaluation by creating a dag $\mathcal{D}$ similar to the code in Figure 4.3.[4] In order to improve cache-locality and amortize the overheads of dag nodes, however, we block the grid so that every dag node represent a B by B block of cells (instead of every node representing a single cell as in Figure 4.3). Block (b_i, b_j) represents the block with upper left corner at cell (b_iB, b_jB). In the dag, block (b_i, b_j) depends on (at most) two

[4]Although $M(i,j)$ depends on the entire row i to the left of the cell and the entire column j above the cell, when creating a dag, it is sufficient to create dependencies to $M(i,j)$ only from $M(i-1,j)$ and $M(i,j-1)$; other dependencies are ensured by transitivity.

```
ComputeM(n) { ComputeMHelper(0, 0, n); }

// Computes M for an n by n grid,
// with upper left corner at (i, j)
ComputeMHelper(i, j, n) {
  if (n <=B) { ComputeMBase(i, j, n); }
  else {
    ComputeMHelper(i, j, n/2);
    cilk_spawn ComputeMHelper(i+n/2, j, n/2);
    cilk_spawn ComputeMHelper(i, j+n/2, n/2);
    cilk_sync;
    ComputeMHelper(i+n/2 j+n/2, n/2);
  } }
```

Figure 4.7: Psuedocode for a parallel divide-and-conquer algorithm to compute $M(n, n)$ for the dynamic program in Equation (4.3). For simplicity, we only show the code when $m = n$ is a power of 2.

blocks $(b_i - 1, b_j)$ and $(b_i, b_j - 1)$. The compute method for each node computes the values of M for the entire block sequentially. Theorem 4.6 analyzes the span of this computation.

Theorem 4.6 *For Equation (4.3), if the size of the matrix is n and the block size is B, an eager traversal of $\mathcal{D}$ has a span of $\Theta(n^2 B)$, assuming $n > B$.*

PROOF. The span of $\mathcal{D}$ consists of $\Theta(n/B)$ blocks, with a least half the blocks requiring $\Theta(n^2 B)$ work. The result follows from the T_∞ term in Theorem 4.5. □

DP as a divide and conquer program

We can also devise a divide-and-conquer algorithm for the dynamic program, as shown in Figure 4.7. This algorithm divides the grid into 4 sub-grids, and then computes the cells in each subgrid recursively. The computations for the two sub-grids along the antidiagonal can be performed in parallel. We also block the divide and conquer implementation for locality and to provide a large enough base case to overcome overheads. Theorem 4.9 computes the span of this algorithm (proof omitted due to space constraints).

To compute bounds on the span of the divide and conque algorithm, let $S(i, j, m, n)$ denote the span of the `ComputeMHelper` method. We look at the recursion tree (i.e., call tree) generated when calling `ComputeMHelper(0, 0, m, n)`. Every node x in this tree represents some call to `ComputeMHelper(i, j, m, n)`; an x which is an internal node has either 2 or 4 children (corresponding to recursive calls to `ComputeMHelper`, while an x which is a leaf calls `ComputeMBase`. For shorthand, we let $S(x)$ denote the span corresponding to x's `ComputeMHelper` call.

To compute an upper bound on $S(x)$, we assign every x in the recursion tree a height $h(x)$, which is 0 if x is a leaf, or 1 plus the maximum of the heights of its children if x is an internal node. We claim that $S(x)$ is upper-bounded by the

Lemma 4.7 *Let x be a node in the recursion tree corresponding to a call* `ComputeMHelper(i, j, m, n)`. *Then for some constant c, $S(x)$ satisfies*

$$S(x) \leq c \cdot 3^{h(x)} \left(B^3 \cdot 2^{h(x)} + B^2 \cdot (i+j)\right).$$

PROOF. The proof is by induction on the height of the recursion tree. In the base case, any x which is a leaf in the recursion tree has height $h(x) = 0$, and corresponds to some base-case call `ComputeMBase(i, j, m, n)`. From the code in Figure 4.7, in this call, we know $m \leq B$ and $n \leq B$. We know the work for the base case is at most

$$T(i,j,m,n) \leq cmn\left(\frac{m+n}{2} + (i+j)\right) \leq B^3 + B^2(i+j).$$

This expression matches exactly the formula above with $h(x) = 0$.

In the inductive step, suppose for all nodes y with $h(y) < k$, the lemma on $S(y)$ holds. Consider a node x at height $h(x) = k$, corresponding to a call `ComputeMHelper(i, j, m, n)`. Since $k > 0$, x must have either 2 or 4 children. First, consider the case where x has 4 children. Denote these children as x_{00}, x_{10}, x_{01}, and x_{11}, corresponding to the recursive subproblems in Figure 4.7. The span of these children satisfy the following relationships:

$$\begin{aligned} S(x_{00}) &= S(i,j,m/2,n/2) \\ S(x_{10}) &= S(i+m/2,j,m-m/2,n/2) \\ S(x_{01}) &= S(i,j+n/2,m/2,n-n/2) \\ S(x_{11}) &= S(i+m/2,j+n/2,m-m/2,n-n/2) \\ S(x) &= S(x_{00}) + \max(S(x_{10}), S(x_{01})) + S(x_{11}) \end{aligned} \tag{4.4}$$

Each of the 4 children has height most $k-1$; thus, by applying the inductive hypothesis and collecting terms which are independent of m and n, we have

$$\begin{aligned} S(x) \leq\ & 3c \cdot 3^{k-1}\left(B^3 \cdot 2^{k-1} + B^2(i+j)\right) \\ & + c \cdot 3^{k-1} B^2 \max\left(\tfrac{m}{2}, \tfrac{n}{2}\right) \\ & + c \cdot 3^{k-1} B^2 \left(\tfrac{m}{2} + \tfrac{n}{2}\right) \end{aligned} \tag{4.5}$$

For any node x at height k, it is straightforward to show that $m \leq 2^k B$ and $n \leq 2^k B$, since as we walk up from the leaves of the recursion tree, each dimension can at most double. Substituting these bounds into Equation (4.5), we can bound the sum of the last two lines by $c \cdot 3^k B^3 \cdot 2^{k-1}$. Adding all these terms together proves the inductive hypothesis for node x at height k. □

To compute a lower bound on the span, for simplicity, we consider only the prefix of the recursion tree which is the maximum number of levels of the tree which still forms a complete 4-ary tree. For each node x in the prefix recursion tree, let $g(x)$ denote the height of x (with leaves having height 0). We know the root of the tree must have height at least $\lg(\min m, n/B)$, otherwise, a base case could not have been reached, and the recursion would divide into 4 subproblems for at least one more level. Also, any x in the prefix recursion tree at height $g(x)$ must correspond to a problem with $m \geq 2^{g(x)}\frac{B}{2}$ and $n \geq 2^{g(x)}\frac{B}{2}$.

From these results, we can prove a lower bound for $S(x)$ in terms of $g(x)$, in a fashion analogous to Lemma 4.7.

Lemma 4.8 *Let x be a node in the complete prefix of the recursion tree, with height $g(x)$ in this prefix tree. Suppose x corresponds to a call* `ComputeMHelper(i, j, m, n)`. *Then for some constant c, $S(x)$ satisfies*

$$S(x) \geq \frac{c}{4} \cdot 3^{g(x)} \left(\frac{B^3}{2} \cdot 2^{g(x)} + B^2 \cdot (i+j) \right).$$

PROOF. The proof is by induction on the height $g(x)$. In the case where $g(x) = 0$, x must correspond to a subproblem of size at least $m \geq \frac{B}{2}$ and $n \geq \frac{B}{2}$. (If the problem was any smaller, then x's parent should have executed a base case). This subproblem has span which must be at least the time to execute a block of the same size serially, i.e., at least $T(i, j, B/2, B/2)$. Setting $g(x) = 0$ matches the work for $T(i, j, B/2, B/2)$.

In the inductive step, suppose for all nodes y with $g(y) < k$, the lemma on $S(y)$ holds. Consider a node x at height $g(x) = k$, corresponding to a call `ComputeMHelper(i, j, m, n)`. For the recursion, Equation (4.4) still holds; in place of Equation (4.5), however, we have:

$$\begin{aligned} S(x) \quad \geq \quad & \tfrac{3c}{4} \cdot 3^{k-1} \left(\tfrac{B^3}{2} \cdot 2^{k-1} + B^2(i+j) \right) \\ & + \tfrac{c}{4} \cdot 3^{k-1} B^2 \left(\tfrac{m}{2} + \tfrac{n}{2} + \max\left(\tfrac{m}{2}, \tfrac{n}{2} \right) \right) \end{aligned} \tag{4.6}$$

Since we know at height k, $m \geq 2^k \frac{B}{2}$ and $n \geq 2^k \frac{B}{2}$, we substitute into Equation (4.6) to prove the inductive hypothesis. □

Theorem 4.9 *The span of the divide-and-conquer algorithm in Figure 4.7 is $\Theta(n^{\lg 6} B^{3-\lg 6}) \approx \Theta(n^{2.585} B^{0.415})$, where n is the matrix size and B is the block size.*

Theorems 4.6 and 4.9 indicate that the parallelism (i.e., work divided by span) of an eager traversal dag evaluation is $\Theta(n/B)$, but only about $\Theta((n/B)^{0.415})$ for the divide-and-conquer algorithm. One can asymptotically decrease the span of a divide-and-conquer algorithm by dividing M into more subproblems, but the code becomes more complex. In the limit, the resulting parallelism of divide-and-conquer would also approach $\Theta(n/B)$.

Implementation

In our experiments, we compared four parallel implementations of Equation (4.3): the three dag traversals (eager, helping and blocking) and the divide-and-conquer algorithm shown in Figure 4.7. 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 by B block. For all our implementations, when computing at a cell $M(i, j)$, we do not parallelize the computation of $E(i, j)$ and $F(i, j)$; instead, we loop over the prefix of row i and over column j serially. For problems with $N \leq 1000$, we found that the overhead of using Cilk++ parallel constructs in this loop introduced noticeable overhead. Problems with $N > 1000$ tend to already have sufficient parallelism, at least for the number of processors on our test machines.

Since memory layout impacts performance significantly for large problem sizes, we stored both $M(i, j)$ and $s(i, j)$ in a cache-oblivious [FLPR99] layout. The computations of $E(i, j)$ and $F(i, j)$ require scanning along a column and row, respectively; thus, simply storing M in a row-major

or column-major layout would be suboptimal for one of the computations.[5] To support efficient iteration over rows and columns, we use dilated integers as indices into the grid [WF99], and techniques for fast conversion between dilated and normal integers from [RW08].

Experimental results

We ran two different types of experiments on our implementations of the dynamic program. The first experiment measures the parallel speedup of the four different techniques for various problem sizes N. The second experiment measures the sensitivity of the eager dag evaluation and the divide-and-conquer approaches to different choices in block size B.

Speedup of various techniques

In this experiment, we compare the speedup provided by the three dag traversal strategies and divide-and-conquer algorithm. For this experiment, we fix the block size at $B = 16$; each dag node is responsible for computing a 16 by 16 block of the original grid, and the divide-and-conquer algorithm recurses down to blocks of size 16 by 16.

Figures 4.8, 4.9 and 4.10 shows the speedup on P processors for $N \in \{1000, 5000, 15000\}$ on our second machine.[6] All the approaches run in virtually the same time on $P = 1$; the dag evaluation using an eager traversal has a greater speedup than the divide-and-conquer version, however. For example, at $N = 1000$, the divide-and-conquer algorithm achieves a speedup of less than 5 on 16 processors, while an eager traversal exhibits a speedup of about 14. This result is not surprising, since the dag evaluation has a higher asymptotic parallelism than the divide-and-conquer algorithm. As N increases to 5000, both the eager traversal and the divide-and-conquer algorithm improve in scalability. Locking and helping traversals, however, both appear to achieve a speedup of no more than 2.

As N increases even more, however, the speedup starts to level off, and eventually decrease. We conjecture that this slowdown is due to increased data bus traffic and a lack of locality when computing the terms $E(i, j)$ and $F(i, j)$. In Equation (4.3), if we replace the γ term with indices which are independent of k, then we see a significant improvement in speedup on $N = 15000$.

Effect of block size

To measure the sensitivity of eager traversal to block size, we fix N and vary B. Figure 4.11 shows the results for $N = 4000$. For small block sizes, we see that eager traversal performs worse than divide and conquer due to overheads. At small block sizes, in addition to having large computation overhead for each dag node, DAGEVAL also has significant space overhead. dag nodes are 72 byte objects (plus pointers to memory that stores children and parent lists). This overhead is significant if each node only represents a small block which contains only a few integers. As the block size increases, however, at $P = 1$, the runtime for eager traversal approaches the runtime for divide and conquer, and it scales better than divide and conquer for $B \geq 16$.[7]

[5] As a point of comparison, the divide-and-conquer algorithm for $N = 2000$ took about 300 s using a Morton-order layout, but 460 s using a row-major layout.

[6] Experiments on our first machine show similar results.

[7] Preliminary results show that dividing the grid into 25 subproblems instead of 4 narrows, but does not eliminate, this gap.

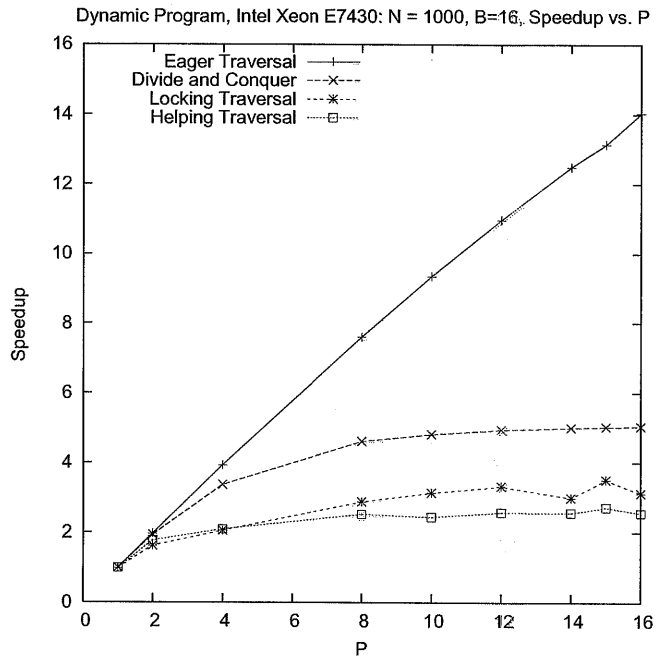

Figure 4.8: Dynamic program on an N by N grid ($N = 1000$ and $B = 16$). Speedup is normalized to the fastest run with $P = 1$ (3.52 s for divide-and-conquer). Results are from the second machine, with 16 cores total.

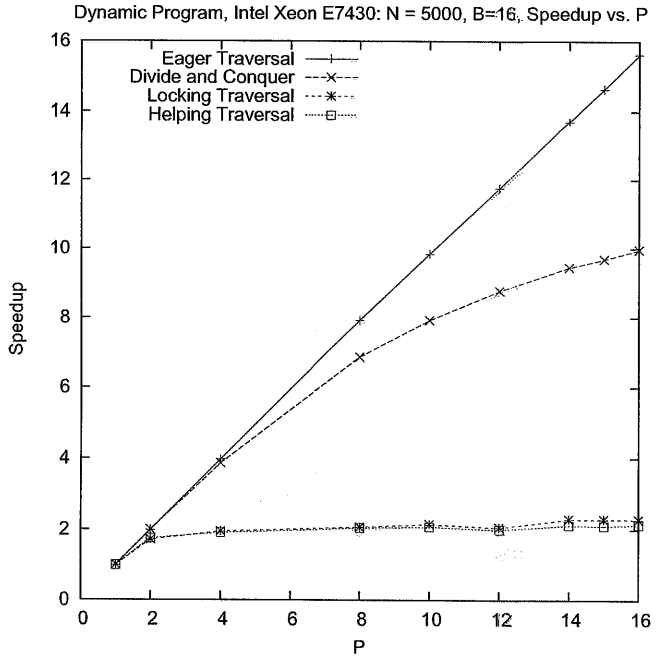

Figure 4.9: $N = 5000$ and $B = 16$. Speedup is normalized to the fastest run with $P = 1$ (442 s for divide-and-conquer).

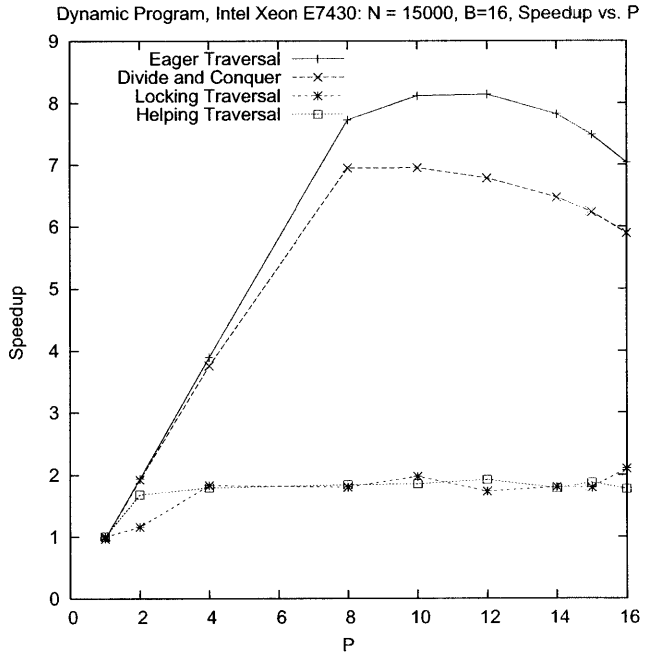

Figure 4.10: $N = 15000$ and $B = 16$. Speedup is normalized to the fastest run with $P = 1$ (12,376 s for helping traversal).

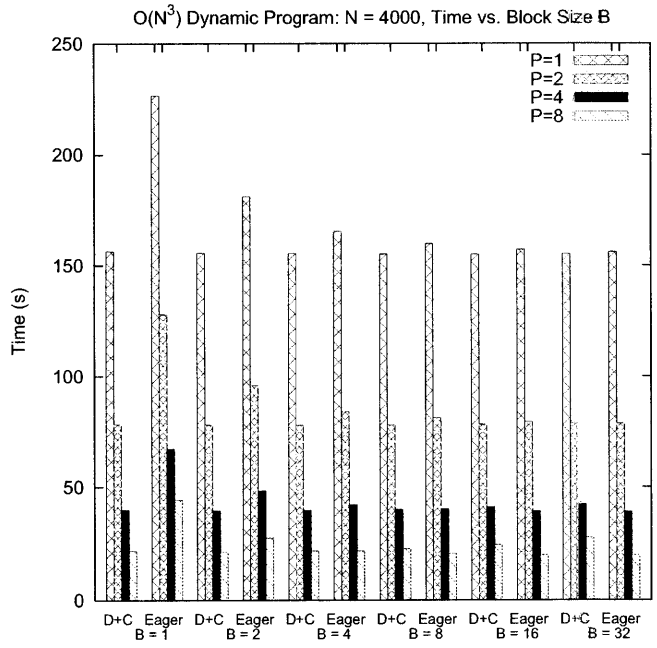

Figure 4.11: Running time for $O(N^3)$ dynamic program for $N = 4000$, varying block size for base case B. Results are from the first machine.

We also modified the dynamic program in Equation (4.3) to perform only $O(1)$ work at cell (i, j) instead of $O(i + j)$ work. Our results (not shown) indicate that this modified version requires larger block sizes (e.g., $B \geq 32$) before the eager traversal outperforms the divide-and-conquer algorithm on 8 processors. Again, this result is not surprising; since each node does less work for a particular block size, we require larger block sizes to overcome the overheads.

In summary, our experiments indicate that although dag evaluations may suffer from high overheads when each node does very little work, in general for this dynamic program, an eager traversal has relatively small overheads and is competitive with a divide-and-conquer implementation for reasonably sized blocks.

4.6 Random Dag Microbenchmark

To assess the overhead of the dag evaluation library and to understand its performance on an irregular application, we constructed a microbenchmark which evaluates randomly constructed dags. We generate random dags $\mathcal{D}$ based on three parameters: d – the maximum outdegree of each node, U – the size of the universe from which keys are chosen, and W – the work in the compute of each dag node. $\mathcal{D}$ has a single root node B_0 with key 0. Then, iterating over k from 0 to $U - 1$, we repeat the following process:

- If $\mathcal{D}$ has a node B_k, choose an integer d_k uniformly at random from the interval $[1, d]$.
- 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 edges $(B_k, B_{k'})$ to the dag (creating $B_{k'}$ if it doesn't already exist).

In $\mathcal{D}$, each dag node A_k performs W work, computing $k^W \bmod p$ using repeated multiplication (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

We use the random dag benchmark in three experiments: (1) to measure the overhead of performing parallel dag evaluation, (2) to compare the various traversal strategies, and (3) to evaluate the benefits of allowing parallelism inside the computes of nodes.

To measure the approximate overhead for manipulating dag node objects, and for parallel bookkeeping, we construct a medium-sized random dag and vary W. For $W = 1$, Figure 4.12 shows that the overhead in our synthetic benchmark can be on the order of thousands of cycles per node, or hundreds of cycles per edge. We observe that each node generally requires a W on the order of at least 1,000 to 10,000, before the overhead per node no longer dominates the cost of computation. In Figure 4.12, the overhead of bookkeeping for eager traversal was between a factor of 2 to 3 times over the serial traversal when $W = 1$. As W increases, however, the differences became negligible.

To compare the eager traversal to other traversals, we first created large random dag, with very little work per node, i.e., $W = 1$. For eager, blocking, and helping traversals, Figure 4.13 shows the

	Eager Traversal			Serial Traversal		
W	T_1(s)	$\frac{\text{Cycles}}{W\|V\|}$	$\frac{\text{Cycles}}{W\|E\|}$	T_1(s)	$\frac{\text{Cycles}}{W\|V\|}$	$\frac{\text{Cycles}}{W\|E\|}$
1	0.016	3202	583	0.007	1401	255
10	0.017	340	62.0	0.009	180	32.8
100	0.029	58.0	10.6	0.021	42.0	7.66
1000	0.153	30.6	5.58	0.144	28.8	5.25
10^4	1.379	27.6	5.03	1.370	27.4	5.00
10^5	13.64	27.3	4.97	13.63	27.3	4.97

Figure 4.12: Evaluation of $\mathcal{D}$ with $|V| = 15489$ and $|E| = 85012$, for $P = 1$. $\mathcal{D}$ was randomly generated with $d = 10$, $U = 150000$ and $W = 1$.

speedup on P processors over the serial traversal. Eager traversal provides only limited speedup (at most 2.5) compared to blocking and helping traversals, since eager traversal uses larger dag nodes and has more overhead for bookkeeping. Also, the locking and helping traversals of a randomly constructed dag spend relatively little time spin-waiting (compared to the grid in Section 4.5), since paths through the dag are not likely to overlap.

On the other hand, we can see from Figure 4.14, when each node has a substantial amount of work to do, then the eager traversal is more scalable than both the blocking and the helping traversals. In this case, the dag has relatively few nodes (only 128). Therefore, if we look at the version where each node is computed sequentially, the theoretical parallelism is only about $127/29 = 4.37$. The eager traversal exploits most of this available parallelism, reaching a maximum speedup of more than 4.2. In contrast, the blocking and the helping versions provide a speedup of about 3.6.

More importantly, however, Figure 4.14 demonstrates that to attain the best performance, one needs to exploit parallelism both at the dag level and in the COMPUTE functions. The eager traversal with parallelism inside COMPUTE functions shows the best speedup. Parallelism within nodes also improves the blocking and helping traversals slightly, but not as much as the eager traversal. In fact, a serial dag traversal with parallelism inside nodes outperforms blocking and helping traversals.

4.7 Future Work

We have described a library, DAGEVAL, which implements eager traversal for parallel dag evaluation in Cilk++. DAGEVAL allows programmers to exploit both dag-level parallelism and parallelism within the COMPUTE functions. We would like to extend the library to support a more dynamic interface. In this chapter, we saw an interface which assume the children of a node are known before traversal of the dag begins. In fact, DAGEVAL contains mechanisms for allowing children of a node to be discovered on the first visit to a node. One might further extend this interface, however, and allow children of a node B to be added dynamically, after computing the results of some of B's other children. One direction for future work is to explore what the interface for specifying such dynamic dags might be, and whether one can support the interface with low overhead. We would also like to explore other types of applications where using dag evaluations might simplify parallel programming or improve program performance. Finally, from our dynamic programming benchmark, we see that the performance of a dag evaluation may be limited by locality and memory bandwidth issues. It is an interesting research question to understand whether one

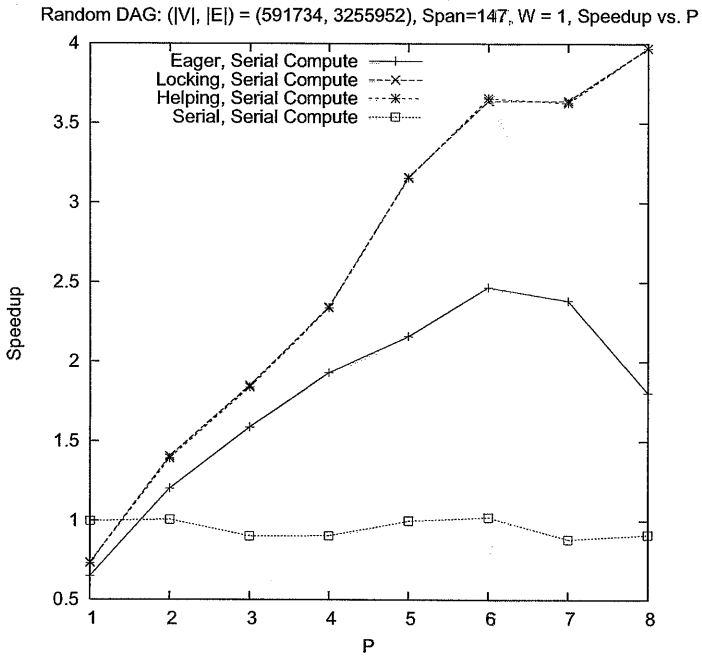

Figure 4.13: Comparison of eager, helping, and locking traversals for a large random dag with $W = 1$. Speedup is normalized over time for serial traversal when $P = 1$ (0.65 s).

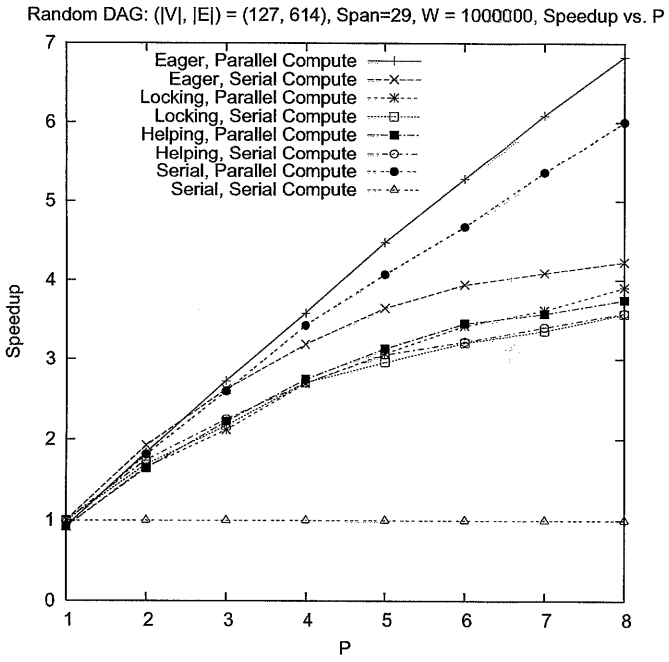

Figure 4.14: Comparison of eager, helping, and blocking traversals, with and without parallelism in the COMPUTE function. Speedup is normalized over time for serial traversal with serial compute, for $P = 1$ (1.12 s).

can take advantage of locality in a dag evaluation, particularly for dags with an irregular structure.

Chapter 5

Region Helper Locks

As mentioned in Chapter 1, dynamic multithreaded languages often allow for efficient scheduling. In particular, work-stealing provides asymptotically optimal completion time and good space bounds for programs with fork-join parallelism. Unfortunately, most of these theoretical bounds do not hold when programs have locks. Consider an example of a concurrent hash table, which is resized when it gets too full. Normally, two inserts, say `G1` and `G2`, can run in parallel, assuming `G1` and `G2` access different buckets. However, if `G1` triggers a task `H` which resizes the table, then `G1` and `G2` should not run concurrently. This property is typically enforced by using locks. Using locks to protect large critical sections often compromises scalability, however. For example, if worker p is performing a resize operation, all processors that wish to perform inserts must wait for the resize to finish.

Since the resize `H` is an expensive operation, one would like to execute it in parallel. In addition, since the worker p executing an insert `G2` must already wait for `H` to finish, one might want p to "help" execute the parallel work within `H`. In this chapter, we introduce ***helper locks*** into a fork-join parallel programming language. We add two types of helper locks, namely ***region helper locks*** and ***short helper locks***. A region helper lock protects a large critical section, called a ***parallel region***, which contains nested parallelism. In the hash table example, the resize operation should be enclosed in a region helper lock. A short helper lock is like an ordinary lock which normally protects a short serial critical section, but is linked to a region helper lock. In our design, when a worker p blocks while trying to acquire a helper lock ℓ, and some parallel region A either holds ℓ or the region helper lock linked to ℓ, p can help complete the region A. More specifically, our contributions are the following:

1. The ***parallel region lock runtime (PRL)***, which can execute series-parallel computations augmented with helper locks and parallel regions. While a parallel region is active, multiple workers may enter and help complete the region, either because they blocked on the region's helper lock, or due to random work stealing. PRL supports nested regions of unbounded nesting depth.

2. Theoretical bounds on the completion time and stack space usage of computations executed using PRL.

This work was done in collaboration with Charles E. Leiserson and Jim Sukha.

3. Language and runtime support for helper locks in MIT Cilk. The PRL design extends the Cilk language to allow users to spawn a function as a ***parallel region***, protected by a region helper lock. The runtime system extension uses "deque pools" and "deque chains" to support parallel regions.

In our theoretical work, we extend the results given in [ABP98, BL99] for work-stealing schedulers, and show that PRL completes a "deadlock free" computation (with region helper locks) on P processors in time $O\left(T_1/P + \widetilde{T}_\infty + PN + \lg(1/\epsilon)\right)$ with probability at least $1 - \epsilon$, where $\widetilde{T}_\infty$ is an "aggregate span" which is bounded by the sum of spans of all regions. Our completion time bounds are asymptotically optimal for certain computations with parallel regions and helper locks. In addition, the bounds imply that PRL provides linear speedup if all parallel regions in the computation are sufficiently parallel. Roughly, if for all regions A, the non-nested work of region A is asymptotically larger than P times the span of A, then PRL executes the computation with linear speedup. We also show that PRL completes $\mathcal{E}$ using only $O(P\widetilde{S}_1)$ stack space, where $\widetilde{S}_1$ is the sum over all regions A of the stack space used by A in a serial execution of the same computation $\mathcal{E}$.

As a proof-of-concept, we implemented a prototype of PRL in MIT Cilk, and used the system to implement a concurrent hash table which uses helper locks to protect resize operations. We performed some experiments which demonstrate that implementation of PRL is feasible.

The rest of the chapter is organized as follows. In Section 5.1, we motivate the usefulness of helper locks by discussing the resizable hash table example in greater detail, and we explain the shortcomings of existing languages such as MIT Cilk for this example. Section 5.2 presents our design for a work-stealing scheduler which supports parallel regions and helper locks. Section 5.3 proves theoretical bounds on the completion time and space usage for our design. In Section 5.4, we describe our prototype implementation of parallel regions and helper locks in MIT Cilk. Section 5.5 presents some experimental results on our prototype system for a simple concurrent hash table benchmark. Section 5.6 discusses related work.

5.1 Motivating Example

In this section, we motivate the utility of region helper locks in a dynamically multithreaded system by looking at example code for a resizable hash table. In particular, we present code written using MIT Cilk, an open-source fork-join parallel programming language. First, we briefly review key features of the Cilk language and runtime. Then, we discuss the challenges of using Cilk with ordinary locks to exploit parallelism within a hash table's resize operation.

Overview of MIT Cilk

We review the characteristics of MIT Cilk and its work-stealing scheduler using the sample program in Figure 5.1. This pseudocode shows a Cilk function that concurrently inserts n random keys into a resizable hash table.

Cilk extends C with two main keywords: `spawn` and `sync`. In Cilk, the `spawn` keyword specifies that a function can potentially execute in parallel with the ***continuation*** of the function, i.e., code which immediately follows the spawn statement. The `sync` keyword forces any code after the `sync` to execute only after all previous spawned functions in the function have completed.

```
1  cilk void rand_inserts(HashTable* H, int n) {
2    if (n <= 32) { random_inserts_serial(H, n); }
3    else {
4      spawn rand_inserts(H, n/2);
5      spawn rand_inserts(H, n-n/2);
6      sync;
7    }
8  }
9  void rand_inserts_serial(HashTable* H, int n) {
10   for (int i = 0; i < N; i++) {
11     int res; Key k = rand();
12     do {
13       res = try_insert(H, k, k);
14     } while (res == FAILED);
15     resize_table_if_overflow(H);
16   }
17 }
```

Figure 5.1: Example Cilk function which performs n hash table inserts, potentially in parallel. After every insert, the `rand_inserts` method checks whether the insert triggered an overflow and resizes the table if necessary.

In Figure 5.1, the `rand_inserts` function uses `spawn` and `sync` to perform n insert operations in parallel in a divide-and-conquer fashion.

The Cilk runtime executes a program on P workers, where P is specified at runtime. Conceptually, every worker maintains a double-ended queue, or ***deque***, which stores the work of the processor. When a worker spawns a function f, it pushes the continuation of f onto the bottom (tail) of its deque and continues working on f. When a worker completes its current work, it pops work from the bottom of its deque. When a worker's deque becomes empty, however, it chooses a victim worker uniformly at random and tries to ***steal*** work from the top (head) of the victim's deque.

One can use a reader/writer lock to implement a resizable concurrent hash table in Cilk, as shown in Figure 5.2. In this code, 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) are allowed to run in parallel, but a table resize can not execute in parallel with any insert.

Challenges for parallel regions

In Figure 5.2, one would like to be able to exploit parallelism within the resize operation, e.g., by spawning the `rehash_list` operations for each bucket in Line 20. Unfortunately, several factors prevent ordinary Cilk from efficiently exploiting this parallelism. First, if a worker p_1 performing a resize operation holds the table lock in writer mode, any other worker p_2 which tries to concurrently acquire the lock in reader mode will spin waiting for the resize to complete. Cilk currently has no "suspend" mechanism which allows the worker to temporarily stop execution of one function (insert) and begin stealing work from another function (resize).

Second, even if one modified Cilk to implement a suspend mechanism, worker p_2 is unlikely to randomly steal work from a resize operation, since work corresponding to `rehash_list` is likely

```
1  int try_insert(HashTable* H, Key k, void* value) {
2    int success = 0, overflow = 0;
3    success = table_try_read_lock(H);
4    if (!success) { return FAILED; }
5    int idx = hashcode(H, k);
6    List* L = H->buckets[idx];
7    success = list_try_lock(L);
8    if (!success) {release_read_lock(H); return FAILED;}
9    list_insert(L, k, value);
10   list_unlock(L);  release_read_lock(H);
11   return SUCCESS;
12 }

13 void resize_table_if_overflow(HashTable* H) {
14   if (is_overflow(H)) {
15     table_acquire_write_lock(H);
16     List** new_buckets;
17     int new_n = H->num_buckets*2;
18     new_buckets = create_buckets(new_size);
19     for (int i = 0; i < H->num_buckets; i++) {
20       rehash_list(H->buckets[i], new_buckets, new_n);
21     }
22     free_buckets(H->buckets);
23     H->buckets = new_buckets;
24     H->num_buckets = new_n;
25     release_write_lock(H);
26   }
27 }
```

Figure 5.2: Code for insert and resize for a concurrent hash table.

to be at the bottom of worker p_1's deque and steals occur at the top of the deque. Instead, p_2 is likely to steal work corresponding to a call to `rand_inserts` which will cause p_2 to block again, suspend, and try stealing again. Thus, this strategy has two drawbacks: it doesn't necessarily lead to p_2 being productive and it may cause p_2 to use excessive stack space due to repeatedly blocking and suspending tasks. The code in Figure 5.1 could use $\Omega(n)$ stack space, where n is the total number of inserts, if a constant fraction of the inserts are spawned while a resize occurs.

Finally, it is difficult to implement a deadlock-free design which simultaneously allows workers to suspend a blocked task and work-steal arbitrarily, and which supports nested locking. In particular, to prevent deadlock in programs with nested locks, in order to support the above strategy, the language must support continuations at all points when a function tries to acquire a lock. Without such support, only the worker who suspends a task can resume it. This requirement can lead to deadlocks (due to the scheduler), even if the program itself is deadlock-free, that is, the programmer acquired locks according to a fixed partial order. For example, say a worker p acquires a lock ℓ_1 which protects the execution of a function `F`. Then, suppose p inside `F` fails to acquire a lock ℓ_2, suspends, and randomly steals some other function `G`. If `G` also tries to acquire lock ℓ_1, we have a deadlock, since only p can complete `F` and release ℓ_1.

5.2 Design for Helper Locks

In this section, we present the parallel region lock (PRL) runtime, our design for supporting helper locks in a Cilk-like work stealing runtime system. First, we describe helper locks, which allow a program to express and exploit parallelism inside critical regions. We then provide an overview of how we support parallel regions. Finally, we describe how PRL supports nested regions. We present PRL in the context of the MIT Cilk, the system we used to implement our prototype. The design can be applied more generally, however, to other fork-join parallel languages which use a work-stealing scheduler.

Helper locks

In order to support parallelism inside critical sections, we propose adding ***helper locks*** to MIT Cilk. We modify the language to add two types of helper locks. A region helper lock, which protects a large and parallel critical section, is specified using the construct `spawn_region`. A short helper lock is like an ordinary lock and normally protects a short serial critical section, but it is linked to a region helper lock. If a worker p fails to acquire a short helper lock ℓ_1 which is linked to a region helper lock ℓ_2, then rather than spinning, it triggers the `help_region` construct for ℓ_2. This construct causes worker p to help complete the critical section currently protected by ℓ_2, or does nothing if ℓ_2 is not currently held.

Figure 5.3 shows pseudocode which modifies Figure 5.2 to use helper locks to exploit parallelism within the hash table resize. In Line 1, `L` is declared as the region helper lock for resize operation `H`. If the insert in Line 3 fails because the insert cannot acquire the necessary bucket lock, in Line 4, the current worker tries to help complete the region protected by `L` (if the resize lock `L` is held). Conceptually, each bucket lock is a short helper lock, linked to `L`. Finally, in Line 7, the worker uses the `spawn_region` construct to try to acquire lock `L`, and then spawn the function `resize_if_overflow` as a parallel region protected by `L`. The lock `L` is specified as the first

```
1  CilkRegionLock* L = H->resize_lock;
2  do {
3    res = try_insert(H, k, k);
4    if (res == FAILED) { help_region(L); }
5  } while (res == FAILED);
6  if (is_overflow(H)) {
7    spawn_region(resize_if_overflow)(L, H);  sync;
8  }
```

Figure 5.3: Pseudocode for resizable hash table using a helper lock `L`. This code represents a modification of the inner loop of the `rand_inserts` function (lines 12–15 of Figure 5.1).

argument to the region function. If `spawn_region` detects that `L` is already held, it implicitly performs a `help_region` call for `L` before trying to acquire `L` again.

Parallel regions

Conceptually, a ***parallel region*** is a subcomputation of a Cilk program, but with its own set of deques used for scheduling. In the context of helper locks, a parallel region is an execution instance of a critical region, protected by a region helper lock. Therefore, in the hash table example, each call to the resize function create a new parallel region, but all are protected by the same lock.

When the runtime system starts executing a parallel region A, it creates a ***deque pool*** for A, called $\texttt{dqpool}(A)$. At any point during its execution, a parallel region has certain workers ***assigned*** to it, and all these workers have deques in A's deque pool. Initially, when a worker p spawns a parallel region A, only p is assigned to the region A. As the program executes, more workers may be assigned to A. When a worker p is assigned to a parallel region A, the runtime system allocates a deque, $\texttt{dq}(p, A)$, to p from $\texttt{dqpool}(A)$. We say $\texttt{dq}(p, A)$ is `NULL` if p is not assigned to A. The runtime uses $\texttt{dqpool}(A)$ 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 in $\texttt{dqpool}(A)$.

In PRL, a worker can enter (be assigned to) a region A for three reasons. First, p_1 is assigned to A when p_1 successfully spawns region A. Second, p_1 may enter A when it calls `help_region` on a lock ℓ, and A holds lock ℓ. Finally, a worker p_1 can enter a region A because of random work stealing: when p_1 tries to steal from p_2, and discovers p_2 is assigned to A, p_1 may also enter A.

Conceptually, workers may also leave a parallel region A before A completes. Allowing workers to leave a region raises several issues, however. First, a worker p might repeatedly leave and enter the same region A, repeatedly incurring the synchronization overhead of entering and leaving A. In addition, we wish to maintain a property that work belonging to region A remains in A's deque pool, since otherwise, it becomes difficult for workers assigned to A to find A's work. Therefore, if a worker p is allowed to leave A while the deque $q = \texttt{dq}(p, A)$ is not empty, then p must abandon deque q, i.e., leave it without an assigned worker. If workers repeatedly abandon deques, then it may be difficult to limit the space used to maintain deque pools. For these reasons, in PRL, once a worker p is assigned to A, p remains in A until A finishes executing.

Nested regions

We would like PRL to support nested helper locks and parallel regions, i.e., a region A protected by region lock ℓ_1 should be able to spawn a region B protected by lock ℓ_2. In addition, once regions can be nested, the following scenario may occur. A worker p may enter a region A due to a `help_region` call on lock ℓ_1, and then enter region B due to a `help_region` call on lock ℓ_2. By maintaining a ***deque chain*** on every worker p, our design is able to support nesting of regions, with arbitrary nesting depth.

With nested regions, every worker p may be assigned to many parallel regions, and thus have many deques. In PRL, these deques of one processor form a chain, with each deque along the chain belonging to the deque pool of a distinct region.[1] The top deque in every worker's chain belongs to the global deque pool, which is the original set of deques for a normal Cilk program context. The bottom deque in p's chain represents p's ***active deque***, denoted $\texttt{activeDQ}(p)$. We let $\texttt{activeR}(p)$ denote the region for the pool that $\texttt{activeDQ}(p)$ belongs to. When p is working normally, it changes only the tail of $\texttt{activeDQ}(p)$. In addition, p always work-steals from deques within the deque pool of $\texttt{activeR}(p)$.

Whenever a worker p with active region A enters a region B, it adds a new deque for region B to the bottom of its chain, i.e., it adds $\texttt{dq}(p, B)$ as a child of $\texttt{dq}(p, A)$, and sets $\texttt{activeDQ}(p)$ to $\texttt{dq}(p, B)$. When B completes, for every worker p assigned to A, p removes its deque $q_p = \texttt{dq}(p, B)$ from the end of its deque chain, sets the parent of q_p as its active deque, and starts working on the region that q_p's parent belongs to. Note that different workers assigned to B may return to different regions.

Deque chains also help a worker p efficiently find deque pools for other regions during random work-stealing. A worker p_1 with an empty active deque randomly steals from other pools in the same region ($A = \texttt{activeR}(p_1)$). If p_1 finds a deque $q = \texttt{dq}(p_2, A)$ which is also empty, but which has a child deque q', then instead of failing the steal attempt, p_1 enters the region corresponding to q'. If q' is also empty but also has a child deque, etc., p_1 can continue to enter the regions for deques deeper in the chain.

Figure 5.4 illustrates deque pools and deque chains for a simple computation. In this example, worker 1 enters regions A through F (all regions are assumed to acquire different helper locks). Initially, worker 1 spawns a region B, worker 4 randomly steals from 1 in $\texttt{dqpool}(A)$, and spawns region D, and worker 2 steals from 4 in A, and spawns region F. Next, worker 1 spawns a region C nested inside B. Worker 3 then randomly work-steals from 1 in A, and enters B and C. Then, worker 1 inside C makes a `help_region` call on the lock for D, enters D, and steals from worker 4 in D. Finally, worker 1 makes a `help_region` call on the lock for F and enters F.

Deadlock freedom

As with ordinary (non-helper) locks, an arbitrary combination of `spawn_region` and `help_region` constructs with arbitrary locks can potentially introduce deadlock into a program. As with ordinary locks, however, maintaining a partial order on region locks is sufficient for PRL to guarantee that a program is deadlock-free.

[1] If a program deadlocks, the chain might contain multiple deques from the same region. One can potentially use this fact to detect deadlock.

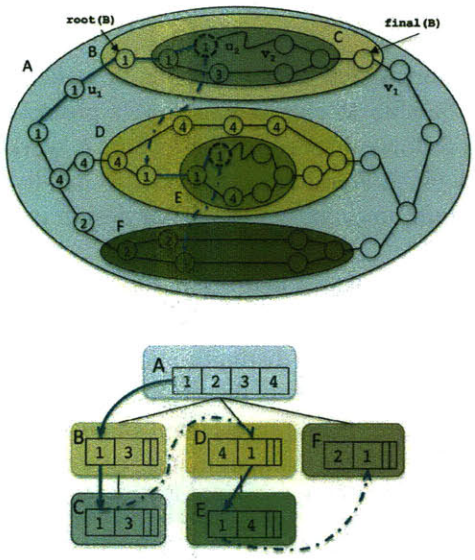

Figure 5.4: A computation dag with regions, and a snapshot of deque pools during execution.

Definition 1 *For a program with helper locks, construct a graph G graph of locks, with an edge from lock ℓ_1 to lock ℓ_2 if a worker in A_1 protected by ℓ_1 ever directly calls* `spawn_region` *or* `help_region` *for region A_2 protected by ℓ_2. We say that a program is* ***deadlock-free*** *if the graph G it generates is always acyclic.*

For programs with (region or short) helper locks which satisfy Definition 1, one can show the PRL does not introduce deadlock due to scheduling. Given Definition 1, we know the regions along every deque chain on every worker is consistent with the edges of G. Since random steals only cause a worker p to enter regions which are deeper in some worker's chain, steals do not introduce new edges into G. Thus, it is always possible for the worker with the "largest" $\texttt{activeR}(p)$ to complete its region.

5.3 Completion Time and Space Usage

In this section, we prove bounds on completion time and space usage for a computation with helper locks executed using PRL.

In order to state our results, we first define some terminology. We model the program as a computation dag $\mathcal{E}$. Each node in $\mathcal{E}$ represents a unit-time task, and each edge represents a dependence between tasks. As in [ABP98], we assume each node in $\mathcal{E}$ has degree at most 2. We model regions as subdags of $\mathcal{E}$. The entire computation $\mathcal{E}$ itself is considered a region, which encloses all other regions. Let $\texttt{regions}(\mathcal{E})$ denote the set of all regions enclosed within $\mathcal{E}$, and let $N = |\texttt{regions}(\mathcal{E})|$ denote the total number of regions in $\mathcal{E}$. For any node v, we say that a region A *contains* v if v is a node in A's dag. We say that v *belongs to* region A if A is the innermost region that contains v.

We assume that regions in $\mathcal{E}$ have a canonical structure which satisfies the following assumptions. First, the (sub)dag for each region A has a unique initial node called $\texttt{root}(A)$, and a unique final node called $\texttt{final}(A)$, such that all other nodes in the region are descendants of $\texttt{root}(A)$ and ancestors of $\texttt{final}(A)$. Second, the regions are properly nested; if A contains one node from B, then it contains all nodes from B. Third, for every region B which is directly nested inside a parent region A, $\texttt{final}(B)$ has outdegree 1, and the successor node of $\texttt{final}(B)$ (call it v) has indegree 1. Conceptually, v represents the resumption of A after B completes.

It is useful to prove results only for programs which are free from deadlock. We say that a computation $\mathcal{E}$ is ***deadlock-free*** if it was generated by a program which satisfies Definition 1.

For a given region A, the (total) ***work*** of region A, denoted $T_1(A)$, is defined as the number of nodes contained in A. We define the ***span*** of A, denoted $T_\infty(A)$, as the number of nodes along the longest path from $\texttt{root}(A)$ to $\texttt{final}(A)$. The span represents the time it takes to execute a region on infinite number of processors, assuming all nested regions are eliminated and flattened into the outer region (and ignoring any mutual exclusion requirements for locked regions). For the full computation $\mathcal{E}$, we leave off the superscript and say that $T_1 = T_1(\mathcal{E})$ and span is $T_\infty = T_\infty(\mathcal{E})$.

It is also useful to define a work and span for a region A which considers only nodes belonging to A. We define the ***region work*** of A, denoted $\tau_1(A)$, as the number of nodes in the dag which belong to A. For any path h through the graph $\mathcal{E}$, the ***path length for region A*** of h is the number of nodes u along path h which belong to A. We define the ***region span*** of a region A, denoted $\tau_\infty(A)$, as the maximum path length for A over all paths h from $\texttt{root}(A)$ to $\texttt{final}(A)$. Intuitively, the region span of A is the time to execute A on an infinite number of processors, assuming A's nested regions complete instantaneously. As an example, in the computation dag in Figure 5.4, region D has $T_1(D) = 13$, $\tau_1(D) = 7$, $T_\infty(D) = 8$, and $\tau_\infty(D) = 5$. Finally, we define the aggregate region span:

Definition 2 *For a computation $\mathcal{E}$ with parallel regions, define the* ***aggregate region span*** $\widetilde{T}_\infty$ *as* $\widetilde{T}_\infty = \sum_{A \in \texttt{regions}(\mathcal{E})} \tau_\infty(A)$.

To start with, we consider only computations without contention on short helper locks, i.e., we assume that no worker ever blocks due to a short lock. More precisely, we prove that PRL completes a deadlock-free computation $\mathcal{E}$ with work T_1 and aggregate span $\widetilde{T}_\infty$ on P processors in expected time $O(T_1/P + \widetilde{T}_\infty + PN)$. Moreover, for any $\epsilon > 0$, the execution time is $O(T_1/P + \widetilde{T}_\infty + PN + \lg(1/\epsilon))$ with probability at least $1 - \epsilon$. Later, we extend this bound to handle short helper locks.

We also prove a bound on space usage. Generalizing the notation in [BL99], we define the ***region stack depth*** of a node u and region A, denoted $\texttt{stackDepth}(A, u)$, as the sum of the sizes of all the frames of u's ancestors (including u itself) in the execution stack, considering *only* frames belonging to region A. Thus, $\texttt{stackDepth}(A, u) = 0$ if u does not belong to region A. For any region A and computation $\mathcal{E}$, we let $\mathcal{S}(A)$ be the maximum over all nodes $u \in \mathcal{E}$ of $\texttt{stackDepth}(A, u)$. Let $\mathcal{S}_1(A)$ be the minimum over all possible 1-processor executions of $\mathcal{S}(A)$. Finally, let $\widetilde{\mathcal{S}}_1 = \sum_{A \in \texttt{regions}(\mathcal{E})} \mathcal{S}_1(A)$. We show that for any deadlock-free computation $\mathcal{E}$, PRL executes $\mathcal{E}$ on P processors uses at most $O(P\widetilde{\mathcal{S}}_1)$ space. This space bound is not affected by the contention on short locks.

Execution

We use an execution model similar to the one described in [ABP98]. Let P denote the number of processors (workers) being used to execute the computation. A node v is ***ready*** if all of v's predecessors have already been executed. As in the original model of [ABP98], at any a time step when a processor p has work available, p has a currently *assigned* node u which is executed. If executing u makes one other node v ready, then v becomes p's assigned node. If executing u makes two nodes ready, then u represents a `spawn`, and one of u's successor's is pushed onto p's deque and the other successor is assigned. If u does not make any successor ready (this happens when u's successor v is a `sync` node and its other predecessor is not done), then p removes the bottom node w from its deque and assigns it. If p's deque is empty, then in the next step p becomes a ***thief***, chooses a ***victim*** deque uniformly at random, and attempts to steal work from this deque. If the deque is not empty, p steals the top node x from it and x becomes p's assigned node.

For PRL, in order to support nested parallel regions, we extend this execution model with regions by incorporating multiple deques and transitions of workers between regions. In the extended model, worker p's currently assigned node u always belongs to p's active region, $A = \texttt{activeR}(p)$. When p works on u, execution proceeds normally as in the original model, except that when p's deque is empty, it steals work from the deques in A's deque pool. Regions also introduce a few changes to the model:

1. Say p_1 in region A steals from a deque $q = \texttt{dq}(p_2, A)$. If q is empty (or "blocked", as described below), and q has a child deque $\texttt{dq}(p_2, B)$, then p_1 enters B by creating a deque $\texttt{dq}(p_1, B)$ and setting it as the active deque.

2. If u represents a `help_region` call on region B, we say that u is a ***helper node***. Let v be u's successor node; we call v the ***region successor*** of u. Then, v is said to be ***blocked on B***. This node v is considered blocked on region B until B completes.

3. If the assigned node u has a successor $\texttt{root}(B)$ for another region B (directly nested inside A) we define u's ***region successor*** v as the node v which is the successor of $\texttt{final}(B)$, and again we say that v is blocked on B. Note that by the canonical structure of $\mathcal{E}$, $\texttt{final}(B)$ can have only one successor node v which must belong to A. Therefore, this case is conceptually similar to the previous case, except there is a region between u and v.

In our execution model, we conceptually place the blocked node v at the bottom of the deque $\texttt{dq}(p, A)$, even though it is not ready to be executed. No worker can steal a blocked node v however. We say that v is ***invisible*** to other workers. If some worker p' tries to steal from $\texttt{dq}(p, A)$ and v is at the top of the deque, then p' is unable to steal v. Instead, p' enters the region B that v is blocked on. We call a deque $\texttt{dq}(p, A)$ with a blocked node on top a ***blocked deque***.

After blocking on a region B (due to either of the above reasons), p enters B by creating a deque $\texttt{dq}(p, B)$ in B's deque pool and making it the active deque. In our model, a worker p leaves a region B only when B completes. When p leaves a region B, it deallocates $\texttt{dq}(p, B)$, and sets $\texttt{dq}(p, B)$'s parent deque (say q) as its active deque. If q has a blocked node u as its bottom node, it removes this blocked node and assigns it.

Finally, we have to model synchronization costs. When a worker enters or leaves a region, it has to synchronize with other workers in that region. We pessimistically assume that it has to synchronize with all other workers. Therefore, we call a step on which any worker is entering or

leaving a region a ***waiting step***. A step when no worker is entering or leaving a region is called a ***running step***.

In summary, from the description of the execution model, one can prove the invariants in Lemma 5.1.

Lemma 5.1 *Let $v_1, v_2, \ldots, v_k$ be the nodes on a deque $\texttt{dq}(p, A)$, arranged from bottom to top. Let u_i be any predecessor of v_i if v_i is an unblocked node, or the region predecessor of v_i if v_i is a blocked node. An execution of a computation with helper locks maintains the following invariants for deques:*

1. *For all i, u_i is unique.*
2. *For all unblocked nodes v_i, u_i is a* `spawn` *node.*
3. *For all $i = 2, 3, \ldots, k$, u_i is an ancestor of u_{i-1} in $\mathcal{E}$.*
4. *If $\texttt{dq}(p, A)$ is an active deque with assigned node w, then w is a descendant of u_1.*
5. *For all i, v_i belongs to region A.*
6. *If v_i is blocked, then $i = 1$, i.e., a blocked node must be at the bottom of a deque.*
7. *Any blocked node v which belongs to region A must be in an inactive deque $\texttt{dq}(p, A) \neq \texttt{activeDQ}(p)$ for some worker p.*

PROOF.

Invariants 1 and 2 hold because nodes v are added to a deque only when a worker p works on an assigned node u representing a `spawn`, or when an assigned node u pushes its region successor v as a blocked node onto the deque.

Invariant 5 holds because the execution model only assigns a node u to worker p if u belongs to the active region $\texttt{activeR}(p)$, and because a `spawn` node u can only push nodes belonging to the same region.

One can show that Invariants 3 and 4 hold by induction on the actions of the execution model.

Invariants 6 and 7 hold because a blocked node is put on the deque $\texttt{dq}(p, A)$ only when p enters another region B, and entering a region always creates a new active deque. □

Potential function

We shall bound the completion time of a program with parallel regions by using a potential function argument. In order to define the potential, we first define the ***depth*** and ***weight*** of nodes. For every node u belonging to A, we define the depth of u, called $d(u)$, as the maximum path length *for region A* over all paths from $\texttt{root}(A)$ to u. In addition, we define a weight of a node belonging to region A, denoted by $w(A)$ as the $\tau_\infty(A) - d(A)$.[2]

As with the analysis in [ABP98], one can show that the weights of nodes along any deque strictly decreases from top to bottom.

[2] In [ABP98], depth and weight are defined in terms of an execution-dependent enabling tree. We use this simpler definition because we are only considering series-parallel computations $\mathcal{E}$.

Lemma 5.2 *Let $v_1, v_2, \ldots, v_k$ be the nodes in any deque $\mathtt{dq}(p, A_i)$ ordered from bottom of the deque to the top. In addition, let v_0 be the assigned node if $\mathtt{dq}(p, A_i) = \mathtt{activeDQ}(p)$. Then, we have $w(v_0) \leq w(v_1) < \ldots < w(v_k)$.*

PROOF. If the deque is not empty, then either v_1 is a blocked node, or an assigned node v_0 exists. Invariants 6 and 7 from Lemma 5.1 show that these two conditions are mutually exclusive. Define u_i's as in Lemma 5.1.

In the first case, suppose v_1 is a blocked node and v_0 does not exist. Since we assume $\mathcal{E}$ 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 2, for all the unblocked nodes $v_2, v_3, \ldots, v_k$ in the deque, $d(u_i) = d(v_i) - 1$. Similarly, we know that for a blocked node v_1, the region predecessor u_1 satisfies $d(u_1) = d(v_1) - 1$. By Invariant 3, we know that u_i is an ancestor of u_{i-1} in $\mathcal{E}$; thus $d(v_{i-1}) > d(v_i)$ for all $i = 2, 3, \ldots, k$. Converting from depth to weight, we get $w(v_1) < w(v_2) < \ldots < w(v_k)$.

In the second case, suppose v_0 does exist (and thus, the deque in question contains only unblocked nodes). Applying the same logic to the nodes on the deque as in the first case, we have $d(v_{i-1}) > d(v_i)$ for $i = 2, 3, \ldots, k$. By Invariant 4, we know v_0 is a descendant of u_1; thus, $d(v_0) > d(u_1) = d(v_1) - 1$. Thus, $d(v_0) \geq d(v_1)$. Converting from depth to weight reverses the inequalities, and we get $w(v_0) \leq w(v_1) < \ldots < w(v_k)$. □

As in [ABP98], we use the weight to define a potential function for ready or blocked nodes u, and for a region A. We define the potential of a node u belonging to region A, denoted by $\Phi(u)$, as $\Phi(u) = 3^{2w(u)-1}$ if u is assigned and $3^{2w(u)}$ if u is ready or blocked. The potential of an active deque $\mathtt{dq}(p, A) = \mathtt{activeDQ}(p)$ is defined as the $\sum_{v \in \mathtt{dq}(p,A)} \Phi(v) + \Phi(u)$, where u is p's assigned node. The potential of an inactive deque $\mathtt{dq}(p, A)$ is defined as $\sum_{v \in \mathtt{dq}(p,A)} \Phi(v)$. The potential of a region, $\Phi(A)$, is defined as the sum of all the potentials of all the deques in A's deque pool.

Lemma 5.3 *In the course of the computation, the potential of a particular region $\Phi(A)$ increases from 0 to $3^{2\tau_\infty(A)}$ when $\mathtt{root}(A)$ becomes ready. At all other times, there is no potential increase.*

PROOF.

The scheduler performs four actions that can change the potential. First, it may remove a node u (belonging to A) from the deque and assign it. Second, it may executes an assigned node u, which makes one or two children nodes from the same region A ready. In both these cases, the potential of only region $\Phi(A)$ is affected, and it always decreases by the argument given in [ABP98].

The third action is that a worker p may execute a node u from region A and enable (exactly one) node $w = \mathtt{root}(B)$ from region B. In our execution model, the node v which is the region successor of u blocks and is added to $\mathtt{dq}(p, A)$. Since v belongs to A, $\Phi(A)$ decreases because v is a successor of u and has a lower potential. At the same time, $\mathtt{dq}(p, B)$ becomes a new active deque for p and $\mathtt{root}(B)$ becomes ready, and the $\Phi(B)$ increases from 0 to $3^{2\tau_\infty(B)}$.

The fourth and final action occurs when a worker p executes the final node of a region, $\mathtt{final}(B)$, and enables a previously blocked node v belong to A. In this case, the potential of B decreases to 0. Since v is a blocked node, changing v from blocked to assigned results in a decrease in the potential $\Phi(A)$. □

Bounding steal attempts

We say that a steal attempt occurred in region A if the thief's active deque is in A's deque pool. In addition, we define a ***counted*** steal attempt as a steal attempt which occurs on a running step. To prove a bound on completion time, we need to bound the total number of counted steal attempts in all regions. We first bound the number of steal attempts in each region separately. Later, we aggregate the steal attempts from different regions.

To bound the number of counted steal attempts which occur in a region A, we divide steal attempts in A into phases. Intuitively, a phase $R_k(A)$ in region A consists of $O(P)$ counted steal attempts in region A . We classify phases as either "entering" or "contributing" phases. In our analysis, we amortize contributing phases against the potential of region A, and amortize the entering phases against the number of regions N.

We first define the boundaries of phases more precisely. The first phase $R_1(A)$ in region A begins when $\texttt{root}(A)$ is assigned to some worker. A ***phase*** $R_k(A)$ ends after at least P counted steal attempts in A have occurred, or some p executes $\texttt{final}(A)$. For $k > 1$, we call a phase $R_k(A)$ an ***entering phase*** if at the beginning of the phase, one of the deques in the A's deque pool is either blocked, or empty and inactive. Otherwise, we call $R_k(A)$ a ***contributing phase***. Note that any phase can have at most $2P - 1$ counted steal attempts, if P counted steals occur in the last time step of the phase.

First, we bound the number of contributing phases for a region A. At the beginning of phase k for A, let $E_k(A)$ be the sum of potentials on the due to empty or blocked deques in A's deque pool, and let $D_k(A)$ be the potential on all other deques. Therefore, the potential of A at the beginning of phase k is $\Phi_k(A) = E_k(A) + D_k(A)$.

Lemma 5.4 *If $R_k(A)$ is a contributing phase for A*

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

PROOF. First, we bound the potential decrease in $D_k(A)$. By definition, every phase (except possibly the last phase) has at least P steal attempts. Thus we can apply Lemma 8 from [ABP98] directly to claim that $D_k(A)$ decreases by a factor of $1/4$ with probability $1/4$. If $R_k(A)$ is a contributing phase, then at the beginning of phase k, A has no blocked deques, and any empty deques in A are active. For any empty and active deque q, the worker p assigned to q either had no assigned node (and thus contributes nothing to $E_k(A)$), or had an assigned node u which executes during the phase (since each phase contains at least one running time step), thereby reducing q's contribution to $E_k(A)$ by more than 1/4. The lemma holds trivially for the last phase. □

Lemma 5.5 *For a region A, the expected number contributing phases is $O(\tau_\infty(A))$.*

PROOF. The proof is analogous to Theorem 9 in [ABP98]. Call phase k *successful* if $\Phi_{k+1}(A) - \Phi_k(A) \geq \Phi_k(A)/4$, i.e., the potential decreases by at least $1/4$ fraction. From Lemma 5.4, we know $\Pr\{\Phi_{k+1}(A) \leq 3\Phi_k(A)/4\} \geq 1/4$, i.e., a phase is successful with probability $1/4$. The potential for region A starts at $3^{2\tau_\infty(A)}$, ends at 0, and is always an integer. Thus, a region A can have at

most $8\tau_\infty(A)$ successful phases. Thus, the expected number of phases needed to finish A is at most $32\tau_\infty(A)$. □

We now bound the number of entering phases. Note that at the beginning of an entering phase $R_k(A)$ for region A, there exists some deque $q = \mathtt{dq}(p, A)$ which is empty-and-inactive or blocked (and inactive). Let B be the region such that $\mathtt{dq}(p, B)$ is the child deque of q in the deque chain of p. We consider $R_k(A)$ a ***successful entering phase*** (for A) if during this phase, either B finishes, or some active worker p' in A enters B.

Lemma 5.6 *Any entering phase $R_k(A)$ is successful with probability at least $1/2$. In addition, the expected number of entering phases in a computation is at most $4PN$.*

PROOF. Let q be any deque which is empty-and-inactive or blocked at the beginning of phase k, and let B be the region such that $\mathtt{dq}(p, B)$ is the child deque of q. Until $\mathtt{dq}(p, B)$ finishes, any worker who tries to steal from q will enter into region B and the phase is successful. Also, if B finishes during the phase, then the phase is automatically successful. Thus, the phase can be unsuccessful only if B does not finish, at least P steal attempts occur, and all fail to hit q. Since each steal attempt hits q with probability $1/P$, the probability that none of the steal attempts hit q is $(1 - 1/P)^P \leq 1/e < 1/2$. Therefore, with probability at least $1/2$, the entering phase is successful.

Each worker can enter a region A at most once, since the scheduler does not allow workers to leave A until A is finished. Therefore, the total number of successful entering phases which cause a worker to enter any region is at most PN. Similarly, when a B finishes, since B has at most P deques, it can cause at most P successful entering phases for for other regions. Therefore, the total number of successful entering phases is at most $2PN$. Since each phase is successful with probability $1/2$, the expected number of entering phases is at most $4PN$.

□

Using Lemmas 5.5 and 5.6, we can bound the total number of counted steal attempts for a computation $\mathcal{E}$.

Lemma 5.7 *Using PRL, the expected number of counted steal attempts in entering phases is at most $O(P\widetilde{T}_\infty + P^2N)$. In addition, the number of counted steal attempts is $O(P\widetilde{T}_\infty + P^2N + P\lg(1/\epsilon))$ with probability $1 - \epsilon$.*

PROOF. Summing over all regions A, the expected number of phases is $32\widetilde{T}_\infty + 4PN$. Each phase contains at most $2P - 1$ counted steal attempts. Therefore, the expected number of counted steal attempts is at most $64P\widetilde{T}_\infty + 8P^2N$. For the high probability bound, say the execution takes more than $n = 32\widetilde{T}_\infty + 8PN + m$ contributing phases to A. Since each phase succeeds with probability at least $1/4$, the expected number of successes is at least $8\widetilde{T}_\infty + 4PN + m/4$. We can then use Chernoff bounds as in [ABP98] to prove the result by choosing $m = 32\widetilde{T}_\infty + 16\ln(1/\epsilon)$. □

Completion time bound

We are now ready prove Theorem 5.8, a bound on the completion time of a computation $\mathcal{E}$.

Theorem 5.8 *The parallel regions work-stealing scheduler completes a deadlock-free computation $\mathcal{E}$ with work T_1 and aggregate span $\widetilde{T}_\infty$ on P processors in expected time $O(T_1/P + \widetilde{T}_\infty + PN)$.*

Moreover, for any $\epsilon > 0$, the execution time is $O(T_1/P + \widetilde{T}_\infty + PN + \lg(1/\epsilon))$ with probability at least $1 - \epsilon$.

PROOF. We bound running and waiting steps separately. On running steps, no worker is entering or leaving a region.

On a running time step, a worker p can take one of two actions. First, p can be executing a node v which is ready. Second, p can be trying to steal from a deque in the pool of its active region, $\texttt{activeR}(p)$. In the first case, the total number of steps that workers can spend executing ready nodes is T_1. In the second case, by Lemma 5.4 and Lemma 5.7, the expected number of steal attempts is $P\widetilde{T}_\infty + P^2N$. Since there are P workers active on every step, the expected number of running steps is $O(T_1/P + \widetilde{T}_\infty + PN)$. The high probability bound also follows similarly.

To bound the number of waiting steps, we notice that a worker can enter and leave a region A at most once. Thus, there are at most $O(PN)$ waiting steps, even if each worker enters every region. □

Discussion of bounds

To understand Theorem 5.8, we can compare it to the completion time bound for a computation without regions. The ordinary bound for randomized work stealing [BL99] says that the expected completion time is $O(T_1/P + T_\infty)$. In Theorem 5.8, there are two differences.

First, there is an additive term of 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 program,. For example, in the hash table example from Section 5.1, if we performs n inserts, we expect to have only $O(\lg(n))$ parallel regions for resizes. Furthermore, even if there are a large number of parallel regions, if the each parallel region A is sufficiently big ($\tau_1(A) = \Omega(P^2)$) or long ($\tau_\infty(A) \geq \Omega(P)$), then the term $PN = O(T_1/P + \widetilde{T}_\infty)$, i.e., the PN term is asymptotically absorbed by the other terms. In general, we expect small critical sections to protected using short locks, not region locks.[3]

Second, Theorem 5.8 has the term $\widetilde{T}_\infty$ instead of T_∞. One can understand both the work-stealing bound and the parallel regions bound in terms of parallelism. The ordinary work-stealing bound means that the program gets linear speedup if $P = O(T_1/T_\infty)$. That is, the program gets linear speedup if the parallelism of the program is at least $\Omega(P)$. We can restate the PRL bound as follows:

$$O\left(\frac{T_1}{P} + \widetilde{T}_\infty + PN\right) = O\left(\sum_{A \in \texttt{regions}(\mathcal{E})} \left(\frac{\tau_1(A)}{P} + \tau_\infty(A)\right) + PN\right)$$

Using this restated bound, one can see that (ignoring the PN term), PRL provides linear speedup if the parallelism of each region is at least $\Omega(P)$. In addition, if the number of parallel regions is small (as we expect) then the term $\widetilde{T}_\infty$ is generally small as well. In the hash table example, a region A which can resize an array of length k completely in parallel would ideally have $\tau_\infty(A) = O(\lg(k))$. Then, for n inserts, $\widetilde{T}_\infty = O(\lg^2(n))$, whereas the total work is $\Theta(n)$.

[3]The PN bound is overly pessimistic for small parallel regions, since it assumes that all processors enter all critical sections. In fact, it is very unlikely that all processors will enter a small parallel region, and a small critical region will generally execute sequentially as if it was a short lock.

Time bound with short helper locks

We now add the contention on short helper locks into the analysis. In general, a short helper lock functions like a normal lock, and it is difficult to get interesting contention bounds on computations with locks, even in the absence of parallel regions. Here we present a simple bound just for completeness.

We define the ***bondage*** b of an execution graph $\mathcal{E}$ as the total number of nodes enclosed within short locks in $\mathcal{E}$. The bondage represents the amount of computation enclosed within short locks. Since short helper locks are designed to protect small critical sections, we assume that `spawn_region` or `help_region` calls never occur inside these critical sections. In our execution model, when a worker p holds a short lock ℓ, any other worker which tries to acquire ℓ waits for p to release ℓ if no parallel region holds the region lock linked to ℓ. We assume worst case contention and say that whenever a short lock ℓ is held by worker p, all other $P-1$ workers are waiting on ℓ. Thus, we consider any time step when any worker is waiting on a lock to be a waiting step (as we did earlier when a worker was entering or leaving a region). Then, we can bound the number of waiting steps due to short locks as follows:

Lemma 5.9 *For a deadlock-free computation, the total number of waiting steps due to short locks is at most b.*

PROOF. In any waiting step due to short locks, at least one worker is holding a short lock and is therefore executing some node that is enclosed within a short helper lock. Therefore, the remaining bondage (the number of unexecuted nodes enclosed in short helper locks) decreases by at least 1 after each waiting step due to short locks. There can be at most b such waiting steps. □

One can bound the completion time for a computation with both region and short helper locks by adding b to the bound in Theorem 5.8. Even though this bound seems simple, there exist computations for which this bound is asymptotically optimal. For example, if all parallel regions are protected by the same lock ℓ_1 and all short lock acquires are for the same lock ℓ_2, then asymptotically, one cannot do better than the bound we give (ignoring the PN term).

Space

Now we prove Theorem 5.10, which bounds the stack space.

Theorem 5.10 *Let $\widetilde{\mathcal{S}}_1 = \sum_{A \in \texttt{regions}(\mathcal{E})} \mathcal{S}_1(A)$. For a deadlock-free computation $\mathcal{E}$, PRL executes $\mathcal{E}$ on P processors using at most $O(P\widetilde{\mathcal{S}}_1)$ space.*

PROOF. At any point fixed in time, consider the tree T of active frames for the entire computation $\mathcal{E}$. For any region A, let T_A be the subset of T which consists of only frames which belong to A.

The only time a worker p stops working on its current active frame f in region A without completing f is if p enters a new region B. Also, a worker p can only enter a region A once. Using these two facts, one can show that the set T_A is in fact a tree with at most k leaves, where k is the number of workers which have entered A. This fact is a generalization of the busy-leaves property from [BL99], applied to only nodes of a specific region A. Thus, A uses at most $O\left(k\mathcal{S}_1(A)\right)$ stack space for T_A. In the worst case, all P workers have entered every region $A \in \texttt{regions}(\mathcal{E})$, and we must sum the space over all regions, giving us a space usage of $\widetilde{\mathcal{S}}_1 = \sum_{A \in \texttt{regions}(\mathcal{E})} \mathcal{S}_1(A)$. □

Comparison with alternative implementations

We now compare the bounds of our implementation of parallel regions with 2 other alternatives. The first option does not allow helping. When a worker p blocks on a lock ℓ, it just spins or waits for 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 $\Omega(T_1/P+\sum_{A\in\texttt{regions}(\mathcal{E})}\tau_1(A)+b)$. 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 when one uses helper locks.

Second, we can compare against an alternative we describe in Section 5.1, where if a worker p blocks on a lock, it suspends its current work, and then work steals normally. The implementation of this design may be easier than the one we propose in this chapter, since it does not need deque pools and chains of deques. Each worker maintains just one deque. As we mentioned in Section 5.1, this implementation may deadlock in the scheduler unless all region and short locks have continuations. However, even if each lock has continuations, this implementation still has the disadvantage that it may use more space than our design. In fact, even for programs with just one parallel region (and many short locks), this implementation may use $\Omega(T_1)$ space. In contrast, for one parallel region, our implementation uses $O(PT_\infty)$ space, which is much smaller than T_1 for reasonable parallel programs.

5.4 Prototype Implementation

This section describes our implementation of parallel regions and helper locks. We based our prototype on MIT Cilk, since it is readily available as open source[Sup03]. First, we review the existing implementation of deques in Cilk [FLR98]. Then, we discuss deque chains and deque pools, the two major additions to the runtime system needed to support parallel regions.

Cilk deques

Each Cilk deque is represented by pointers into a ***shadow stack***, a per-worker stack which stores frames corresponding to Cilk functions. For example, for the code in Figure 5.1, when a worker executes a call to `rand_inserts(H, 63)`, and executes the `spawn` in Line 4, it begins executing the first call to `rand_inserts(H, 31)`, and pushes a frame for the continuation of `rand_inserts(H, 63)` on to the shadow stack, marked so that some other worker can later steal this frame and resume execution at Line 5.

Each deque consists of three pointers that point to slots in the shadow stack: a tail pointer T, a head pointer H, and exception pointer E. The runtime uses the THE protocol described in [FLR98] to manage deques. T points to the first empty slot in the stack; when a worker pushes and pops frames onto its own deque, it modifies T. 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 where the worker should not pop above; if a worker working on the tail end encounters $E > T$, some exceptional condition has occurred, and control returns to the Cilk runtime.

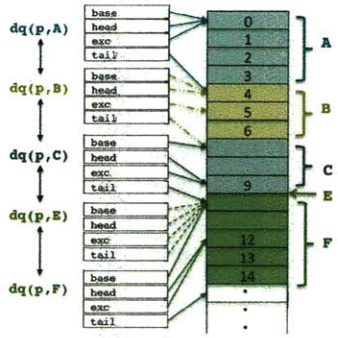

Figure 5.5: A chain of deques for a given worker p. Each deque $\texttt{dq}(p, A)$ consists of pointers (tail, head, and exception) into the p's stack of Cilk frames. For clarity, we show a pointer for the base of each deque q; in practice, q's base is always equal to the tail pointer of q's parent deque.

Deque chains

To implement parallel regions, we must conceptually maintain a chain of deques for each worker p. Usually, p modifies only its the bottom, active deque; however, other workers must be able to steal from any deque along p's chain. For example, in Figure 5.4, worker 1's deque chain has 6 deques, one for each region. Worker 2 can steal from $\texttt{dq}(1, F)$, while worker 4 can steal from $\texttt{dq}(1, E)$.

One straightforward implementation of deque chains is to allocate a separate shadow stack for each deque. Such a scheme, however, might allocate many stacks (i.e., PN instead of P). Dynamically allocating stacks every time a worker enters a region is potentially expensive. Statically allocating the correct number of shadow stacks is tricky because N might vary depending on an execution.

Instead, PRL maintains the entire chain of deques for a given worker p on the same shadow stack, as shown in Figure 5.5. 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 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, for a deque, the head H never grows upwards since H only changes when steals remove frames. Second, only the tail pointer T of the active deque $\texttt{activeDQ}(p)$ grows downwards, and $\texttt{activeDQ}(p)$ is the bottom deque on the deque chain.

Figure 5.5 shows an example arrangement of a deque chain on a worker's stack. In this example, no worker has stolen any frames from region B, while two frames have been stolen from region C. Worker p's deque for region E is empty, and $\texttt{activeR}(p) = F$.

Deque pool implementation

In our prototype, we implement 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 region lock. When a region is active, one or more slots of this array are occupied by workers.

To enter the pool, workers try to reserve the next available slot i in the array by using a

compare-and-swap operation to increment a size field from i to $i+1$. When a worker p tries a `spawn_region` call specifying a region helper lock ℓ, it succeeds in acquiring ℓ only if acquires slot 0 in the array. A worker which manages to reserve slot 0 is considered the ***master worker thread*** for the region. A p that tries a `help_region` call succeeds in adding itself as a helper to the region if it acquires a slot $i \geq 1$ in the array. When the region completes, each worker clears its deque in the lock's deque pool and increments a counter to signal its departure. The master worker thread waits until the last worker has left before it releases the lock (i.e., sets the size field to 0). Work stealing occurs within a region by choosing a slot in the deque pool uniformly at random, and trying to steal from that deque.

Our current 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 $\texttt{activeDQ}(p) = \texttt{dq}(p, A)$ tries enters another region B, it locks $\texttt{dq}(p, A)$, creates $\texttt{dq}(p, B)$ as $\texttt{dq}(p, A)$'s child deque, locks $\texttt{dq}(p, B)$ and then releases the lock on $\texttt{dq}(p, A)$ and finally releases the lock on $\texttt{dq}(p, B)$. Similarly, when a worker p leaves a region B, it first locks the parent of its active deque $\texttt{dq}(p, A)$, then locks $\texttt{dq}(p, B)$, deallocates $\texttt{dq}(p, B)$ and then releases the lock on $\texttt{dq}(p, A)$.

5.5 Hash Table Benchmark

This section presents some experimental results for an implementation of a resizable hash table, using our prototype of PRL. Although our hash table implementation is not optimized, we present these results, to show that implementation of PRL is feasible.

Hash table implementation

Our hash table table maintains an array of pointers for hash buckets, with each bucket implemented as a linked list. The table supports search, and an atomic `insert_if_absent` function. A search or insert locks the appropriate bucket in the table. When an insert causes the chain in a bucket to overflow beyond a certain threshold, it atomically increments a global counter for the table. An insert triggers a resize operation when more than a constant fraction of the buckets have overflowed.

A resize operation sets a flag marking that a resize operation is in progress, and then acquires all the bucket locks, scans the buckets to compute the current size of the table, doubles the size of the table until the density of the table is below a certain threshold, allocates a new array for buckets, and rehashes the elements from the old buckets into new buckets. We parallelized the lock acquires, the size computation, and the inserts into the new table.

We implemented two flavors of the hash table; the first flavor performs the resize operation serially, and the second spawns the resize operation as a parallel region, protected by a resize region helper lock. Each bucket lock functions as a short helper lock linked to this resize lock; if the resize lock is held, then an attempt to acquire a bucket lock cause workers to help resize.

Our benchmark performs n `insert_if_absent` operations on the hash table by spawning 20 functions, with each function performing $n/20$ inserts serially.[4] Keys are chosen uniformly at random, based on a deterministic seed chosen for each of the 20 functions. Since keys k are

[4]The number 20 is chosen arbitrarily, to be a number larger than P.

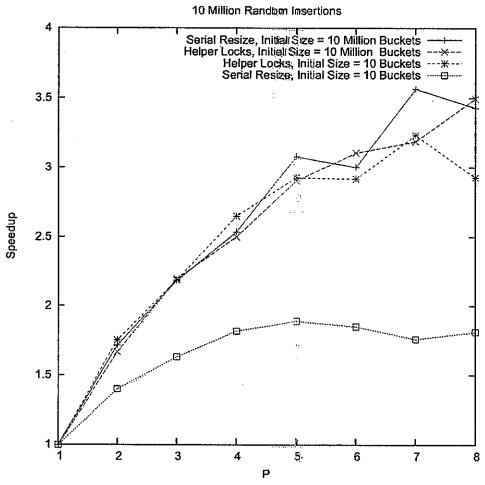

Figure 5.6: Results from an experiment inserting $n = 10^7$ random keys into a concurrent resizable hash table. Speedup is normalized to the runtime of the hash table with the same initial size and serial resize for $P = 1$. For tables with initial size of 10 and n buckets, the runtime was 8.66 s and 3.60 s, respectively. Each data point represents the average of 5 runs with the same parameters. This experiment was run on a two-socket machine, quad-core (3.16 GHz Intel Xeon X5460) machine with 8 GB RAM.

random, the benchmark uses a simple hash function of k modulo the number of buckets in the hash table.

Experimental results

Figure 5.6 shows results from performing insertions into the resizable hash table. We ran two versions of the experiment, one which contains no resize operations and one that does. In both experiments, the number of inserts is $n = 10^7$.

In the first experiment, the table began with 10^7 buckets; thus, with 10^7 inserts, no resize operations were triggered. In this experiment, the implementations with both serial and parallel resize were comparable, and both provided a speedup of about 3.5 on 8 processors. These results indicate that the overhead of for using a region lock in this application is relatively small. In the second experiment, the table began with 10 buckets, and the table size repeatedly doubled on resizes. In this case, the implementation which uses a parallel resize operation with region locks provided speedup of about 3. In contrast, the implementation that used the serial resize provided speedup of at most 2. This experiment indicates that there is some potential advantage to using region locks.

Note that in Figure 5.6, the plots for the two experiments (with and without resize) are not directly comparable to each other. The serial table which did not resize ran about 2.4 times faster than the serial hash table which does resize. This additional factor is approximately consistent with the amortized cost of table doubling. Conceptually, every insert in a resizable table pays 3 times the cost of a normal insert: once for the original insert, once to move it when the table is expanded, and once to move another item which has already been moved once.

Our current hash table implementation does not appear to scale beyond 4 processors, even when

no resizes occur. Thus, additional work is needed to improve scalability in this benchmark. In practice, a program will hopefully have a more realistic and scalable mix of concurrent operations (i.e., a combination of searches and inserts). Our primary goal for the benchmark, however, was to test and evaluate the feasibility of our design; thus, we tried to trigger resize operations as often as possible.

5.6 Related Work

In this section, we briefly review some related work.

OpenMP uses a `parallel` construct to support nested parallelism [Boa08]. Our implementation has some similarities with the implementation of this construct, although the design goal is different. Like in OpenMP, every parallel region in our implementation has a master (worker) thread, which is the first to enter the region, and which is guaranteed to resume execution after the region completes.[5] In OpenMP, the number of workers for a region is fixed when the region begins. In PRL, however, additional workers can enter the region, either through random work stealing, or because they are blocked on the lock for the region.

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 [Bar93, TSP92, IR94] 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 describe [LAK03] a concurrent B^{link} tree algorithm which uses cooperative locking to handle nodes with concurrent underflow.

[5] We made this decision for convenience in implementation. Our PRL design does not require a master thread.

Chapter 6

Memory Models for Transactions

This chapter describes a framework for defining and exploring memory semantics of transactional memory (TM) systems and mechanisms. This framework is inspired by the computation-centric framework proposed by Frigo [Fri98, FL98], allows TM semantics to be specified in an implementation-independent way. We will use this framework to both define existing TM memory models such as serializability, and to explore new memory models for TM namely ***race freedom*** and ***prefix-race freedom*** using this framework. We also use this framework to prove properties of these memory models.

In subsequent chapters (Chapters 7, 8, and 9), we continue to use this model to both describe new TM designs and to prove semantic properties of these designs. For example, in Chapter 7, we use the framework to describe the operational semantics of open-nesting and to prove that it provides the memory model we call prefix-race freedom. Similarly, in Chapter 8 we use the framework to prove that ownership-aware transactions provide the memory model we call ***serializability by modules***.

This chapter is organized as follows: Section 6.1 provides some background on transactional memory and nesting. Section 6.2 provides our framework for describing TM semantics. Section 6.3 defines sequential consistency using our framework and then generalizes this definition in the transactional setting. Section 6.4 formally defines the memory models of serializability, race freedom, and prefix-race freedom. In Section 6.5 we prove that all three memory models are equivalent for computations with only committed transactions, but are distinct when we model aborted transactions or have open transactions. Section 6.6 explores some related work on semantics of TM.

6.1 Background and Motivation

Atomic transactions represent a well-known and useful abstraction for programmers writing parallel code. Database systems have utilized transactions for decades [GR93]. Typically, ***serializability*** [Pap79] is used as a correctness condition for transactions. Under memory model of serializability, transactions affect global memory as if they were executed one at a time in some order, even if in reality, several executed concurrently. Transactional memory with either flat or closed nesting still guarantees serializability.

This work was done in collaboration with Charles E. Leiserson and Jim Sukha [ALS06].

Open nesting provides a loophole in the strict guarantee of transaction serializability by allowing an outer transaction to "ignore" the operations of its open subtransactions. Moss [Mos06] describes open nesting as a high-level methodology that incorporates an ***open-nested commit mechanism***, where the commit of a nested transaction I globally commits the memory locations accessed by it (as described in Section 1.2. In addition to the behavior of the commit mechanism, the methodology of open nesting requires high-level constructs such as abstract locks and "compensating actions" [Mos06]. For example, if a nested transaction I (nested inside transaction A) commits, and A later aborts, a compensating action for I may be executed in order to undo the changes made by I. Therefore, generally, open nesting can be viewed at two levels. At the memory level, open nesting is described by the open-nested commit mechanism. At the program level, it consists of other mechanisms that are involved in the methodology.

Indeed, even TM without any nesting can be viewed at two levels of abstraction. For example, the hardware may implement rollback of memory state, but rely on the programmer or compiler to retry transactions that abort, sometimes using backoff protocols to ensure that a given transaction eventually commits. Thus, it is helpful to distinguish the ***memory model*** for TM, as the essential memory semantics that the hardware (or the basic software) implements, from the ***program model***, as the semantics that the programmer sees.

In this chaper, our focus will be on memory models for TM. We shall not concern ourselves with retry mechanisms, compensating transactions, and the like. A TM system should have well-specified behavior even as a target for compilation, when all program-level support for transactions and nesting are put aside. Low-level software may build upon the memory model to provide a higher level of abstraction, e.g., for open nesting, but the semantics of open nesting must be understood by the programmers of this low-level software. Moreover, although one may ignore the semantics of aborted transactions at the program-model level, at the level of the memory model, even aborted transactions must have a reasonable semantics, at least up to the point where they abort. Thus, we shall be interested in defining memory semantics even for aborted transactions.

This chapter describes a framework for defining transactional memory models. Our framework, which is inspired by the computation-centric framework proposed by Frigo [Fri98, FL98], allows TM semantics to be specified in an implementation-independent way. Within this framework, we define the traditional model of serializability and two new transactional memory models, race freedom and prefix-race freedom. We prove that these three memory models are equivalent for computations that contain only closed transactions, as long as aborted transactions are "ignored." For systems that support open nesting, however, the three models are distinct. We will use these three models in subsequent chapters to understand semantics of TM implementations.

6.2 Transactional Computation Tree Framework

This section defines our framework for modeling transactional computations. Our model is inspired by Frigo's computation-centric modeling of a program execution as a computation dag (directed acyclic graph) [Fri98] with an "observer function" which essentially tells what write operation is "seen" by a read. Our model uses a "computation tree" to model both the computation dag and the nesting structure of transactions. We first define computation trees without transactions, then we show how transactions can be specified, and finally, we define Lamport's classical sequential-consistency model [Lam79].

Our computation-centric model focuses on an *a posteriori* analysis of a program execution. After a program completes, we assume the execution has generated a ***trace*** which is abstractly modeled as a pair (C, Φ), where C is a "computation tree" 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 (C, Φ).

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

Definition 3 *A **memory model** is a subset $\Delta \subseteq \mathcal{U}$.*

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

Computation trees without transactions

The computation tree C summarizes the information about the control structure of a program together with the structure of nested transactions. We first describe how a computation tree models the structure of a program execution in the special case where the computation has no transactions.

Structurally, a ***computation tree*** C is an ordered tree with two types of nodes: ***memory-operation nodes*** $\texttt{memOps}(C)$ at the leaves, and ***control nodes*** $\texttt{spNodes}(C)$ as internal nodes. Let $\texttt{nodes}(C) = \texttt{memOps}(C) \cup \texttt{spNodes}(C)$ denote the set of all nodes of C.

We define $\mathcal{M}$ to be the set of all memory locations. Each leaf node $u \in \texttt{memOps}(C)$ 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.

The internal nodes $\texttt{spNodes}(C)$ of C represent the parallel control structure of the computation. In the manner of [FL97], each internal node $X \in \texttt{spNodes}(C)$ is labeled as either an S-node or P-node to capture fork/join parallelism. All the children of an S-node are executed in series from left to right, while the children of an P-node can be executed in parallel.

Several structural notations will help. Denote the ***root*** of a computation tree C as $\texttt{root}(()C)$. For any internal node $X \in \texttt{spNodes}(C)$, let $\texttt{children}(X)$ denote the ordered set of X's children. For any tree node $X \in \texttt{nodes}(C)$, let $\texttt{ances}(X)$ denote the set of all ancestors of X in C, and let $\texttt{desc}(X)$ denote the set of all X's descendants. Denote the set of proper ancestors (and descendants) of X by $\texttt{pAnces}(X)$ (and $\texttt{pDesc}(X)$). Denote the ***least common ancestor*** of two nodes $X_1, X_2 \in C$ by $\texttt{LCA}(X_1, X_2)$.

Since every subtree of a computation tree is also a computation tree, we shall sometimes overload notation and use a subtree and its root interchangeably. For example, if $X = \texttt{root}(()C)$, then $\texttt{memOps}(X)$ refers to all the leaf nodes in C, and $\texttt{children}(()C)$ refers to the children of X.

Computation dags

A computation tree C defines a ***computation dag*** $G(C) = (V(C), E(C))$ constructed as follows and illustrated in Figure 6.1. For every internal node $X \in \texttt{spNodes}(C)$, we create and place two corresponding vertices, $\texttt{begin}(X)$ and $\texttt{end}(X)$ in $V(C)$. For every leaf node $x \in \texttt{memOps}(C)$, we place the single node x in $V(C)$. For convenience, for all $x \in \texttt{memOps}(C)$, we define $\texttt{begin}(x) = \texttt{end}(x) = x$.

Formally, the vertices of the graph $V(C)$ are defined as follows:

$$V(C) = \texttt{memOps}(C) \cup \left(\bigcup_{X \in \texttt{spNodes}(C)} \{\texttt{begin}(X), \texttt{end}(X)\} \right) .$$

For any computation tree rooted at node X, we define the edges $E(X)$ for the graph $G(X)$ recursively:
Base case: If $X \in \texttt{memOps}(C)$, then define $E(X) = \emptyset$.
Inductive case: If $X \in \texttt{spNodes}(C)$, let $\texttt{children}(X) = \{Y_1, Y_2, \ldots, Y_k\}$. If X is an S-node, then

$$E(X) = \{(\texttt{begin}(X), \texttt{begin}(Y_1)), (\texttt{end}(Y_k), \texttt{end}(X))\} \cup \left(\bigcup_{i=1}^{k-1} \{(\texttt{end}(Y_i), \texttt{begin}(Y_{i+1}))\} \right) \cup \left(\bigcup_{i=1}^{k} E(Y_i) \right) .$$

If X is a P-node, then

$$E(X) = \left(\bigcup_{i=1}^{k} E(Y_i) \right) \cup \left(\bigcup_{i=1}^{k} \{(\texttt{begin}(X), \texttt{begin}(Y_i)), (\texttt{end}(Y_i), \texttt{end}(X))\} \right) .$$

We shall find it convenient to overload the LCA function, and define the least common ancestor of two graph vertices $u, v \in V(C)$ as the LCA of the corresponding tree nodes.

The computation dag $G(C)$ is a convenient way of representing the flow of the program execution specified by C. Unfortunately, our specification of computation dags via computation trees limits the set of computation dags that can be described. In particular, computation trees can only specify "series-parallel" dags [FL97]. We might have founded our framework for transactional-memory semantics on more-general computational dags, but the added generality would not affect any of our theorems, and it would have greatly complicated definitions and proofs.

We shall find it useful to define some graph notations. For a graph $G = (V, E)$ and vertices $u, v \in V$, we write $u \preceq_G v$ if there exists a path from u to v in G, and we write $u \prec_G v$ if $u \neq v$ and $u \preceq_G v$. For any dag $G = (V, E)$, a ***topological sort*** $\mathcal{S}$ of G is an ordering of all the vertices of V such that for all $u, v \in V$, we have $u \prec_G v$ implies that $u <_{\mathcal{S}} v$ (u comes before v in $\mathcal{S}$). For a dag G, we define $\texttt{topo}(G)$ as the set of all topological sorts of G.

Classical theories on serializability refer to a particular execution order for a program as a ***history*** [Pap79]. In our framework, a history corresponds to a topological sort $\mathcal{S}$ of the computation dag $G(\mathcal{C})$. We define our models of TM using these sorts. Reordering a history to produce a serial history is equivalent to choosing a different topological sort $\mathcal{S}'$ of $G(\mathcal{C})$ which has all transactions appearing contiguously, but which is still "consistent" with the observer function associated with $\mathcal{S}$.

Transactional computation trees

We can specify transactions in a computation tree C by marking internal tree nodes. Marking a node $T \in \texttt{spNodes}(C)$ as a transaction corresponds to defining a transaction T that contains the

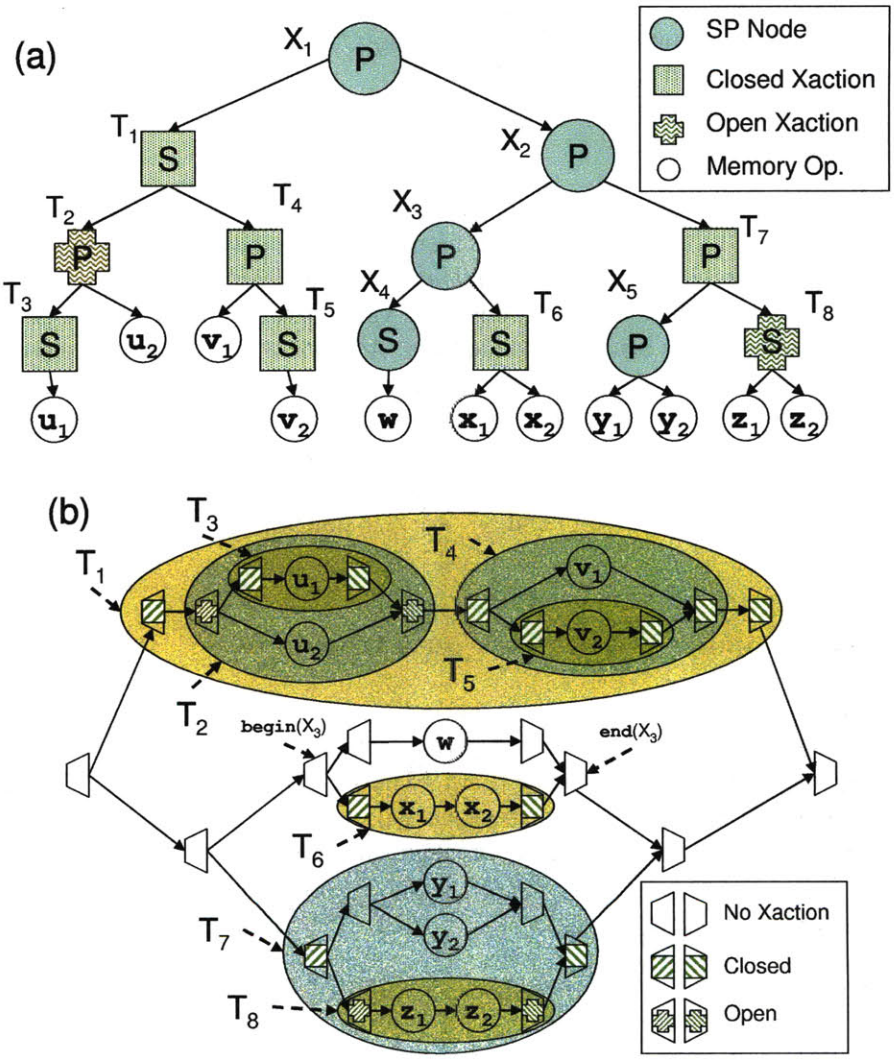

Figure 6.1: A sample (a) computation tree C and (b) the corresponding dag $G(C)$ for a computation that has closed and open transactions. In this example, T_2 is open-nested inside T_1 and T_8 is open-nested inside T_7. The X_i's are tree nodes that are not marked as transactions. We have not specified whether each transaction is committed or aborted.

computation subdag $G(T)$, where $\texttt{begin}(T)$ is the start of the transaction and $\texttt{end}(T)$ is the end of the transaction.[1] Formally, the computation tree C specifies a set $\texttt{xactions}(C) \subseteq \texttt{spNodes}(C)$ of internal nodes as ***transactions***, and a set $\texttt{open}(C) \subseteq \texttt{xactions}(C)$ of ***open*** transactions. The set of ***closed*** transactions is $\texttt{closed}(C) = \texttt{xactions}(C) - \texttt{open}(C)$. In Figure 6.1, nodes T_1 through T_8 are transactions, and X_1 through X_5 are ordinary nodes. Define a transaction $T \in \texttt{xactions}(C)$ as ***nested*** inside another transaction $T' \in \texttt{xactions}(C)$ if $T' \in \texttt{ances}(T)$. Two transactions T and T' are ***independent*** if neither is nested in the other.

Observer functions

Instead of specifying the value that a vertex $v \in \texttt{memOps}(C)$ reads from or writes to a memory location $\ell \in \mathcal{M}$, we follow Frigo's computation-centric framework [Fri98, FL98] which abstracts away the values entirely. An ***observer function***[2] $\Phi(v) : \texttt{memOps}(C) \to \texttt{memOps}(C) \cup \{\texttt{begin}(C)\}$ tells us which vertex $u \in \texttt{memOps}(C)$ writes the value of ℓ that v sees. For a given computation tree C, if $v \in \texttt{memOps}(C)$ accesses location $\ell \in \mathcal{M}$, then a well-formed observer function must satisfy $\neg(v \prec_{G(C)} \Phi(v))$ and $W(\Phi(v), \ell)$. In other words, v can not 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 $\texttt{begin}(C)$ writes initial values to all of memory.

Together, a computation tree C and an observer function Φ defined on $\texttt{memOps}(C)$ specify a trace.

6.3 Transactional Sequential Consistency

This section uses the computation tree framework to describe sequential consistency and then generalizes the definition to transactional programs. First, we use our framework to define Lamport's classic model of sequential consistency [Lam79] in our transactional model. We then define some notation in order to define transactional sequential consistency. Transactional sequential consistency is the basis of our definition for more interesting transactional memory models.

Sequential consistency without transactions

We now follow Frigo [Fri98] in defining a "last-writer" observer function.

Definition 4 *Consider a trace (C, Φ) with no transactions and a topological sort $\mathcal{S} \in \texttt{topo}(G(C))$. For all $v \in \texttt{memOps}(C)$ 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 $u \in \texttt{memOps}(C) \cup \{\texttt{begin}(C)\}$ that satisfies three conditions:*

1. *$W(u, \ell)$,*
2. *$u <_{\mathcal{S}} v$, and*

[1] We assume that every leaf $x \in \texttt{memOps}(C)$ is its own committed, closed transaction, but we do not mark leaves as a transactions in our model.

[2] Our definition of Φ is similar to Frigo's [Fri98], but with a salient difference, namely, Frigo's observer function gives values for all memory locations, not just for the location that a vertex accesses. Moreover, if $W(v, \ell)$, Frigo defines $\Phi(v) = v$, whereas we define $\Phi(v) = u$ for some $u \neq v$.

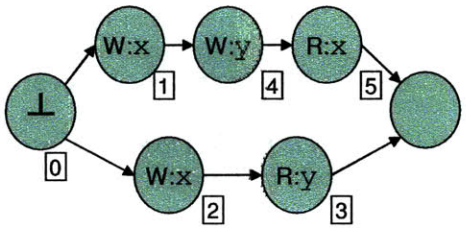

Figure 6.2: Examples of sequential consistency for a computation C with only committed transactions. Shown is the computation dag $G(C)$. For the observer function Φ_1 given by $\langle \Phi_1(1) = 0, \Phi_1(2) = 1, \Phi_1(3) = 0, \Phi_1(4) = 0, \Phi_1(5) = 2 \rangle$, the trace (C, Φ_1) is sequentially consistent, with the topological sort $\mathcal{S} = \langle 0, 1, 2, 3, 4, 5 \rangle$ of $G(C)$. For the observer function Φ_2 given by $\langle \Phi_2(1) = 0, \Phi_2(2) = 1, \Phi_2(3) = 0, \Phi_2(4) = 0, \Phi_2(5) = 1 \rangle$, however, the trace (C, Φ_2) is not sequentially consistent, because there is no topological sort consistent with the last-writer function.

3. $\neg \exists w \; s.t \; W(w, \ell) \wedge (u <_{\mathcal{S}} w <_{\mathcal{S}} v)$.

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 ***Sequential consistency*** *for computations without transactions is the memory model*

$$SC = \{(C, \Phi) : \exists \mathcal{S} \in \texttt{topo}(G(C)) \; s.t. \; \Phi = \mathcal{L}_{\mathcal{S}}\} \; .$$

By this definition, a trace (C, Φ) with no transactions is sequentially consistent if there exists a topological sort $\mathcal{S}$ of $G(C)$ such that the observer function Φ satisfies $\Phi(v) = \mathcal{L}_{\mathcal{S}}(v)$ for all memory operations $v \in \texttt{memOps}(C)$. Definition 5 captures Lamport's notion [Lam79] of sequential consistency: there exists a single order on all operations that explains the execution of program. Figure 6.2 shows a sample computation dag $G(C)$ and two possible observer functions, Φ_1 and Φ_2. The trace (C, Φ_1) is sequentially consistent, but (C, Φ_2) is not.

Transaction Contents

The computation tree C also specifies a set $\texttt{committed}(C) \subseteq \texttt{xactions}(C)$ of ***committed*** transactions. Similarly, transactions belonging to $\texttt{aborted}(()X) = \texttt{xactions}(X) - \texttt{committed}(X)$ are ***aborted*** transactions. First, we define some notation to classify committed transactions as either closed or open. Define the set of closed, committed transactions as

$$\texttt{cCom}(C) = \texttt{committed}(C) - \texttt{open}(C),$$

and the set of open, committed transactions as

$$\texttt{oCom}(C) = \texttt{committed}(C) \cap \texttt{open}(C).$$

To specify content sets, we require the following definition.

Definition 6 *For any* $T \in \texttt{xactions}(\mathcal{C})$ *and a memory operation* $u \in \texttt{memOps}(T)$, *let*

$$Q(u,T) = \texttt{xAnces}(u) - \texttt{ances}(T).$$

Define $q(u,T)$ *as the* $T^* \in Q(u,T)$ *which is the closest ancestor of* u *such that* $T^* \notin \texttt{cCom}(\mathcal{C})$, *or null if no such* T^* *exists. In other words, if* T^* *exists, then we have*

$$(\texttt{xAnces}(u) - \texttt{ances}(T^*)) \subseteq \texttt{cCom}(\mathcal{C})$$

but $T^* \notin \texttt{cCom}(\mathcal{C})$.

Using $q(u,T)$, we define the content sets as follows.

Definition 7 *At a time t, for any* $T \in \texttt{xactions}(\mathcal{C})$, *define* ***closed content*** *set* $\texttt{cContent}(T)$

$$= \{u \in \texttt{memOps}(\mathcal{C}) \ : \ q(u,T) = \mathit{null}\}.$$

Definition 8 *At a time t, for any* $T \in \texttt{xactions}(\mathcal{C})$, *define* ***open content*** *set* $\texttt{oContent}(T)$

$$= \{u \in \texttt{memOps}(\mathcal{C}) \ : \ q(u,T) \in \texttt{oCom}(\mathcal{C})\}.$$

Definition 9 *At a time t, for any* $T \in \texttt{xactions}(\mathcal{C})$, *define* ***aborted content*** *set* $\texttt{aContent}(T)$

$$= \{u \in \texttt{memOps}(\mathcal{C}) \ : \ q(u,T) \in \texttt{aborted}(\mathcal{C})\}.$$

The intuition behind the closed content set for a transaction T is that a memory operation $u \in \texttt{memOps}(T)$ belongs to T's closed content set if every transaction in $\texttt{ances}(u)$ up to, but not including T, is a closed and committed transaction. This case occurs only if $q(u,T)$ is *null*. Since each instruction belongs to one of the three content sets, we can say

$$\texttt{cContent}(T) = V(T) \ - \bigcup_{Z \in \texttt{open}(T) - \{T\}} V(Z) \ - \bigcup_{Z \in \texttt{aborted}(()T) - \{T\}} V(Z).$$

We always have $\texttt{cContent}(T) \subseteq V(T)$, and equality holds when T's subtree contains no open or aborted transactions.[3] For example, in Figure 6.1, memory operations u_1 and u_2 do not belong to $\texttt{cContent}(T_1)$, because T_2 is an open transaction nested within T_1. As another example from the figure, we have $v_2 \in \texttt{cContent}(T_4)$ if and only if $T_5 \in \texttt{committed}(C)$.

We also define the ***holders*** of a vertex $v \in V(C)$ to be the set

$$\texttt{h}(v) = \{T \in \texttt{xactions}(C) : u \in \texttt{cContent}(T)\}$$

of all transactions that contain v.

[3] We consider only ***global open nesting***, meaning that if T' is open-nested in T, then it is open with respect to every transaction in $\texttt{ances}(T)$. Alternatively, one might specify T' as ***open-nested with respect to*** an ancestor transaction T. In this case, the operations of T' are excluded from all transactions T'' on the path from T' up to and including T, but included in transactions that are proper ancestors of T. Intuitively, if T' is open-nested with respect to T, then T' commits its changes to T's context rather than directly to memory. Global open-nesting is then the special case when all open transactions are open with respect to $\texttt{root}(()C)$. As far as we know, there are no implementations of non-global open nesting.

Hidden vertices

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 a another aborted transaction. We take the position that up to the point that a transaction aborts, it should be "well behaved" and act as if it would commit. The well-behavedness of aborted transactions is implicitly assumed by the various proposals for open nesting [MCC+06, MH05, Mos06]. 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.

Definition 10 *For any two vertices $u, v \in V(C)$, let $X = \texttt{LCA}(u, v)$. We say that u is **hidden** from v, denoted uHv, if*

$$u \in \bigcup_{Y \in \texttt{aborted}(()X) - \{X\}} \texttt{cContent}(Y) .$$

In Figure 6.1, we have $v_2 H z_2$ if and only if at least one of T_1, T_4, or T_5 belongs to $\texttt{aborted}(()C)$. Since T_2 is an open transaction, however, we never have $u_1 H z_2$ if $T_2, T_3 \in \texttt{committed}(C)$, even if $T_1 \in \texttt{aborted}(()C)$. If we have $T_1, T_4 \in \texttt{committed}(C)$ and $T_7 \in \texttt{aborted}(()C)$, then we also have $y_1 H v_1$, but not $v_1 H y_1$, and thus the hidden relation H is not symmetric.

Transactional sequential consistency

We now extend the definition of sequential consistency to account for transactions. Our definition does not attempt to model atomicity, however — that is the topic of Section 6.4. It simply models that a transaction outside an aborted transaction cannot "see" values written by the aborted transaction. Moreover, our definition makes the assumption that an aborted computation is consistent up to the point that it aborts.

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 11 *Consider a trace $(C, \Phi) \in \mathcal{U}$ and a topological sort $\mathcal{S} \in \texttt{topo}(G(C))$. For all $v \in \texttt{memOps}(C)$ 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 $u \in \texttt{memOps}(C) \cup \{\texttt{begin}(C)\}$ that satisfies four conditions:*

1. $W(u, \ell)$,
2. $u <_{\mathcal{S}} v$,
3. $\neg(uHv)$, *and*
4. $\forall w \, (W(w, \ell) \wedge (u <_{\mathcal{S}} w <_{\mathcal{S}} v)) \implies wHv$.

The first two conditions for the transactional last-writer function $\mathcal{X}$ are the same as for the last-writer function $\mathcal{L}$. The third and fourth conditions of Definition 11 parallel the third condition of Definition 4, 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, except that the last-writer function $\mathcal{L}_S$ is replaced by the transactional last-writer function $\mathcal{X}_S$.

Definition 12 ***Transactional sequential consistency*** *is the memory model*

$$TSC = \{(C, \Phi) \in \mathcal{U} : \exists S \in \texttt{topo}(G(C)) \text{ s.t. } \Phi = \mathcal{X}_S\} \ .$$

6.4 Transactional Memory Models

In this section, we use our framework to define three different transactional memory models: serializability, race freedom, and prefix-race freedom. The intuition behind all three memory models is to find a single linear order $\mathcal{S}$ on all operations that both "explains" all memory operations and provides guarantees about every transaction. Serializability requires that all transactions appear as contiguous in $\mathcal{S}$. Race freedom weakens serializability by allowing transactions that do not "conflict" to interleave their memory operations in $\mathcal{S}$. Finally, prefix-race freedom weakens race freedom by only prohibiting conflicts with the prefix of a transaction.

Serializability

Serializability [Pap79] is the standard correctness condition for transactional systems.

Definition 13 *The* ***serializability*** *transactional memory model, ST, is the set of all traces $(C, \Phi) \in \mathcal{U}$ for which there exists a topological sort $\mathcal{S} \in \texttt{topo}(G(C))$ that satisfies two conditions:*

1. $\Phi = \mathcal{X}_S$, *and*
2. $\forall T \in \texttt{xactions}(()C)$ *and* $\forall v \in V(C)$, *we have* $\texttt{begin}(T) \leq_S v \leq_S \texttt{end}(T)$ *implies* $v \in V(T)$).

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 $\mathcal{X}_S$, and for every transaction T, the vertices in $V(T)$ appear contiguous in $\mathcal{S}$.

Race freedom

Our definition of race freedom is motivated by the observation that actual TM implementations allow independent transactions to interleave their executions provided that one transaction does not try to write to a memory location accessed by the other transaction. Normally, with only closed-nested transactions and ignoring operations from aborted transactions, we expect to be able to rearrange any interleaved execution order allowed by race freedom into an equivalent serializable order. As we shall see in Section 6.5, the two models are indeed equivalent for computations having

only closed and committed transactions. With aborted and open transactions in the model, however, we shall discover that the models are distinct.

To define race freedom, we first describe 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 14 *Let C be a computation tree, and suppose that $\mathcal{S} \in \texttt{topo}(G(C))$ is a topological sort of $G(C)$. A **(transactional) race** with respect to $\mathcal{S}$ occurs between $v \in V(C)$ and $T \in \texttt{xactions}(C)$, denoted by the predicate $\text{RACE}_{\mathcal{S}}(v, T)$, if $v \notin V(T)$ and there exists a $w \in \texttt{cContent}(T)$ 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. $\texttt{begin}(T) \leq_{\mathcal{S}} v \leq_{\mathcal{S}} \texttt{end}(T)$.

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 T and a vertex $v \notin V(T)$ appearingbetween $\texttt{begin}(T)$ and $\texttt{end}(T)$ in $\mathcal{S}$ if v "conflicts" with some vertex $u \in \texttt{cContent}(T)$, where by "conflicts," we mean that v writes to a location that u reads or writes, or vice versa.

We can now define race freedom.

Definition 15 *The **race-free** transactional memory model RFT is the set of all traces $(C, \Phi) \in \mathcal{U}$ for which there exists a topological sort $\mathcal{S} \in \texttt{topo}(G(C))$ satisfying two conditions:*

1. $\Phi = \mathcal{X}_{\mathcal{S}}$, *and*
2. $\forall v \in V(C)$ *and* $\forall T \in \texttt{xactions}(C)$, $\neg\text{RACE}_{\mathcal{S}}(v, T)$.

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 $\texttt{begin}(T)$ and $\texttt{end}(T)$ in $\mathcal{S}$, but only provided no race between v and T exists.

Prefix-race freedom

The notion of a prefix-race is motivated by the operational semantics of TM systems. As two transactions T and T' execute, if T' discovers a memory-access conflict between a vertex $v \in T'$ and T, then the conflict must be with a vertex in T that has already executed, that is, the prefix of T that executes before v. For prefix-race freedom, no such conflicts may occur.

Definition 16 *Let C be a computation tree, and let $\mathcal{S} \in \texttt{topo}(G(C))$ be a topological sort of $G(C)$. A **(transactional) prefix-race** with respect to $\mathcal{S}$ occurs between $v \in V(C)$ and $T \in \texttt{xactions}(C)$, denoted by the predicate $\text{PRACE}_{\mathcal{S}}(v, T)$, if $v \notin V(T)$ and there exists a $w \in \texttt{cContent}(T)$ 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. $\texttt{begin}(T) \leq_{\mathcal{S}} w <_{\mathcal{S}} v \leq_{\mathcal{S}} \texttt{end}(T)$.

Thus, this definition is identical to Definition 14, except that the potential conflicting vertex w must occur before v in $\mathcal{S}$.

The notion of a prefix-race gives rise to an corresponding memory model in which prefix-races are absent.

Definition 17 *The **prefix-race-free** transactional memory model $PRFT$ is the set of all traces $(C,\Phi) \in \mathcal{U}$ for which there exists a topological sort $\mathcal{S} \in \texttt{topo}(G(C))$ satisfying two conditions:*

1. $\Phi = \mathcal{X}_{\mathcal{S}}$, *and*

2. $\forall v \in V(C)$ *and* $\forall T \in \texttt{xactions}(C)$, $\neg \textsc{Prace}_{\mathcal{S}}(v,T)$.

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 T that appear before v in $\mathcal{S}$. If a "nontransactional" leaf node $v \in \texttt{memOps}(C)$ runs in parallel with a transaction T, all of Definitions 13, 15, and 17 check whether v interleaves within T's execution. Thus, these models can be thought of as guaranteeing "strong atomicity" in the parlance of Blundell, Lewis, and Martin [BLM05]. In Scott's model [Sco06], $\textsc{Race}_{\mathcal{S}}(v,T)$ and $\textsc{Prace}_{\mathcal{S}}(v,T)$ can be viewed as particular "conflict functions."

Relationships among the models

The following theorem shows that the memory models as presented are progressively weaker.

Theorem 6.1 $ST \subseteq RFT \subseteq PRFT$.

PROOF. Follows directly from Definitions 13, 15, and 17. □

For computations with only closed and committed transactions, prefix-race freedom and serializability are equivalent, as we shall see in Section 6.5. When open and aborted transactions are considered, all three models are distinct.

6.5 Distinctness of the Models

In this section, we study the memory models of serializability, prefix-race freedom, and race freedom. Specifically, we show that for computations containing only committed and closed transactions, all three models are equivalent. We also demonstrate that when aborted and/or open transactions are allowed, all three models are distinct.

Since these models are distinct under certain conditions and not distinct under others, we define certain special sets of traces. Definition 3 states that a memory model Δ is a subset of $\mathcal{U}$, the

universe of all possible traces. Sometimes, we wish to restrict our attention to computations with only closed and/or committed transactions. Thus, we define the following subsets of $\mathcal{U}$:

$$\begin{aligned}
\mathcal{U}_0 &= \{(C,\Phi) \in \mathcal{U} : \texttt{xactions}(C) = \emptyset\} \ , \\
\mathcal{U}_{\text{clo}} &= \{(C,\Phi) \in \mathcal{U} : \texttt{open}(C) = \emptyset\} \ , \\
\mathcal{U}_{\text{com}} &= \{(C,\Phi) \in \mathcal{U} : \texttt{aborted}(()C) = \emptyset\} \ , \\
\mathcal{U}_{\text{c\&c}} &= \mathcal{U}_{\text{clo}} \cap \mathcal{U}_{\text{com}} \ .
\end{aligned}$$

In other words, $\mathcal{U}_0$ contains traces (whose computations) include no transactions, $\mathcal{U}_{\text{clo}}$ contains traces that include only closed transactions, $\mathcal{U}_{\text{com}}$ contains traces that include only committed transactions, and $\mathcal{U}_{\text{c\&c}}$ contains traces that include only committed and closed transactions.

Dependency graphs

Before addressing the distinctness of the memory models directly, 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 18 *The set of* ***dependency*** *edges of a trace* $(C,\Phi) \in \mathcal{U}_{\text{com}}$ *is* $\Psi_d(C,\Phi) = \{(u,v) \in V(C) \times V(C) :$ *and the set of* ***antidependency*** *edges is* $\Psi_a(C,\Phi) = \{(u,v) \in V(C) \times V(C) : (\Phi(u) = \Phi(v)) \wedge W(v,\ell)\}$. *The* ***dependency graph*** *of* (C,Φ) *is the graph* $\mathcal{DG}(C,\Phi) = (V,E)$, *where* $V = V(C)$ *and* $E = E(C) \cup \Psi_d(C,\Phi) \cup \Psi_a(C,\Phi)$.

The sets Ψ_d and Ψ_a capture the usual notions of dependency and antidependency edges from the study of compilers [KKP+81]. 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, presented without proof, shows that in the universe of all traces with only committed transactions, a trace (C,Φ) is sequentially consistent if and only if the dependency graph $\mathcal{DG}(C,\Phi)$ is acyclic.[4]

Lemma 6.2 *Suppose that* $(C,\Phi) \in \mathcal{U}_{\text{com}}$. *Then, we have* $(C,\Phi) \in SC$ *if and only if the dependency graph* $\mathcal{DG}(C,\Phi)$ *is acyclic.* □

Figure 6.3 shows the dependency graphs for the example traces from Figure 6.2. Whereas the trace (C,Φ_1) is sequentially consistent, the trace (C,Φ_2) is not. Equivalently by Lemma 6.2, the dependency graph $\mathcal{DG}(C,\Phi_1)$ is acyclic, but the graph $\mathcal{DG}(C,\Phi_2)$ 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 transactions.

Theorem 6.3 $ST \cap \mathcal{U}_{\text{c\&c}} = RFT \cap \mathcal{U}_{\text{c\&c}} = PRFT \cap \mathcal{U}_{\text{c\&c}}$.

[4]One must extend the definition of an antidependency edge to prove an analogous result when the computation C has aborted transactions. Lemma 6.2 does not hold without the assumption that every write to a location also performs a read.

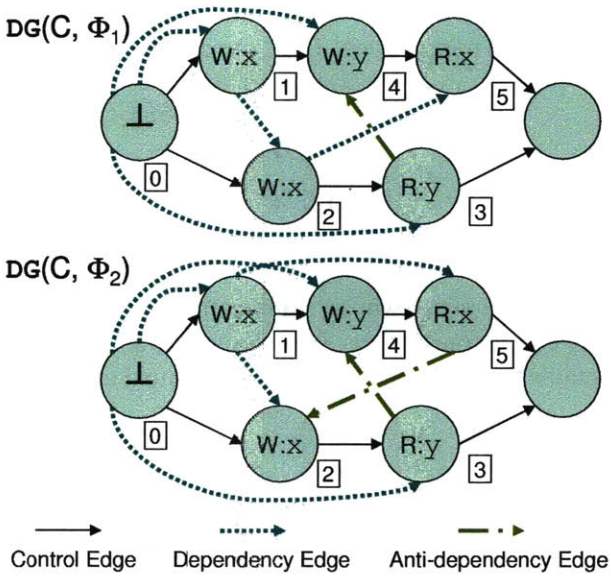

Figure 6.3: Dependency graphs $\mathcal{DG}(C, \Phi_1)$ and $\mathcal{DG}(C, \Phi_2)$ for the traces from Figure 6.2. Since $(C, \Phi_1) \in SC$, the graph $\mathcal{DG}(C, \Phi_1)$ is acyclic, but since $(G, \Phi_2) \notin SC$, the graph $DG(C, \Phi_2)$ contains a cycle, namely $\langle 2, 3, 4, 5, 2 \rangle$.

PROOF. Since Theorem 6.1 shows that $ST \subseteq RFT \subseteq PRFT$, it suffices to prove that $PRFT \cap \mathcal{U}_{\text{c\&c}} \subseteq ST \cap \mathcal{U}_{\text{c\&c}}$.

We start by defining some terminology. For $u, v \in V(C)$, define the ***alternation count*** of u and v as

$$A(u, v) = |\mathtt{h}(u)| + |\mathtt{h}(v)| - 2\,|\mathtt{h}(\mathtt{LCA}(u, v))| \ .$$

(The holders function h() was defined in Section 6.2.) Thus, $A(u, v)$ counts the number of transactions $T \in \mathtt{xactions}(C)$ that contain either u or v, but not both. For any topological sort $\mathcal{S}$ of $G(C)$, define the ***alternation count*** of $\mathcal{S}$, denoted $alt(\mathcal{S})$, as the sum of all $A(u, v)$ 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 $(C, \Phi) \in \mathcal{U}_{\text{c\&c}}$, we have $(C, \Phi) \in PRFT$ implies $(C, \Phi) \in SC$. Suppose that a trace $(C, \Phi) \in \mathcal{U}_{\text{c\&c}}$ exists that is prefix-race free but not serializable. Consider any prefix-race-free topological sort $\mathcal{S} \in \mathtt{topo}(\mathcal{DG}(C, \Phi))$ that has a minimum alternation count $alt(\mathcal{S})$ over all sorts in $\mathtt{topo}(\mathcal{DG}(C, \Phi))$. By Lemma 6.2, $\mathcal{S}$ satisfies the condition $\Phi = \mathcal{X}_{\mathcal{S}}$ (the first condition for all three transactional models).

Since $(C, \Phi) \notin ST$, some transaction T exists that is not contiguous in $\mathcal{S}$ (and therefore violates the second condition in Definition 13). Let T be such a transaction, and let v_1 be the first vertex such that $v_1 \notin V(T)$ and $\mathtt{begin}(T) <_{\mathcal{S}} v <_{\mathcal{S}} \mathtt{end}(T)$. Choose vertices $t <_{\mathcal{S}} u_1 \leq_{\mathcal{S}} u_2 <_{\mathcal{S}} v_1 \leq_{\mathcal{S}} v_2 <_{\mathcal{S}} w_1 <_{\mathcal{S}} w_2$, such that $u_1 = \mathtt{begin}(T)$ as shown in Figure 6.4(a). Define the sets A_1, A_2, and

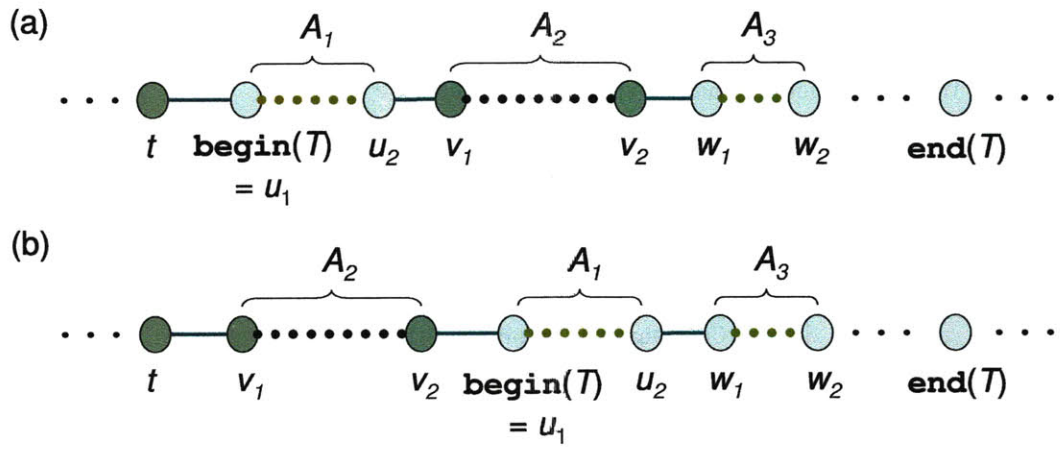

Figure 6.4: Two topological sorts of a computation graph $G(C)$ for a hypothetical trace (C, Φ) which is prefix-race free, but not serializable. Transaction T is not contiguous in the topological sort $\mathcal{S}$ in (a). One can convert $\mathcal{S}$ into the topological sort $\mathcal{S}'$ in (b). Doing so reduces the alternation count.

A_3 as follows:

$$\begin{aligned} A_1 &= \{x \in V(T) : u_1 \leq_{\mathcal{S}} x \leq_{\mathcal{S}} u_2\}, \\ A_2 &= \{x \in V(C) - V(T) : v_1 \leq_{\mathcal{S}} x \leq_{\mathcal{S}} v_2\}, \text{ and} \\ A_3 &= \{x \in V(T) : w_1 \leq_{\mathcal{S}} x \leq_{\mathcal{S}} w_2\} . \end{aligned}$$

Define two sets $A_1 = \{x \in V(T) : u_1 \leq_{\mathcal{S}} x \leq_{\mathcal{S}} u_2\}$ and $A_3 = \{x \in V(T) : w_1 \leq_{\mathcal{S}} x \leq_{\mathcal{S}} w_2\}$ whose vertices all belong to $V(T)$. Define $A_2 = \{x \in V(C) - V(T) : v_1 \leq_{\mathcal{S}} x \leq_{\mathcal{S}} v_2\}$ as the set interleaved between the contiguous fragments of T.

From $\mathcal{S}$, we construct the new order $\mathcal{S}'$ shown in Figure 6.4(b) in which the intervals A_1 and A_2 are interchanged. We shall show that (1) $\mathcal{S}' \in \texttt{topo}(\mathcal{DG}(C, \Phi))$ (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) $alt(\mathcal{S}') < alt(\mathcal{S})$, thereby obtaining the contradiction that $\mathcal{S}$ is not a prefix-race-free topological sort with minimum alternation count.

To prove these three facts, we shall use a "nonconflicting" property: no pair of vertices $y \in A_1$ and $z \in A_2$ exist such that y and z access the same memory location and one of them is a write. Otherwise we have $\text{PRACE}_{\mathcal{S}}(z, T)$ by definition because $y \in \texttt{cContent}(T)$, $z \notin V(T)$, and $\texttt{begin}(T) <_{\mathcal{S}} y <_{\mathcal{S}} z <_{\mathcal{S}} \texttt{end}(T)$. Thus, A_1 and A_2 do not perform "conflicting" accesses to memory.

To establish (1), that $\mathcal{S}' \in \texttt{topo}(\mathcal{DG}(C, \Phi))$, we show that for any $y \in A_1$ and $z \in A_2$, no edge (y, z) belongs to the graph $\mathcal{DG}(C, \Phi)$. If we have $(y, z) \in \Psi_d(C, \Phi) \cup \Psi_a(C, \Phi)$, then y and z access the same memory location and one of those accesses is a write, contradicting the nonconflicting property above. Alternatively, if we have $(y, z) \in E(C)$, then $\texttt{LCA}(y, z)$ must be an S-node with y to the left of z. Since $z \notin V(T)$, we have $\texttt{LCA}(T, z)$ $(= \texttt{LCA}(y, z))$ is an S-node, and thus we have $\texttt{end}(T) \prec z$. Thus, $\mathcal{S}$ was not a valid sort of $\mathcal{DG}(C, \Phi)$, and $(y, z) \notin E(C)$.

To establish (2), that $\mathcal{S}'$ is prefix-race free, we show that swapping A_1 and A_2 cannot introduce any prefix races that weren't already there in $\mathcal{S}$. Suppose that there is a prefix-race in $\mathcal{S}'$. Then,

there must exist a $v \in V(C)$ and a transaction $T_1 \in \texttt{xactions}(C)$ satisfying all three conditions of Definition 16 for $\mathcal{S}'$. Let $w \in \texttt{cContent}(T_1)$ be the candidate vertex that satisfies the three conditions. In particular, the third condition gives us $\texttt{begin}(T_1) <_{\mathcal{S}'} w <_{\mathcal{S}'} v <_{\mathcal{S}'} \texttt{end}(T_1)$. We consider two cases, each of which leads to a contradiction.

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 $w \in A_2$. But, then they conflict by the second condition of Definition 16, which cannot occur because of the nonconflicting property above.

In the second case, suppose that $w <_{\mathcal{S}} v$. Since there is no prefix-race in $\mathcal{S}$, the only situation in which this can happen is when v falls entirely outside transaction T_1 in $\mathcal{S}$, which is to say that $\texttt{begin}(T_1) <_{\mathcal{S}} w <_{\mathcal{S}} \texttt{end}(T_1) <_{\mathcal{S}} v$. Since $\texttt{end}(T_1)$ and v swapped, we must have $\texttt{end}(T_1) \in A_1$ and $v \in A_2$. Since $A_1 \subseteq \texttt{cContent}(T)$, it follows that $\texttt{end}(T_1) \in \texttt{cContent}(T)$, and thus T_1 must be nested within T. Consequently, we have $w \in A_1$, which cannot occur because of the nonconflicting property.

To establish (3), 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

$$\begin{aligned}
\delta &= A(t,u_1) + A(u_2,v_1) + A(v_2,w_1) \\
&\quad -A(t,v_1) - A(v_2,u_1) - A(u_2,w_1) \\
&= 2\,(|\texttt{h}(\texttt{LCA}(t,v_1))| + |\texttt{h}(\texttt{LCA}(v_2,u_1))| \\
&\quad +|\texttt{h}(\texttt{LCA}(u_2,w_1))| - |\texttt{h}(\texttt{LCA}(t,u_1))| \\
&\quad -|\texttt{h}(\texttt{LCA}(u_2,v_1))| - |\texttt{h}(\texttt{LCA}(v_2,w_1))|)\ .
\end{aligned}$$

By construction, we know that $\{u_1, u_2, w_1, w_2\} \subseteq V(T)$, whereas none of t, v_1, and v_2 have T as an ancestor. For any $y \in V(T)$ and $z \notin V(T)$, we have $\texttt{LCA}(y,z) = \texttt{LCA}(T,z)$, which yields

$$\begin{aligned}
\delta &= 2\,(|\texttt{h}(\texttt{LCA}(t,v_1))| + |\texttt{h}(\texttt{LCA}(u_2,w_1))| \\
&\quad - |\texttt{h}(\texttt{LCA}(t,T))| - |\texttt{h}(\texttt{LCA}(T,v_1))|)\ .
\end{aligned}$$

Since $\texttt{LCA}(u_2,w_1) \in \texttt{desc}(T)$, we know $\texttt{h}(\texttt{LCA}(u_2,w_1)) \supseteq \texttt{h}(T)$ and $|\texttt{h}(\texttt{LCA}(u_2,w_1))| \geq |\texttt{h}(T)|$. Since $t, v_1 \notin V(T)$, we have $\texttt{h}(\texttt{LCA}(T,t)) \subset \texttt{h}(T)$ and $\texttt{h}(\texttt{LCA}(T,v_1)) \subset \texttt{h}(T)$.[5] Thus,

$$|\texttt{h}(\texttt{LCA}(u_2,w_1))| > \max\{|\texttt{h}(\texttt{LCA}(T,t))|, |\texttt{h}(\texttt{LCA}(T,v_1))|\}\ ,$$

and a similar algebra yields

$$|\texttt{h}(\texttt{LCA}(t,v_1))| \geq \min\{|\texttt{h}(\texttt{LCA}(T,t))|, |\texttt{h}(\texttt{LCA}(T,v_1))|\}\ .$$

Consequently, we conclude that $\delta = alt(\mathcal{S}) - alt(\mathcal{S}') > 0$. □

Aborted transactions

We now consider computations with aborted transactions. We are unaware of any prior work on transactional semantics that explicitly models aborted transactions. The reason is simple: when

[5] In this case, we have a proper subset because $\texttt{LCA}(T,t), \texttt{LCA}(T,v_1) \in \texttt{pAnces}(T)$ and we exclude T.

computations have only closed transactions, aborted transactions do not affect a program's output. Since TM systems do not allow committed transactions to observe data directly from aborted transactions, in most cases, vertices from aborted transactions are free to observe arbitrary values.[6]

In a system with open nesting, however, we must include aborted transactions in the memory model if we wish to understand what happens when an open transaction commits but its parent aborts. We contend that a reasonable transactional consistency model for open transactions must not only model aborted transactions, but it should also guarantee that an aborted transaction T is consistent up to the point it aborts. Otherwise, any open subtransactions within T may obtain inconsistent values and still commit.

The next theorem shows that when aborted transactions are modeled, the three transactional memory models are distinct.

Theorem 6.4 $ST \cap \mathcal{U}_{\text{clo}} \not\subseteq RFT \cap \mathcal{U}_{\text{clo}} \not\subseteq PRFT \cap \mathcal{U}_{\text{clo}}$.

PROOF. Since Theorem 6.1 shows that $ST \subseteq RFT \subseteq PRFT$, we need only show that $ST \cap \mathcal{U}_{\text{clo}} \neq RFT \cap \mathcal{U}_{\text{clo}}$ and that $RFT \cap \mathcal{U}_{\text{clo}} \neq PRFT \cap \mathcal{U}_{\text{clo}}$.

We first exhibit a computation that is race free but not serializable. Consider the computation dag G shown in Figure 6.5. Let (C_1, Φ_1) be the trace that generates G, where transactions T_2 and T_3 abort but transaction T_4 commits. We shall show that $(C_1, \Phi_1) \in RFT$, but $(C_1, \Phi_1) \notin ST$.

If transaction T_4 commits, then for any topological sort $\mathcal{S}$ satisfying $\mathcal{X}_{\mathcal{S}} = \Phi$, we must have $0 <_{\mathcal{S}} 3 <_{\mathcal{S}} 6 <_{\mathcal{S}} 9$. Thus, T_1 cannot be contiguous within $\mathcal{S}$, implying that $(C_1, \Phi_1) \notin ST$.

We can show that (C_1, Φ_1) is race free, however. Let $\mathcal{S}$ be $\langle 0, 1, \ldots, 12 \rangle$. One can verify that Φ_1 is indeed the transactional last-writer function according to $\mathcal{S}$ (since T_4 commits, $\neg(6H9)$, and thus $\Phi_1(9) = \mathcal{X}_{\mathcal{S}}(9)$). The only transactions that might violate the second condition of Definition 15 are transactions that do not appear contiguous in $\mathcal{S}$, in this case, only T_1. The only candidate vertex v for $\text{RACE}_{\mathcal{S}}(v, T_1)$ is $v = 6$. Since T_2 is an aborted subtransaction of T_1, however, neither 3 or 9 belong to $\texttt{cContent}(T_1)$. Thus, picking $\mathcal{S} = \langle 0, 1, \ldots, 12 \rangle$ ensures that T_1 causes no races.

We next exhibit a computation that is prefix-race free but not race free. Consider (C_2, Φ_2) as the trace generating the same computation dag G from Figure 6.5, but this time with T_2 aborted and T_3 and T_4 committed. We shall show that $(C_2, \Phi_2) \notin RFT$, but that $(C_2, \Phi_2) \in PRFT$.

To show that (C_2, Φ_2) is not race free, observe that in any topological sort $\mathcal{S} \in \texttt{topo}(G)$ for which $\Phi = \mathcal{X}_{\mathcal{S}}$, we must have $\text{RACE}_{\mathcal{S}}(6, T_1)$, since $\texttt{begin}(T_1) <_{\mathcal{S}} 6 <_{\mathcal{S}} \texttt{end}(T_1)$, vertices 6 and 9 access the same memory location x, and vertex 6 is a write, and $\neg(6H9)$. The order $\mathcal{S} = \langle 0, 1, \ldots, 12 \rangle$ is prefix-race free, however, since $9 \not<_{\mathcal{S}} 6$. The only transactions that might violate the second condition of prefix-race freedom are those that do not appear contiguous in $\mathcal{S}$, in this case, only T_1. When we look at the vertex $v = 6$ that falls between $\texttt{begin}(T_1)$ and $\texttt{end}(T_1)$, we only look at the prefix of T_1 before v (vertices 1 through 4) for a prefix-race conflict, and there is none.

The proof holds whether T_1 commits or aborts. □

Open transactions

We now study computations with open transactions but where all transactions commit. In this context, the three models ST, RFT, and $PRFT$ are distinct.

[6]This intuition is not strictly true in a model that does not analyze an execution *a posteriori*, since control flow can be affected by inconsistent data and prevent a program from terminating.

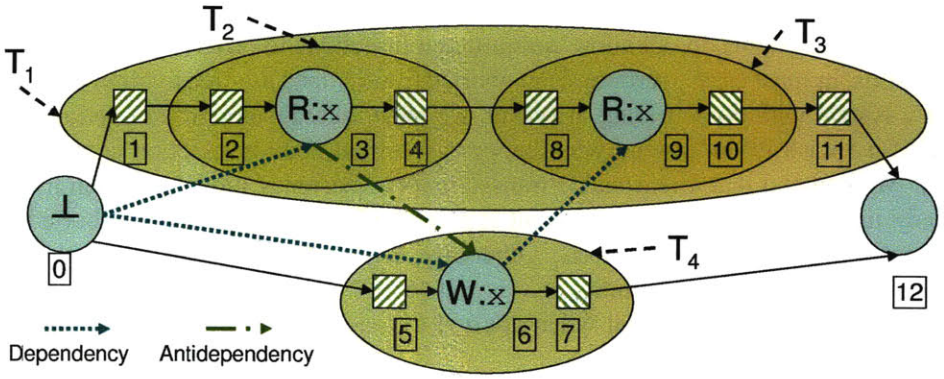

Figure 6.5: An example distinguishing the memory models. The transactions T_2 and T_3 are closed-nested inside of T_1. If transaction T_4 commits, then this computation is not serializable, because T_4 must interleave inside of T_1. If both transactions T_2 and T_3 abort, then the execution is race free. If T_2 aborts and T_3 commits, then this execution is not race free, but it is prefix-race free.

Theorem 6.5 $ST \cap \mathcal{U}_{\text{com}} \not\subset RFT \cap \mathcal{U}_{\text{com}} \not\subset PRFT \cap \mathcal{U}_{\text{com}}$.

PROOF. Since Theorem 6.1 shows that $ST \subseteq RFT \subseteq PRFT$, we need only show that $ST \cap \mathcal{U}_{\text{com}} \neq RFT \cap \mathcal{U}_{\text{com}}$ and that $RFT \cap \mathcal{U}_{\text{com}} \neq PRFT \cap \mathcal{U}_{\text{com}}$. The trace in Figure 6.6 shows a $(C_1, \Phi_1) \notin ST$, but $(C_1, \Phi_1) \in RFT$. Figure 6.7 shows $(C_2, \Phi_2) \notin RFT$, but $(C_2, \Phi_2) \in PRFT$.

Consider a trace $(C_1, \Phi_1) \in \mathcal{U}_{\text{com}}$ that generates the computation dag shown in Figure 6.6. This trace describes Schedule 3 from the code in Figure 7.1. We can show that $(C_1, \Phi_1) \notin ST$, but that $(C_1, \Phi_1) \in RFT$.

Suppose for the purpose of contradiction that $(C_1, \Phi_1) \in ST$. Any topological sort $\mathcal{S}$ that satisfies the first condition of Definition 13 must be consistent with all dependency edges. But then, we must have $3 <_{\mathcal{S}} 7 <_{\mathcal{S}} 10 <_{\mathcal{S}} 11 <_{\mathcal{S}} 15 <_{\mathcal{S}} 19$. Thus, if we consider $V(A)$ and pick $v = 10$ (or 11), we violate the second condition of Definition 13. Said differently, because transaction J_1 reads from I_1, and transaction I_2 reads from J_1, we cannot have $V(A)$ appear contiguous in any topological sort $\mathcal{S}$.

By picking $\mathcal{S} = \langle 1, 2, \ldots, 26 \rangle$, the trace (C_1, Φ_1) is race free. We can verify this fact by checking all transactions T satisfying the second condition of Definition 15. First, the nested transactions I_1, I_2, J_1, and J_2 all appear contiguous in $\mathcal{S}$, and thus there is no such v that appears between the `begin` and `end` vertices to satisfy the third condition of Definition 14. For transaction A, all vertices that appear between $\texttt{begin}(A)$ and $\texttt{end}(A)$ that do not belong to $V(A)$ belong to $V(B)$. But for all $v \in V(B)$ and $w \in \texttt{cContent}(A)$, vertices v and w access different memory locations. (Recall that I_1 and I_2 are excluded from $\texttt{cContent}(A)$ because they are open transactions.) Thus, we can not satisfy the second condition of Definition 14 for A. A similar argument rules out races with B.

To show that $RFT \cap \mathcal{U}_{\text{com}} \neq PRFT \cap \mathcal{U}_{\text{com}}$, consider the trace $(C_2, \Phi_2) \in \mathcal{U}_{\text{com}}$ shown in Figure 6.7, which represents an execution of the program from Figure 7.2. If all transactions commit, then $(C_2, \Phi_2) \notin RFT$. Any topological sort $\mathcal{S}$ satisfying the first condition of race freedom must have $13 <_{\mathcal{S}} 15$, because 15 reads the value of b written by 13. Looking at A, we can pick $v = 13$ and $w = 15$. We see that $\texttt{begin}(A) \prec v \prec \texttt{end}(A)$, $w \in \texttt{cContent}(A)$, and the first

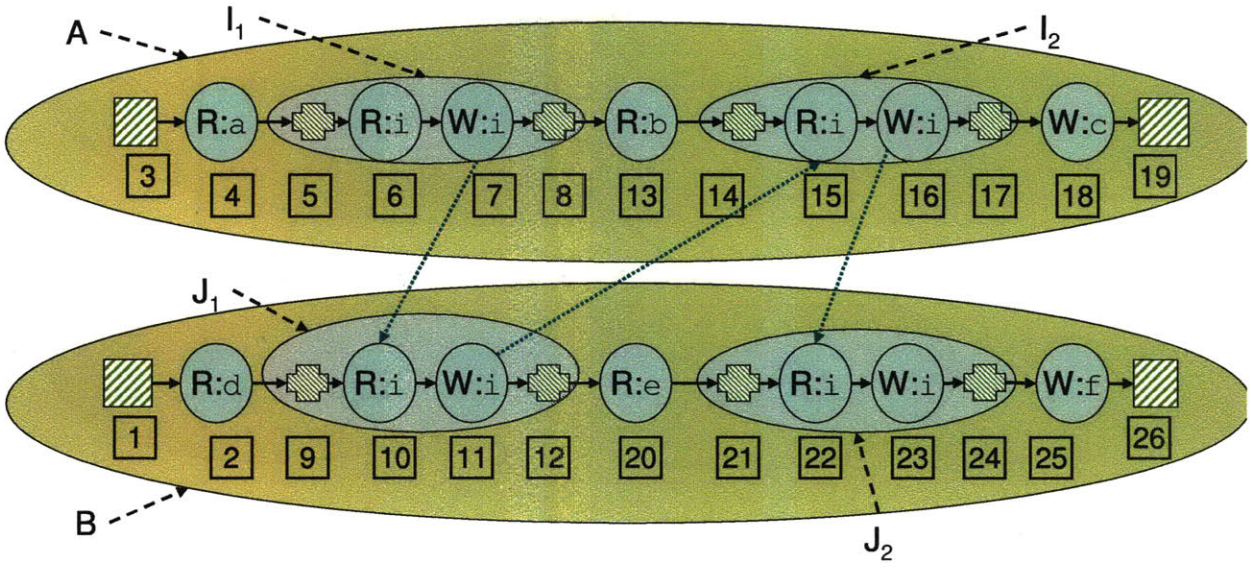

Figure 6.6: When all transactions commit, this computation dag $G(C_1)$ with observer edges Φ_1 is not serializable, but is race free. This trace represents Schedule 3 from the program in Figure 7.1.

and second conditions of Definition 14 are satisfied. Thus, we satisfy $\text{RACE}_{\mathcal{S}}(13, A)$, and hence $(C_2, \Phi_2) \notin RFT$.

We can show that $(C_2, \Phi_2) \in PRFT$, however. Consider $\mathcal{S} = \langle 1, 2, \ldots, 17 \rangle$. Since transactions B and C appear contiguous in $\mathcal{S}$, we only need to check A for prefix-races. The only vertices $v \notin V(A)$ that come between $\texttt{begin}(A)$ and $\texttt{end}(A)$ are $v = 12$ and $v = 13$, neither of which conflicts with w in the prefix of A (vertices 1 through 6). □

Trade-offs among the models

The three transactional memory models of serializability, race freedom, and prefix-race freedom exhibit different behaviors in TM systems that have open transactions.

With serializability, for any trace $(C, \Phi) \in ST$, we can "change" the trace to convert any open transaction T' nested inside a committed transaction T from open to closed while still keeping the same Φ, and still be serializable. Thus, in some sense, with serializability, open nesting only differs from closed nesting if an open transaction commits, but its parent aborts.

Race-freedom appears to be more difficult to implement than either serializability or prefix race-freedom. For example, consider the example from Figures 7.2 and 6.7. After an transaction I_1 (open-nested in A) commits, any number of other transactions (B and C) can read values written by that open transaction and commit their changes, all before the original outer transaction A completes. To support race freedom, it seems we may need to maintain the footprints of B and C even after they have committed to detect a future conflict with A.

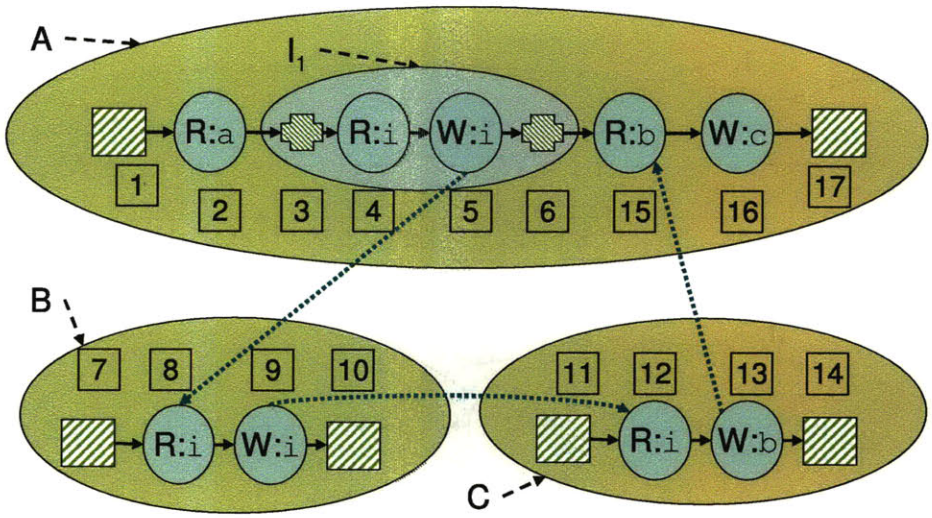

Figure 6.7: When all transactions commit, this computation dag $G(C_2)$ with observer edges Φ_2 is prefix-race free, but not race free, because a race exists between vertices 13 and 15.

6.6 Related Work

When transactional memory research began, researchers assumed that serializability would be the obvious memory model for TM. Recently, since we completed the work described in this chapter, there has been work on formal models for transactional memory. Moore and Grossman [MG08] focus on program semantics of transactional memory and how it might interact with other language features. They introduce a framework to analyze the various features of software transactional memory formally. Guerraoui and Kapalka [GK08] introduce opacity as the correctness criteria for transactional memory.

Most of the debate on the semantics of transactional memory has recently concentrated on ***strong*** versus ***weak isolation***. Informally, strong isolation guarantees that transactional accesses to memory are isolated from nontransactional accesses. In our work, we have concentrated almost exclusively on strong isolation memory models. However, strong isolation may be difficult to provide efficiently in software transactional memory since it prohibits many hardware and compiler optimizations. Therefore, there has been much interest in understanding reasonable memory models that provide weak isolation [MBS+08, SATG+09, DS09, SDMS08].

However, most of this work does not model precise semantics of open nested transactions. Formal models for systems with nested transactions appear as early as the work by Beeri, Bernstein, and Goodman [BBG89]. Most of this work is related to database transactions, however. Papers providing operational semantics for open transactions include [MH05, MCC+06, Lib06]. Although operational semantics of a TM can provide an abstract basis for implementation, inferring emergent properties of the system from these semantics can be quite difficult. The work on opacity as a correctness criteria does consider open nesting. Their work treats an open-nested transaction as single abstract operation, however, and does not concern itself with specific memory operations inside the open-nested transaction. Therefore, the semantics defined by their model is at a sligtly

higher level than that presented in this thesis.

Chapter 7

Semantics of Open-Nested Transactions

This chapter explores open nesting in more detail. Chapter 6, we saw the framework for describing TM semantics. In this chapter, we use that framework to define the precise semantics of open nested transactions. We see examples of why open nested transactions are not serializable, and the strange behaviors that result from this fact. We prove that in fact, open nested transactions exhibit prefix-race freedom.

The chapter is organized as follows: Section 7.1 describes some behaviors, both desirable and undesirable, allowed by open nesting, Section 7.2 describes the operational semantics of open nesting using our computational tree framework, Section 7.3 presents a proof that this operational semantics guarantees prefix-race freedon, and finally, Section 7.4 contains a discussion of undesirablity of an unconstrained use of open-nested transactions.

7.1 Subtleties of Open Nesting

This section motivates the need for a precise description of the memory semantics of open-nested transactions using three examples to illustrate some subtleties with open nesting. The first example shows that some desirable schedules allowed by open nesting are not serializable. The second example shows that the loss of serializability for open nesting sanctions arguably bizarre program behaviors. The third example shows that open nesting compromises composability.

Figure 7.1 describes a program with nested transactions where the use of open nesting admits a desirable schedule which is not serializable. Moreover, a system with only flat or closed nesting prohibits the schedule. In Figure 7.1, transaction A reads from global variable a, adds a key-value pair based on a to a global table, reads from b and adds a corresponding pair to the table, and then stores the sum $a + b$ into c. Transaction B performs analogous operations on d, e, and f. The table data structure is implemented as a simple direct-access table [CLRS01, Section 11.1] with a global `size` field to count the number of elements in the table.

If the nested transactions (the I's and J's) are all flat-nested or closed-nested, then TM guarantees that the transactions are serializable: the program appears to executes as though either A happened before B (Schedule 1) or B happened before A. The system might actually perform the operations in a different, interleaved order (for example, Schedule 2), but this schedule is equivalent to one of the two valid serial schedules (in this case, Schedule 1). Schedule 3 is not serializable,

This work was done in collaboration with Charles E. Leiserson and Jim Sukha [ALS06].

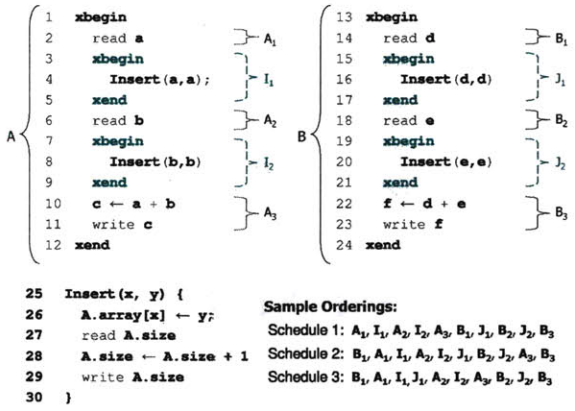

Figure 7.1: Two concurrent transactions that do not share any memory locations except in their nested transactions. Divide transaction A into abstract operations A_1, I_1, A_2, I_2, A_3, and divide B into B_1, J_1, B_2, J_2, B_3. The I's and J's represent inserts to an abstract table data structure. Schedule 1 is a serial order, Schedule 2 is an interleaved order equivalent to Schedule 1, and Schedule 3 is an interleaved order which is not serializable.

however, because J_1 (and thus B) observes the intermediate value of `A.size` written by I_1 (and thus written by A). Consequently, Schedule 3 is prohibited with flat or closed nesting.

To improve concurrency, a programmer may wish to allow certain schedules that are not serializable, but which nevertheless are consistent from the programmer's point of view. A system that can admit nonserializable schedules imposes fewer restrictions on transactions, possibly allowing transactions to commit when they would have otherwise aborted. For example, the programmer may wish to admit Schedule3, even though the I's and J's happen to access the same `size` field. Conceptually, the programmer may not care in which order the table inserts occur. For example, if I_1, I_2, J_1, and J_2 are open transactions, then Schedule 3 is a valid execution.

Once a TM system with open nesting admits some desirable nonserializable schedules, however, 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. Figure 7.2 shows a program execution allowed by the open-nesting implementations of [MH05, MCC+06]. In this example, it is possible for all transactions A, I_1, B, and C to commit, even though A does not appear to execute atomically. Transaction A reads inconsistent data, since C writes to b between A's reads of a and b. Thus, the "snapshot" of the world seen by A when it begins is different from its snapshot part way through its computation.

Our final example illustrates how open nesting can admit subtle program behaviors that affect the composability of transactions. Consider the program in Figure 7.3 which describes an implementation of a simple table library that (arguably) contains an subtle flaw. The program includes a `Contains(x)` method to complement the `Insert(x,y)` method used in Figure 7.1. 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 transaction.

Using TM with open nesting, in any sequence of `Contains` or `Insert` operations, each individual operation still appears atomic. Thus, in transaction A in Figure 7.3, we might expect

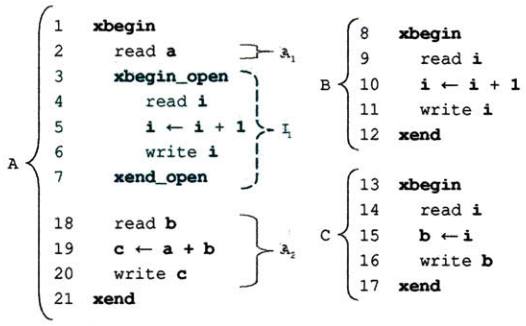

Figure 7.2: A program execution permitted by open nesting. Transaction A does not appear to execute atomically, because it can read an "inconsistent" value for b if B and C interleave between the execution of A_1 and A_2.

```
A                                   B
1 xbegin                            3 xbegin
2   if (!Contains(5))               4   if (!Contains(5))
7     Insert(5,15)                  5     Insert(5,10)
8 xend                              6 xend

Contains(x)                         Insert(x,y)
9  xbegin                           19 xbegin
10  found ← true                    20  A.array[x] ← y
11  xbegin_open                     21  read A.size
12    read A.size                   22  A.size ←A.size + 1
13    if (A.size==0) empty←true     23  write A.size
14  xend_open                       24 xend
15  if (!empty)
16    found ←A.array[x]
17  return ((!empty) && found)
18 xend
```

Figure 7.3: Flawed implementation of a table data structure with two methods, `Contains(x)` and `Insert(x,y)`. Although each method individually appears atomic, transactions A and B, which call those methods, may not appear atomic. In particular, the ordering $< 1, 2, 3, 4, 5, 6, 7, 8 >$ is allowed.

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 T and still have T appear to be atomic. Indeed, the open-nesting implementation described in [MCC+06] allows the entire transaction B to execute between Lines 2 and 7 of transaction A. Thus, this code shows that composability of transactions is not preserved. When using open nesting, simply ensuring the atomicity of individual transactions is not sufficient to guarantee composability.

Admittedly, the examples in Figures 7.2 and 7.3 are somewhat contrived. In particular, unlike in Figure 7.1, transactions in Figures 7.2 and 7.3 cannot be partitioned into clear abstraction levels, with each level accessing disjoint memory locations, as Moss suggests may be necessary [Mos06]. These examples suggest, however, that for open nesting, the distinction between the abstract program model and the low-level memory model is much more significant than for closed or flat nesting. Thus, these examples motivate the need to understand memory models for open nesting so that at the very least we can understand what properties should be enforced by higher-level mechanisms.

7.2 The Operational Model

This section presents an abstract operational model for open nesting, called the ON model. This model is a generalization of most open nesting implementations, specifically the Stanford model [MCC+06]. We prove that the ON model implements at least prefix race-freedom but is strictly weaker than race freedom.

We begin our description of the ON model by defining some notation. For any set $S \subseteq \texttt{nodes}(C)$ of tree nodes, let $\texttt{lowest}(S)$ be the node $X \in S$ such that $S \subseteq \texttt{ances}(X)$, if such a X exists. Otherwise, define $\texttt{lowest}(S) = \texttt{null}$. Thus, if all nodes in S all fall on one root-to-leaf path in C, then $\texttt{lowest}(S)$ is the lowest node on that path. Define $\texttt{highest}(()S)$ in a similar fashion. For any $T \in \texttt{xactions}(C)$, define $\texttt{xparent}[T] = \texttt{lowest}(\texttt{ances}(T) \cap \texttt{xactions}(C))$, that is, $\texttt{xparent}[T]$ is the transactional parent of T. For any $X \in \texttt{nodes}(C)$, let $\texttt{xAnces}(X) = \texttt{ances}(X) \cap \texttt{xactions}(C)$ be the set of transactional ancestors of X.

Abstractly, we shall view the ON model for open nesting as a nondeterministic state machine ON that constructs a sequence of traces. The initial trace contains a computation tree consisting of a single S-node $\texttt{root}(C) \in \texttt{spNodes}(C)$ with associated sets $\texttt{xactions}(C) = \{\texttt{root}(C)\}$ and $\texttt{open}(C) = \texttt{committed}(C) = \texttt{aborted}(C) = \emptyset$ and an empty observer function Φ. By assuming that $\texttt{root}(C) \in \texttt{xactions}(C)$, we simplify the description of the model by treating the entire computation C as a global closed transaction in which other transactions are nested. The computation also maintains an initially empty auxiliary set $\texttt{done}(C) \subseteq \texttt{nodes}(C)$ of nodes that have finished their execution. The computation tree C and all these associated sets only grow during the execution.

At any time during the computation, a subset $\texttt{ready}(()C)$ of S-nodes are designated as ***ready***, meaning that they can issue a program instruction, which include `read`, `write`, `fork`, `join`, `xbegin`, `xbegin_open`, and `xend`. The ON machine nondeterministically chooses a ready S-node to issue an instruction, and the machine processes the instruction which augments (C, Φ) by adding nodes to the tree and to its associated sets. Unlike other associated sets $\texttt{ready}(()C)$ may grow and shrink during execution.

We shall factor the description of the state machine ON by describing the creation of the computation tree C and the observer function Φ separately.

Creating the computation tree

How the computation tree C evolves depends on the instructions that are issued nondeterministically. Let X be the S-node that issues an instruction. The instructions are handled as follows:

- `read` from a location $\ell \in \mathcal{M}$: If the read causes a conflict (more about conflicts when we describe the creation of the observer function) with one or more transactions, abort[1] the deepest such transaction T by adding all transactions $T' \in \texttt{desc}(T) \cap \texttt{xactions}(T) - \texttt{done}(C)$ both to $\texttt{aborted}(C)$ and to $\texttt{done}(C)$. Keep checking for and aborting conflicting transactions T, deepest to shallowest, until no such conflicting transactions exist. Then,

[1]The ON machine uses a "pessimistic" concurrency control mechanism in that it immediately aborts a conflicting transaction T upon conflict. Moreover, it always aborts T rather than its own transaction. One could abort the transaction performing the `read`, but the model is simpler by always aborting T and not providing a nondeterministic choice.

create a new read node $v \in \texttt{memOps}(C)$ as the last child of the S-node X. Add v to $\texttt{done}(C)$.

- write to a location $\ell \in \mathcal{M}$: Similar to read.
- fork: Create a new P-node $Y \in \texttt{nodes}(C)$ as a child of X, and create two new S-nodes as children of Y. Add these two children to $\texttt{ready}(()C)$, and remove X from $\texttt{ready}(()C)$.
- join: Test whether X's sibling belongs to $\texttt{done}(C)$. If yes, then add X and then $\texttt{parent}[X]$ to $\texttt{done}(C)$. Remove X from $\texttt{ready}(()C)$, and add $\texttt{parent}[\texttt{parent}[X]]$ (the grandparent of X which is an S-node) to $\texttt{ready}(()C)$. If no, then remove X from $\texttt{ready}(()C)$, and add X to $\texttt{done}(C)$.
- xbegin: Create a new S-node $Y \in \texttt{nodes}(C)$ as the last child of X. Add Y to $\texttt{xactions}(C)$. Remove X from $\texttt{ready}(()C)$, and add Y to $\texttt{ready}(()C)$.
- xbegin_open: Similar to xbegin, but also add Y to $\texttt{open}(C)$.
- xend: Test whether $X \in \texttt{xactions}(C)$. If yes, remove X from $\texttt{ready}(()C)$, and add $\texttt{parent}[X]$ to $\texttt{ready}(()C)$. Add X to $\texttt{done}(C)$ and to $\texttt{committed}(C)$. If no, error.

The ON machine maintains several invariants. All transactions are S-nodes. Every P-node has an S-node as its parent and has exactly two S-nodes as children. If an S-node is ready, none of its ancestors are ready.

Creating the observer function

To create the observer function, the ON model maintains auxiliary state to keep track of how values are propagated among transactions and global memory. Specifically, every transaction $T \in \texttt{xactions}(C)$ maintains a ***readset*** $\texttt{R}(T)$ and a ***writeset*** $\texttt{W}(T)$. The readset $\texttt{R}(T)$ is a set of pairs (ℓ, v), where $\ell \in \mathcal{M}$ is a memory location and $v \in \texttt{memOps}(C)$ is the memory operation that read from ℓ, that is, we maintain the invariant $R(v, \ell)$ for all $(\ell, v) \in \bigcup_{T \in \texttt{xactions}(C)} \texttt{R}(T)$. The writeset $\texttt{W}(T)$ is similarly defined. We initialize $\texttt{R}(\texttt{root}(C)) = \texttt{W}(\texttt{root}(C)) = \{(\ell, \texttt{begin}(\texttt{root}(C))) : \ell \in \mathcal{M}\}$.

The ON model maintains two invariants concerning readsets and writesets. First, it maintains $\texttt{W}(T) \subseteq \texttt{R}(T)$ for every transaction $T \in \texttt{xactions}(C)$, that is, a write to a location also counts as a read to that location. Second, $\texttt{R}(T)$ and $\texttt{W}(T)$ each contain at most one pair (ℓ, v) for any location ℓ. Because of this second invariant, we employ the shorthand $\ell \in \texttt{R}(T)$ to mean that there exists a node u such that $(\ell, u) \in \texttt{R}(T)$, and similarly for $\texttt{W}(T)$. We also overload the union operator to accommodate this assumption: if we write $\texttt{R}(T) \leftarrow \texttt{R}(T) \cup \{(\ell, u)\}$, then if there exists $(\ell, u') \in \texttt{R}(T)$, we mean to replace it with (ℓ, u). Likewise, if u accesses a location ℓ, we employ the shorthand $u \in \texttt{R}(T)$ to mean that $(\ell, u) \in \texttt{R}(T)$, and similarly for $\texttt{W}(T)$.

The state machine ON handles events as follows, where X is the S-node that issues the instruction:

- read from location $\ell \in \mathcal{M}$: If there exists a $T \in \texttt{xactions}(C) - \texttt{done}(C) - \texttt{ances}(X)$ such that $\ell \in \texttt{W}(T)$, then a conflict occurs. Let v be the read operation added as the last child of X. Define $S_\ell = \{T \in \texttt{xactions}(C) \cap \texttt{ances}(v) : \ell \in \texttt{R}(T)\}$, let $T' = \texttt{lowest}(S_\ell)$, and let $(\ell, u) \in \texttt{R}(T')$. Add (ℓ, u) to $\texttt{R}(T)$, and set $\Phi(v) = u$.

- `write` to a location $\ell \in \mathcal{M}$: Similar to `read`, but to check for a conflict, test whether there exists a $T \in \texttt{xactions}(C) - \texttt{done}(C) - \texttt{ances}(X)$ such that $\ell \in \texttt{R}(T)$. Find u in the same way, and add (ℓ, u) both to $\texttt{R}(T)$ and to $\texttt{W}(T)$, and set $\Phi(v) = u$.
- `xbegin` and `xbegin_open`: Initialize $\texttt{R}(Y) = \emptyset$ and $\texttt{W}(Y) = \emptyset$.
- `xend`: If $X \in \texttt{closed}(C)$, then add $\texttt{R}(X)$ to $\texttt{R}(\texttt{xparent}[X])$ and add $\texttt{W}(X)$ to $\texttt{W}(\texttt{xparent}[X])$. If $X \in \texttt{open}(C)$, then let $Q = \texttt{xAnces}(T)$. For any $(\ell, u) \in \texttt{W}(T)$, let $\alpha_\ell = \{T' \in Q \mid \ell \in \texttt{R}(T')\}$. For all such $T' \in \alpha_\ell$, $\texttt{R}(T') \leftarrow \texttt{R}(T') \cup \{(\ell, u)\}$. Similarly, let $\beta_\ell = \{T' \in Q \mid \ell \in \texttt{W}(T')\}$. For all $T' \in \beta_\ell$, $\texttt{W}(T') \leftarrow \texttt{W}(T') \cup \{(\ell, u)\}$.
- `fork` or `join`: No action.

The Stanford model [MCC+06] is similar to the *ON* model, except that it only supports "linear" nesting (transactions can have no parallel transactions within them) and the choice of which transaction to abort is nondeterministic. Neither of these differences affects the theorems that deal with the *ON* model, assuming they implement their system with pessimistic concurrency control.

7.3 Prefix Race-Freedom of the Operational Model

We now prove that the *ON* model is prefix-race free with respect to the natural topological sort $\mathcal{S}$ of $G(()C)$ created by the nondeterministic operation of the *ON* machine. Specifically, as the *ON* model generates a trace (C, Φ), it creates tree nodes $\texttt{nodes}(C) = \texttt{spNodes}(C) \cup \texttt{memOps}(C)$ and eventually marks these nodes as "done" by placing them in $\texttt{done}(C)$. We can view this process as determining the topological sort $\mathcal{S}$ of $G(()C)$ as follows. When a node $X \in \texttt{nodes}()$ is created, the vertex $\texttt{begin}(X) \in V(C)$ is appended to $\mathcal{S}$. When a node is marked as done, the vertex $\texttt{end}(X) \in V(C)$ is appended to $\mathcal{S}$. If the node X is a memory operation, we have $\texttt{begin}(X) = \texttt{end}(X) = X$, and we view it as being appended only once. It is straightforward to verify that $\mathcal{S}$ is indeed a topological sort of $G(()C)$, and indeed of $\mathcal{DG}(C, \Phi)$.

We begin with a definition of time in the *ON* model. If $v \in V(C)$ is the tth element of $\mathcal{S}$, we say that v occurs at ***time*** t, and we write $t = \mathcal{S}(v)$. Thus, for all $u, v \in V(C)$, we have $u \leq_{\mathcal{S}} v$ if and only if $\mathcal{S}(u) \leq \mathcal{S}(v)$. We can view the evolution of (C, Φ) over time as a sequence $(C^{(t)}, \Phi^{(t)})$ for $t = 0, 1, \ldots$, where the operation that occurs at time t creates $(C^{(t)}, \Phi^{(t)})$ from $(C^{(t-1)}, \Phi^{(t-1)})$. For convenience, however, we shall omit time indices unless clarity demands it.

We define two time-sensitive sets. The set of ***active*** transactions at any given time is $\texttt{activeX}(C) = \texttt{xactions}(C) - \texttt{done}(C)$. The ***spine*** of a memory location $\ell \in \mathcal{M}$ at any given time is $\texttt{spine}(\ell) = \{T \in \texttt{activeX}(C) : \ell \in \texttt{W}(T)\}$.

We now state a structural lemma that describes invariants of the computation tree C as it evolves.

Lemma 7.1 *The ON machine maintains the following invariants:*

1. *If* $T \in \texttt{activeX}(C)$, *then we have* $\texttt{xAnces}(T) \subseteq \texttt{activeX}(C)$.
2. *If* $v \in \texttt{W}(T)$, *then* $v \in V(T)$.
3. *All transactions in* $\texttt{spine}(\ell)$ *are on the same root to leaf path in* C, *and hence the node* $\texttt{lowest}(\texttt{spine}(\ell))$ *exists.*

4. *If* $\ell \in \mathtt{R}(T)$, *where* $T \in \mathtt{activeX}(C)$, *then we have either* $\mathtt{spine}(\ell) \subseteq \mathtt{ances}(T)$ *or* $T \in \mathtt{ances}(\mathtt{lowest}(\mathtt{spine}(\ell)))$.

5. *If* $(\ell, u) \in \mathtt{R}(T)$ *for some* $T \in \mathtt{activeX}(C)$, *then* $(\ell, u) \in \mathtt{W}(T')$, *where* $T' = \mathtt{lowest}(\mathtt{xAnces}(T) \cap \mathtt{spine}(\ell))$.

6. *Let* $(\ell, u) \in \mathtt{W}(T_1)$ *and* $(\ell, v) \in \mathtt{W}(T_2)$, *where* $T_1, T_2 \in \mathtt{spine}(\ell)$. *If* $T_1 \in \mathtt{ances}(T_2)$, *then* $u \leq_S v$.

7. *Let* $(\ell, u) \in \mathtt{W}(T)$ *and let* $u <_S v$ *such that* $W(v, \ell)$. *Then, we have* $v \in \mathtt{desc}(T)$.

PROOF. By induction.

For the base case, we start with $\mathtt{xactions}(C) = \{\mathtt{root}(C)\}$, and $\mathtt{done}(C) = \emptyset$. Our initial readset $\mathtt{R}(\mathtt{root}(C))$ and writeset $\mathtt{W}(\mathtt{root}(C))$ contain all pairs $(\ell, \mathtt{begin}(\mathtt{root}(C)))$ for all $\ell \in \mathcal{M}$. Thus, $\mathtt{spine}(\ell) = \{\mathtt{root}(C)\}$ for all locations $\ell \in \mathcal{M}$, and all invariants are satisfied.

For the inductive step, assume all invariants are true at time $t - 1$.

The `fork`, `join`, `xbegin`, `xbegin_open` events do not involve any memory operations or readset/writeset manipulations. Thus, for these events, we only need to argue that Invariant 1 is preserved at time t.

To check the invariants after a memory operation to a location ℓ, we first label some transactions on $\mathtt{spine}(\ell)$ for time $t - 1$. Let $T_1, T_2, \ldots T_k$ be the k transactions along $\mathtt{spine}(\ell)$. By Invariant 3, for all i, $T_i \in \mathtt{ances}(T_{i+1})$ (all the T_i's are along the same root-to-leaf path), and $\ell \in \mathtt{W}(T_i)$. Let x_i be the memory operation such that $(\ell, x_i) \in T_i$. By Invariant 4 and Invariant 5, any transaction T' for which $\ell \in \mathtt{R}(T')$ satisfies $(\ell, x_j) \in \mathtt{R}(T')$ for some j, that is, T' has the value x_j from its closest transactional ancestor on T_j on the spine. By Invariant 6, we know $x_i <_S x_j$ for all $1 \leq i < j \leq k$, and by Invariant 7, we know for any $y \in V(C)$ that satisfies $W(y, \ell)$ and $x_i <_S y$, $y \in \mathtt{desc}(T_i)$.

Consider a `read` operation to ℓ issued by an S-node X, that generates a conflict. Then there exists a $T \in \mathtt{xactions}(C) - \mathtt{done}(C) - \mathtt{ances}(X)$ such that $\ell \in \mathtt{W}(T)$. By Invariant 3, $T \in \mathtt{spine}(\ell)$, and suppose $T = T_i$. The ON model aborts T_i and all other transactions $T^* \in \mathtt{desc}(T_i)$, thereby truncating $\mathtt{spine}(\ell)$ to be $\mathtt{spine}(\ell) \cap \mathtt{ances}(T_{i-1})$ (we never abort $\mathtt{root}(C) = T_1$). In this case, Invariant 3 through Invariant 5 are all maintained. Invariant 6 and Invariant 7 are all also maintained because we have only removed elements from the spine.

Conflicts for a `write` operation are similar; if we detect a conflict with T such that $\ell \in \mathtt{W}(T)$, then we have the same behavior as before. If $\ell \in \mathtt{R}(T)$ but $\neg(\ell \in \mathtt{W}(T))$, then we abort T which only reads ℓ, leaving the spine intact.

After checking for conflicts and aborting the appropriate transactions, X then adds a memory operation v to the computation tree. Let $T^* = \mathtt{lowest}(\{Y | Y \in \mathtt{xAnces}(T) \text{ and } \ell \in W(Y)\})$, i.e., the lowest transactional ancestor of X which has ℓ in its writeset. We know that v gets its

Let $S_\ell = \{T \in \mathtt{xAnces}(v) : \ell \in \mathtt{R}(T)\}$ By the ON model, we know v reads the value of u from $T^* = \mathtt{lowest}(S_\ell)$. Let $T_j = \mathtt{lowest}(\{Y : Y \in \mathtt{xAnces}(T^*) \text{ and } \ell \in W(Y)\})$ be the lowest transaction on $\mathtt{spine}(\ell)$ that is an ancestor of T^*. By Invariant 5, we know that T^* and T_j must have the same pair (u, ℓ) in their readset/writeset, respectively. We know $T_j \in \mathtt{spine}(\ell)$, and if T_j is not the last transaction T_k, on the spine, then T^* must still conflict with T_k. Thus, when we add operation v, we are still reading (u, ℓ) from the end of the spine, thereby maintaining Invariant 4 and Invariant 5. Since x_j already happened, then $x_j <_S v$, and Invariant 6 is satisfied. We have $X \in \mathtt{desc}(T'')$, maintaining Invariant 7.

If the memory operation v to be added is a write, then by similar logic as a read, v is added onto the end of $\texttt{spine}(\ell)$.

On an xend operation for a closed transaction T, consider two cases. If $\ell \in \texttt{R}(T)$, but $\ell \notin \texttt{W}(T)$, then committing (u, ℓ) to $\texttt{xparent}[T]$ does not alter the above invariants because u must be the same as the value x_k from T_k, the last transaction on $\texttt{spine}(\ell)$; $\texttt{spine}(\ell)$ itself remains unchanged. If $\ell \in \texttt{W}(T)$, then we must be committing T_k to some transaction T^* between T_{k-1} and T_k on the spine. If $T^* \neq T_{k-1}$, then T^* becomes the new last transaction on the spine, with pair (ℓ, x_k). If $T_{k-1} = T_k$, then the value (ℓ, x_{k-1}) disappears from the spine.

The commit operation for an open transaction replaces all the values (ℓ, x_i) along the spine. Then, since (ℓ, x_k) goes all the way up the spine, the last two invariants are still preserved.

□

The next three lemmas describe additional structure of the computation tree.

Lemma 7.2 *For all $T \in \texttt{aborted}(C)$ and $T' \in \texttt{activeX}(C)$, if $v \in \texttt{cContent}(T)$, then we have $v \notin \texttt{W}(T')$.* □

Lemma 7.3 *If $v \in \texttt{memOps}(C)$ accesses $\ell \in \mathcal{M}$, then at time $\mathcal{S}(v)$, we have $\texttt{spine}(\ell) \subseteq \texttt{ances}(v)$.* □

Lemma 7.4 *For all $v \in V(C)$, $T \in \texttt{aborted}(C)$, and $w \in \texttt{cContent}(T)$, if $\texttt{end}(T) <_{\mathcal{S}} v$, then we have wHv.* □

The next lemma shows that a memory location written within a transaction remains in the writeset of some active descendant of the transaction.

Lemma 7.5 *Let $w \in \texttt{memOps}(C) \cap \texttt{cContent}(T)$ be a memory operation in a transaction $T \in \texttt{xactions}(C)$, and suppose that $W(w, \ell)$ for some location $\ell \in \mathcal{M}$. Then, at all times t in the range $\mathcal{S}(w) < t < \mathcal{S}(\texttt{end}(T))$, we have $\ell \in \texttt{W}(T')$ for some $T' \in \texttt{desc}(T) \cap \texttt{activeX}(T)$.*

PROOF. We proceed by induction on time. For the base case, at time $\mathcal{S}(w)$, location ℓ is added to $\texttt{W}(\texttt{xparent}[w])$, and $\texttt{xparent}[w] \in \texttt{desc}(T) \cap \texttt{activeX}(T)$. For the inductive step, let $\ell \in \texttt{W}(T')$ for some $T' \in \texttt{desc}(T) \cap \texttt{activeX}(T)$. Once a location is added to a transaction's writeset, it is never removed until the transaction commits or aborts. If $T' = T$, then we are done. Otherwise, we have $T' \in \texttt{pDesc}(T)$ and by definition of $\texttt{cContent}(T)$, it follows that $T' \notin \texttt{open}(C) \cup \texttt{aborted}(C)$. Therefore, at time $\mathcal{S}(\texttt{end}(T'))$, location ℓ is added to $\texttt{W}(\texttt{xparent}[T'])$, at which time $\texttt{xparent}[T']$ is an active descendant of T. □

We can now prove that the *ON* model admits no prefix-races.

Lemma 7.6 *For all $v \in \texttt{memOps}(C)$ and $T \in \texttt{xactions}(C)$, we have $\neg\textsc{Prace}_{\mathcal{S}}(v, T)$.*

PROOF. Suppose for contradiction that $\textsc{Prace}_{\mathcal{S}}(v, T)$. Then, by Definition 16, we have $v \notin V(T)$ (or equivalently, $T \notin \texttt{ances}(v)$), and there exists a $w \in \texttt{cContent}(T)$ such that $\neg(vHw)$ and $\texttt{begin}(T) <_{\mathcal{S}} w <_{\mathcal{S}} v <_{\mathcal{S}} \texttt{end}(T)$, where v and w access the same location $\ell \in \texttt{memOps}(C)$ and one of those accesses is a write.

Consider the case when $W(w, \ell)$. By Lemma 7.5, at time $\mathcal{S}(w)$ we have $\ell \in \texttt{W}(T')$, where $T' \in \texttt{desc}(T)$. At time $\mathcal{S}(v)$, vertex v is added to $\texttt{R}(\texttt{xparent}[v])$, and $\texttt{xparent}[v] \notin \texttt{desc}(T')$,

because otherwise $v \in \texttt{desc}(T') \subseteq \texttt{desc}(T)$. Therefore, at time $\mathcal{S}(v)$, we have $\ell \in \texttt{R}(\texttt{xparent}[v])$ and $\ell \in \texttt{W}(T')$, which violates Invariant 4 in Lemma 7.1.

The case when $R(u, \ell)$ is analogous. □

The next series of lemmas show that the observer function created by the ON machine is the transactional last-writer function according to $\mathcal{S}$.

Lemma 7.7 *For all $T \in \texttt{xactions}(C)$, $T' \in \texttt{activeX}(C)$, and $u \in \texttt{cContent}(T)$, if $T \notin \texttt{committed}(C)$ at time t and $u \in \texttt{W}(T')$ at time t, then $T' \in \texttt{desc}(T)$.*

PROOF SKETCH. One can prove by induction that at any time t such that $\mathcal{S}(u) \leq t < \mathcal{S}(\texttt{end}(T))$, we have $\texttt{h}(u) \subseteq \texttt{xAnces}(u) - \texttt{pAnces}(T)$ and $\texttt{h}(u) \cap (\texttt{open}(T) - \{T\}) = \emptyset$. □

Lemma 7.8 *For any $v \in \texttt{memOps}(C)$, if $\Phi(v) = u$, then $\neg(uHv)$.*

PROOF. Assume for contradiction that uHv holds. Then, there exists $T \in \texttt{pDesc}(\texttt{LCA}(u, v)) \cap \texttt{aborted}(C)$ such that $u \in \texttt{cContent}(T)$. If the ON machine sets $\Phi(v) = u$, then $u \in \texttt{R}(T')$ for some $T' \in \texttt{xAnces}(v)$. By Invariant 5 in Lemma 7.1, it follows that $u \in \texttt{W}(T'')$, where $T'' \in \texttt{ances}(T')$, and hence $T \in \texttt{ances}(T'')$ by Lemma 7.7. Therefore, we have $T \in \texttt{ances}(v)$, and $\texttt{LCA}(u, v) = T \in \texttt{pDesc}(T)$. Contradiction. □

We say that a vertex $v \in \texttt{memOps}(C)$ is ***alive***, denoted $\texttt{alive}(v)$, if $\texttt{h}(v) \cap \texttt{aborted}(C) = \emptyset$.

Lemma 7.9 *Let $w \in V(C)$ be the last vertex in $\mathcal{S}$ such that $W(w, \ell)$ and $\texttt{alive}(w)$. Then, there exists $(T) \in \texttt{spine}(\ell)$ such that $(\ell, w) \in \texttt{W}(T')$.*

PROOF SKETCH. At time $\mathcal{S}(w)$, by Invariant 3 of Lemma 7.1, we have $(\ell, w) \in \texttt{W}(\texttt{xparent}[w])$ and $\texttt{xparent}[w] \in \texttt{spine}(\ell)$. Assume for contradiction that w is not on the spine. Since w is alive, w can only be removed from $\texttt{spine}(\ell)$ by being overwritten by some y such that $W(y, \ell)$ holds, and $w <_{\mathcal{S}} y$ (from Invariant 6 from Lemma 7.1). Since w is the last writer to ℓ which is alive, we have $\neg\texttt{alive}(y)$. One can show that $\neg\texttt{alive}(w)$ in this case. □

Lemma 7.10 *For $u, v \in \texttt{memOps}(C)$ that both access a memory location $\ell \in \mathcal{M}$, if $\Phi(v) = u$, then for any $w \in \texttt{memOps}(C)$ such that $u \prec_{\mathcal{S}} w \prec_{\mathcal{S}} v$ and $W(w, \ell)$, we have wHv.*

PROOF. Assume for the purpose of contradiction that there exists a $w \in \texttt{memOps}(C)$ such that $u \prec_{\mathcal{S}} w \prec_{\mathcal{S}} v$, $W(w, \ell)$, and $\neg wHv$. Consider the last such w.

If $w \in \texttt{cContent}(T)$ for some $T \in \texttt{aborted}(C^{(\mathcal{S}(v))})$, then by Lemma 7.4 we have wHv.

If w is not in the contents of any aborted transaction at time $\mathcal{S}(v)$, then by Lemma 7.9, we have $w \in \texttt{W}(T)$ for some transaction $T \in \texttt{spine}(\ell)$ and $T \in \texttt{ances}(v)$ by Lemma 7.3. Let $T_R = \texttt{lowest}(\{T \in \texttt{xAnces}(v) : \ell \in \texttt{R}(T)\})$, and let $T_W = \texttt{lowest}(\{T \in \texttt{xAnces}(v) : \ell \in \texttt{W}(T)\})$. If $\Phi(v) = u$, then we have $u \in \texttt{R}(T_R)$, since the ON machine always reads from the lowest ancestor that has ℓ in its readset. By Invariant 5, we have $u \in \texttt{W}(T_W)$, but since $u <_{\mathcal{S}} w$, we have $T_W \in \texttt{pAnces}(T)$ by Invariant 6 in Lemma 7.1. Therefore, T is a lower ancestor of v than T_W, contradicting the fact that T_W is the lowest ancestor of v with ℓ in its writeset. □

We now can prove that the observer function for the ON model is the transactional last-writer function.

Lemma 7.11 *If the ON model generates an execution* (C, Φ)*, then* $\Phi = \mathcal{X}_S$.

PROOF. This proof is technical and is provided in Appendix .3. □

Theorem 7.12 *The ON model implements prefix race-free freedom.*

PROOF. Combine Lemmas 7.6 and 7.11. □

7.4 Discussion

Open Nesting was proposed as a loophole in order to increase the performance and scalability of transactional programs. The open-nesting methodology incorporates the ***open-nested commit mechanism*** [MCC+06, MH06]. When an open-nested transaction Y (nested inside transaction X) commits, Y's changes are committed to memory and Y's read and write sets are discarded. Thus, the TM system no longer detects conflicts with X due to memory accessed by Y. In this methodology, the programmer considers Y's internal memory operations to be at a "lower level" than X; thus X should not care about the memory accessed by Y when checking for conflicts. Instead, Y must acquire an ***abstract lock*** based on the high-level operation that Y represents and propagate this lock to X, so that the TM system can perform concurrency control at an abstract level. Also, if X aborts, it may need to execute ***compensating actions*** to undo the effect of its committed open-nested subtransaction Y. Moss in [Mos06] illustrates use of open nesting with an application that uses a B-tree. Ni et al.[NMAT+07] describe a software TM system that supports the open-nesting methodology.

As we have seen in this section, an unconstrained use of the open-nested commit mechanism can lead to anomalous program behavior that can be tricky to reason about. We believe that one reason for the apparent complexity of open nesting is that the mechanism and the methodology make different assumptions about memory. Consider a transaction Y 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 will ignore *all* the memory operations inside Y. Say Y accesses two memory locations ℓ_1 and ℓ_2, and X does not care about changes made to ℓ_2, but does care about ℓ_1. The TM system can not distinguish between these two accesses, and will commit both in an open-nested manner, leading to anomalous behavior.

Researchers *have* demonstrated specific examples [CMC+07, NMAT+07] that safely use an open-nested commit mechanism. These examples work, however, because the inner (open) transactions never write to any data that is accessed by the outer transactions. 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. The literature on TM offers relatively little in the way of formal programming guidelines which one can follow to have provable guarantees of safety when using open-nested commits.

In the next chapter, we shall see a new TM design that constrains the use of open nesting in order to provide gurantees that are easier to reason about.

Chapter 8

Safe Open-Nested Transactions Through Ownership

In the Chapter 7, we saw that open nesting provides an unintuitive and noncomposable memory model that is hard to reason about. In this chapter, we bridge the gap between memory-level mechanisms for open nesting and the high-level view by explicitly integrating the notions of "transactional modules" (Xmodules) and "ownership" into the TM system. The ***ownership-aware TM system*** allows the programmer to safely use the methodology of open nesting, because the runtime's behavior more closely reflects the programmer's intent. In addition, the structure imposed by ownership allows a language and runtime to enforce properties needed to provide provable guarantees of "safety" to the programmer. Therefore, a concurrency platform that provides ownership aware transactions provides concurrency as well as safety to a programmer who is using transactions.

The chapter is organized as follows. Section 8.1 explains the precise contributions of the work described in this chapter. Section 8.2 present an overview of ownership-aware TM and highlights key features using an example application. Section 8.3 describes language constructs for specifying Xmodules and ownership. In Section 8.4, we extend the transactional computation framework from Chapter 6 to formally incorporate Xmodules and ownership. Section 8.5 describes the precise operational model for ownership-aware transactions, and Section 8.6 gives a formal definition of serializability by modules, and a proof-sketch that the operational model guarantees this definition. Section 8.7 provides conditions under which the ownership-aware TM not exhibit semantic deadlocks. Section 8.8 concludes with a discussion of some related work.

8.1 Contributions

The contributions of this work on ownership-aware transactions are as follows:

1. We suggest a concrete set of guidelines for sharing of data and interactions between transactional modules, called Xmodules. Xmodules can be thought of as software modules that own and manage their own data and provide external functions to provide services to other Xmodules.

This work was done in collaboration with Angelina Lee and Jim Sukha [ALS09].

2. We describe how the Xmodules and ownership can be specified in a Java-like language and propose a type system that enforces most of the above-mentioned guidelines in the programs written using this language extension.

3. We formally describe the operational model for ownership-aware TM, called the OAT model, which uses a new ***ownership-aware commit mechanism***. The ownership-aware commit mechanism is a compromise between an open-nested and a closed-nested commit; when a transaction T commits, a change to memory location ℓ is committed globally if ℓ belongs to the module of T; otherwise, the read or write to ℓ is propagated to T's parent transaction. Unlike an ordinary open-nested commit, the ownership-aware commit treats memory locations differently depending on which Xmodule owns the location. Note that the ownership-aware commit is still a mechanism; programmers must still use it in combination with abstract locks and compensating actions to implement the full methodology.

4. We prove 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. Ownership-aware commit is the same as open-nested commit if no module ever accesses data belonging to other modules. Thus, one corollary of our theorem is that open-nested transactions are serializable when modules do not share data. This observation explains why researchers [CMC+07, NMAT+07] have found it natural to use open-nested transactions in the absence of sharing, in spite of the apparent semantic pitfalls.

5. We prove that under certain restricted conditions, a computation executing under the OAT model can not enter a semantic deadlock.

In later sections, we distinguish between the variations of nested transactions as follows. We say that a transaction Y is ***vanilla open nested*** 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.

8.2 Ownership-Aware Transactions

In this section, we give an overview of ownership-aware TM. To motivate the need for the concept of ownership in TM, we first present an example application which might benefit from open nesting. We then introduce the notion of an Xmodule and informally explain the programming guidelines when using Xmodules. Finally, we highlight some of the key differences between ownership-aware TM and a TM with vanilla open nesting. In this section, we present the intuitive descriptions of the concepts in ownership-aware TM; we defer formal definitions until later sections.

Example application

We describe an example application for which one might use open-nested transactions. This example is similar to the one in [Mos06], 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 programs to have multiple levels of abstraction, we choose a modular application. Consider a user application which 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 hashmap, 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 user application 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 user application is able to request a checkpoint operation which flushes the buffer to disk.

This application is modular, with five natural modules — the user application (`UserApp`), the database (`DB`), the binary search tree (`BST`), the hashtable (`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 non-intuitive 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 Y writes to some field of the book b (memory location ℓ_1), and 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 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; this commit increases concurrency, because 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.[1] 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.

[1] Note that abstract locks [Mos06] 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.

Xmodules and the ownership-aware commit mechanism

The ownership-aware TM system requires that programs be organized into Xmodules. Intuitively, an Xmodule A is as a stand-alone entity that contains data and transactional methods; an Xmodule owns data that it privately manages, and uses its methods to provide public services to other modules. During program execution, a call to a method from Xmodule A generates a transaction instance (e.g., X). If this method in turn calls another method from an Xmodule B, an additional transaction Y, safe nested inside X, is created only if $A \neq B$. 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 is in a transaction T's read or write set, the ownership-aware commit of a transaction T commits this access globally only if T is generated by the same Xmodule that owns ℓ; in this case, we say that T is "responsible" for that access to ℓ. Otherwise, the read or write to ℓ is propagated up to the read or write set of T's parent transaction; that is, the TM system behaves as though T 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.

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). In our system, Xmodules are arranged as a ***module tree***, denoted as $\mathcal{D}$. In $\mathcal{D}$, an Xmodule B is a child of A if B is "encapsulated by" A. 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 19 *We impose two rules on Xmodules based on the module tree:*

1. **Rule 1**: *A method of an Xmodule A can access a memory location ℓ directly only if ℓ is either owned by A or an ancestor of A in the module tree. This rule means that an ancestor Xmodule B of A may pass data down to a method belonging to A, but a transaction from module A can not directly access any "lower-level" memory.*

2. **Rule 2**: *A method from A can call a method from B only if B is the child of some ancestor of A, and $\texttt{xid}(B) > \texttt{xid}(A)$ (i.e., if B is "to the right" of A in the module tree). This rule requires that an Xmodule can call methods of some (but not all) lower-level Xmodules.*[2]

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

[2] An Xmodule can, in fact, call methods within its own Xmodule or from its ancestor Xmodules, but we model these calls differently. We explain these cases condition at the end of this section.

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; some transaction is always "responsible" for every memory access (proved in Section 8.5). In addition, these rules and the ownership-aware commit mechanism guarantee that transactions satisfy the property of "serializability by modules" (proved in Section 8.6).

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, our formalism requires that Xmodules be partially ordered; if an Xmodule A can call Xmodule B, then conceptually A is at a higher level than B (i.e., $\texttt{xid}(A) < \texttt{xid}(B)$), and thus B can not call A. If two components of the program call each other, then, conceptually, neither of these components is at a higher-level than the other, and we would require that these two components be combined into the same Xmodule.

Xmodules in the example application

Consider a Java implementation of the example application described earlier. 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 `Hashmap` for the binary search tree and hashmap 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 `Hashmap`, 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 our example, we identified five Xmodules– `UserApp`, `DB`, `BST`, `Hashmap`, and `Logger`; these classes are stand-alone entities which have 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 `Hashmap` to maintain their internal state.

We organize the Xmodules of the application into the module tree shown in Figure 8.1. `UserApp` is encapsulated by `world`, `DB` and `Logger` are encapsulated under `UserApp`; `BST` and `Hashmap` are encapsulated under `DB`. By dividing 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.

Let us consider the implications of Definition 19 for the example. Due to Rule 1, all of `DB`, `BST`, `Hashmap`, and `Logger` can directly access data owned by `UserApp`, but the `UserApp` can not 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 `Hashmap`, 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

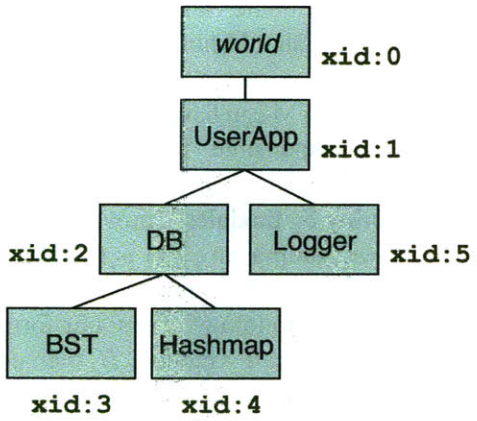

Figure 8.1: A module tree $\mathcal{D}$ for the program described in Section 8.2. The `xid`'s are assigned according to a left-to-right depth-first tree walk, numbering Xmodules in increasing order, starting with `xid(world)` = 0.

the relationship between `BST` and `Logger`. The `BST` Xmodule can call methods from `Logger`, but the `BST` can not 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 Y is open nested inside transaction X. Let v_0 be the initial value of location ℓ, and suppose Y writes value v_1 to location ℓ and then commits. Now a transaction Z 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, Y. This behavior might seem counterintuitive.

Now consider the same example for ownership-aware transactions. Say X is generated by a method of Xmodule A and Y is generated by a method of Xmodule B. If B owns ℓ, X can not access ℓ, since $\texttt{xid}(A) < \texttt{xid}(B)$ (by Definition 19, Rule 2), and no transaction from a higher-level module can access data owned by a lower-level module (by Definition 19, Rule 1). Thus, the problem does not arise. If B does not own ℓ, the ownership-aware commit of Y will not commit the changes to ℓ globally and ℓ will be propagated to X's write set. Therefore, if Z tries to access ℓ before X commits, the TM system will detect a conflict. Thus X can not see an inconsistent value for ℓ.[3]

[3]For simplicity, we have described the case where Y is directly nested inside X. The case where Y is more deeply open nested inside X behaves in a similar fashion.

Callbacks

At first glance, the assumptions we have made regarding methods of Xmodules seem somewhat restrictive. In the description thus far, we prohibit an Xmodule A from calling another transactional method from A or a proper ancestor of A. In particular, it appears as though our model disallows callbacks. Our model, however, does permit both these cases; we simply model these calls differently.

If a method X from Xmodule A calls another method Y from an ancestor Xmodule B, 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 19 allows X to access data belonging to B 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 B or B's ancestors. We say that Y is a ***proper callback*** method for Xmodule B if its nested calls are all proper callback methods belonging to Xmodules which are ancestors of B. In our formal model in Section 8.4, we assume that we only have proper callbacks and model them as direct memory accesses, allowing us to ignore them in the formal definitions.

Closed-nested transactions

In our model, every method call that crosses an Xmodule boundary automatically generates a safe-nested transaction. Ownership-aware TM can effectively provide closed-nested transactions, however, with appropriate specifications of ownership. If an Xmodule A owns no memory, but only operates on memory belonging to its proper ancestors, then transactions of A will effectively be closed nested. In the limit, if the programmer specifies that all memory is owned by the `world` Xmodule, then all changes in any transaction's read or write set are propagated upwards; thus all ownership-aware commits behave exactly as closed-nested commits.

8.3 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, we can guarantee some notion of serializability only if a program has Xmodules which conform to the rules in Definition 19. In this section, we describe language constructs and a type system that can be used to specify Xmodules and ownership in a Java-like language. Our type system — the ***OAT type system*** — statically enforces some of the restrictions described in Definition 19.

The OAT type system extends the ownership types of Boyapati et al. [BLS03], which is described first in this section. We then describe extensions to this type system to enforce some of the restrictions in Definition 19. Next, we present code for parts of the example application described in Section 8.2. Finally, we discuss some restrictions required by Definition 19 which the OAT type system does not enforce statically. The type system's annotations, however, enable dynamic checks for these restrictions.

Boyapati et al.'s parametric ownership type system

The type system of Boyapati et al. provides a mechanism for specifying ownership of objects. The type system enforces the properties stated in Lemma 8.1.

Lemma 8.1 *The type system in [BLS03] 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.*

Boyapati et al.'s type system requires ownership annotations to class definitions and type declarations to guarantee Lemma 8.1. 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, the type system of [BLS03] 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.

The type system enforces the properties in Lemma 8.1 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, $\forall i \in 2..n$, (denoted by $o_1 \preceq o_i$ henceforth).

It is shown in [BLS03] that these type checks guarantee the properties of Lemma 8.1.

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 the class declaration of type `T1` has formal tags $\langle f_1, f_2, f_3 \rangle$, and inside `T1`'s definition, some type `T2` object is instantiated with ownership tags $\langle f_2, f_3 \rangle$. The type system can not 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

The ownership tree described in [BLS03] exhibits some of the same properties as the module tree we described in Section 8.2; however, the type system and ownership scheme of [BLS03] do not enforce two major requirements of our system.

- In [BLS03], any object can own other objects. Our rules, however, require that only Xmodules own other objects.
- In [BLS03], an object can call any of its ancestor's siblings. Our rules (namely Definition 19), however, dictate that an Xmodule A can only call its ancestor's siblings to the right.

With these requirements in mind, we extend Boyapati et al.'s 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 Boyapati et al.'s 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 8.2) is essentially the ownership tree with all non-Xmodule objects removed.

The second requirement is more complicated to enforce. First, we extend 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 type system of [BLS03]. 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 A and B; for instance, if an Xmodule C instantiates A and B with owners `this[i]` and `this[i+1]`, respectively, then A appears to the left of B in the module tree.

Finally, for technical reasons, the OAT system prohibits all Xmodules A from declaring primitive fields. If A had primitive fields, then by Boyapati et al.'s type system, these fields are owned by the A's parent. Since this property seems counter-intuitive, we opted to disallow primitive fields for Xmodules.

In summary, the OAT type system performs these checks:

1. Within the class definition of type `T1`, only the tags $\{f_1, f_2, \ldots f_n\} \cup \{\texttt{this}, \texttt{world}\}$ are visible.
2. In a class declaration, a variable c_2 with type $\texttt{T2}\langle f_2, \ldots\rangle$ can be assigned to a variable c_1 with type $\texttt{T1}\langle f_1, \ldots\rangle$ 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 $\texttt{T}\langle o_1, o_2, \ldots o_n\rangle$, we must have, for all $i \in \{2, \ldots n\}$, 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).[4]

[4] In the ownership tree, for any Xmodule A, the OAT type system implicitly assigns non-Xmodule children of A higher indices than the Xmodule children of A, unless the user specifies otherwise.

4. The ownership tag `this` can only be used within the definition of a class that extends `Xmodule`.

5. Xmodule objects can not have primitive-type fields.

The first three checks are analogous to the checks in Boyapati et al.'s type system. 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 [ALS08].

Example application using the OAT type system

Figure 8.2 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.

Note that 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, i.e. at line 6.

The OAT type system's guarantees

The following lemma about the OAT type system can be proved in a reasonably straightforward manner using Lemma 8.1.

Lemma 8.2 *The OAT type system guarantees the following properties.*

1. *An Xmodule Acan access a (non-Xmodule) object b with ownership tag o_b only if $A \preceq o_b.name$.*

2. *An Xmodule Acan call a method in another Xmodule Bwith owner o_B only if one of the following is true:*

   (a) $A = o_B.name$ (i.e. A owns B);

   (b) The least common ancestor of A and B in the module tree is $o_B.name$.

   (c) $B \succeq A$ (i.e. B is an ancestor of A).

```
1   public class UserApp<app0> extends Xmodule {
2       private Logger<this[1], this[2]> logger;
3       private DB<this[0], this[1], this[2]> db;
          ...
4       public UserApp() {
5           logger = new Logger<this[1], this[2]>();
6           db = new DB<this[0], this[1], this[2]>(logger);
7       }
8   }

9   public class DB<db0, log0, data0>
10          extends Xmodule where (log0 < data0) {
11      private Logger<log0, data0> logger;
12      private BST<this[0], log0, data0> bst;
13      private Hashmap<this[1], log0, data0> hashmap;
14      public DB(Logger<log0, data0> logger) {
15          this.logger = logger;
              ...
16      }
17  }
```

Figure 8.2: Specifying Xmodules and ownership for the example application described in Section 8.2.

Lemma 8.2 does not, however, guarantee all the properties we want from Xmodules (i.e., Definition 19). In particular, Lemma 8.2 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 A, all children of A 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 A from smallest to largest index, and assign the ***xid*** to each Xmodule as described in Section 8.2. Then, the following result holds:

Theorem 8.3 *For a program with unique owner indices, in addition to Lemma 8.2, the OAT type system guarantees that if the least common ancestor of Xmodules A and B in the module tree is $o_B.\,name$, then A can call a method in B only if* $\texttt{xid}(A) < \texttt{xid}(B)$.

PROOF.

We prove (by contradiction) that if least common ancestor of A and B in the module tree is $o_B.\,name$, and $\texttt{xid}(A) > \texttt{xid}(B)$, then A can not have a formal tag with value o_B. Therefore, it can not declare a type with owner tag o_B, and can not access B.

Let C be the least common ancestor of A. Since $C = o_B.\,name$, we know that C is B's parent. Let D be the ancestor of A which is B's sibling, and let o_D be D's ownership tag (i.e., the tag with which D is instantiated). Since B and D have the same parent (i.e. C) in the module tree, we have $o_B.\,name = o_D.\,name = C$. Since $\texttt{xid}(A) > \texttt{xid}(B)$, A is to the right of B in the ownership tree. Therefore, D, which is an ancestor of A, is to the right of B in the ownership tree. Therefore, we have $o_D.\,index > o_B.\,index$.

Assume for contradiction that A does have o_B as one of its tags. Using Lemma 8.1, one can show that the only way for A to receive tag o_B is if D also has a formal tag with value o_B. Thus,

D's first formal owner tag has value o_D and another one of its formals has value o_B.

Let $E_0 = D$, and consider the chain of Xmodule instantiations where Xmodule E_i instantiated E_{i-1}. E_1 has to instantiate D (which is the same as E_0) using its formal ownership tags $\langle f_a^1, f_b^1, ... \rangle$, where f_a^1 has value o_D and f_b^1 has value o_B. (We must have f_a^1 as the first formal, since o_D is the owner of D. Without loss of generality, we can have f_b^1 be the second formal since the type system does not care about the ordering of formal tags after the first one.)

Since $o_B.\,name = o_D.\,name = C$, this chain of instantiations must lead back to C, since that is the only Xmodule that can create ownership tags with values o_B and o_D in its class definition (using the keyword `this`). [5] Let $E_k = C$. For the class declaration of each of the Xmodules E_i for $1 \leq i < k$, the following must be true.

- E_i must have formals f_a^i and f_b^i, with values o_D and o_B, respectively, and E_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_B and o_D down to $P_0 = Q$. The second condition is true for the Xmodules by induction. In the base case, P_1 must know that $f_a^1 < f_b^1$; 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}.

Finally, E_{k-1} is instantiated in the class file corrsponding to $E_k = C$. In this declaration, the formal f_a^k with value o_D is instantiated with `this`$[x]$. Similarly, f_b^k with value o_B is instantiated with `this`$[y]$. Since the class definition of P_k type checks, we must have $f_a^k < f_b^k$. This check contradicts our original assumption that $x > y$ however, since if $x > y$ our type check should fail. Therefore, we must have $o_D.\,index < o_B.\,index$.

□

Theorem 8.3 only modifies the Condition 2b of Lemma 8.2. Therefore, Lemma 8.2 along with Theorem 8.3 imposes restrictions on every Xmodule A which are only slightly weaker than the restrictions required by Definition 19. Condition 1 in Lemma 8.2 corresponds to Rule 1 of Definition 19. 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 19. This case is modeled differently, as we explained in Section 8.2.

The OAT type system is a best-effort type system to check for the restrictions required by Definition 19. The OAT type system can not fully guarantee, however, that a type-checked program does not violate Definition 19. Specifically, the OAT type system can not always detect the following violations statically. First, if the program does not have unique owner indices, then C may instantiate both A and B with the same index. Then, by Lemma 8.2, A and B, can call each other's methods, and we can get cyclic dependencies between Xmodules.[6] Second, the program may perform improper callbacks. Say a method from A calls back to method B from C. An improper callback B can call a method of B, even though the type system knows that A is to the right of B.

[5] Note that C could be the `world` Xmodule, in which case both o_B and o_D were created in the `main` function using the `world` keyword.

[6] Since all non-Xmodule objects are implicitly assigned higher indices than their Xmodule siblings, these non-Xmodule objects can not introduce cyclic dependencies between Xmodules.

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 19.

To have an ownership-aware TM which guarantees exactly Definition 19, one needs to impose additional dynamic checks. The runtime system can use the ownership tags to build a module tree during runtime, and use this module tree to perform dynamic checks to verify that every Xmodule has unique owner indices and contains only proper callbacks. The runtime system can do this by dynamically inferring indices according to which Xmodule calls which other Xmodule, and reporting an error if there is any circular calling.[7]

8.4 Computations with Xmodules

In this section, we formally define the structure of transactional programs with Xmodules. This section converts the informal explanation from Section 8.2 into a formal model that we later use to prove properties of ownership-aware TM. We build on top of the computation tree framework described in Chapter 6. We add Xmodules and ownership to this framework, and provide the formal statement of Definition 19.

.

As mentioned in Section 8.2 we consider programs that contain Xmodules. In our theoretical framework, we consider traces generated by a program which is organized into a set $\mathcal{N}$ of Xmodules. Each Xmodule $A \in \mathcal{N}$ has some number of methods and a set of memory locations associated with it.

We partition the set of all memory locations $\mathcal{M}$ into sets of memory owned by each Xmodule. Let $\texttt{modMemory}(A) \subseteq \mathcal{M}$ denote the set of memory locations owned by A. For a location $\ell \in \texttt{modMemory}(A)$, we say that $\texttt{owner}(\ell) = A$. When a method of Xmodule A is called by a method from a different Xmodule, a safe-nested transaction T is generated.[8] We use the notation $MT = A$ to associate the instance T with the Xmodule A. We also define the instances associated with A as

$$\texttt{modXactions}(A) = \{T \in \texttt{xactions}(\mathcal{C}) \ : \ MT = A\}\,.$$

As mentioned in Section 8.2, Xmodules of a program are arranged as the module tree, denoted by $\mathcal{D}$. Each Xmodule is assigned an `xid` according to a left-to-right depth-first tree walk, with the root of $\mathcal{D}$ being `world` with $\texttt{xid} = 0$. Denote the parent of Xmodule A in $\mathcal{D}$ as $\texttt{modParent}(A)$, and the ancestors of A as $\texttt{modAnces}(A)$ (include A itself). Similarly, let $\texttt{modDesc}(A)$ be the set of A's descendants. We say that $M\texttt{root}(\mathcal{C}) = \texttt{world}$, i.e., the root of the computation tree is a transaction associated with the `world` Xmodule.

We 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 Definition 20.

Definition 20 (Formal Restatement of Definition 19) *A program with a module tree $\mathcal{D}$ should generate only traces $(\mathcal{C}, \Phi)$ which satisfy the following rules:*

1. *For any memory operation v which accesses a memory location ℓ, let $T = \texttt{xparent}[v]$. Then $\texttt{owner}(\ell) \in \texttt{modAnces}(MT)$.*

[7] It is possible to statically check for unique owner indices by imposing additional restrictions on the program. We opted, however, to describe a more flexible programming model with weaker static guarantees.

[8] As we explained in Section 8.2, callbacks are handled differently.

2. *Let* $X, Y \in \texttt{xactions}(\mathcal{C})$ *be transaction instances such that* $MX = A$ *and* $MY = B$. *We can have* $X = \texttt{xparent}[Y]$ *only if* $\texttt{modParent}(B) \in \texttt{modAnces}(A)$, *and* $\texttt{xid}(A) < \texttt{xid}(B)$.

8.5 The OAT Model

In this section, we describe the OAT model, an abstract execution model for TM with ownership and Xmodules. The novel feature of the OAT model is that it uses the structure of Xmodules to provide a commit mechanism which can be viewed as a hybrid of closed and open-nested commits. The OAT model presents an operational semantics for TM, and is not intended to describe an actual implementation, although these semantics can be used to guide an implementation.

Overview

The TM system is modeled as a nondeterministic state machine with two components: a ***program*** and a ***runtime system***. The runtime system, which we call the OAT model, dynamically constructs and traverses a computation tree $\mathcal{C}$ as it executes instructions generated by the program. The OAT model maintains a set of ***ready*** nodes, denoted by $\texttt{ready}(\mathcal{C}) \subseteq \texttt{nodes}(\mathcal{C})$, and at every step, the OAT model nondeterministically chooses one of these ready nodes $X \in \texttt{ready}(\mathcal{C})$ to issue the next instruction. The program then issues one of the following instructions (whose precondition is satisfied) on X's behalf: `fork`, `join`, `xbegin`, `xend`, `xabort`, `read`, or `write`. For shorthand, we sometimes say that X issues an instruction.

The OAT model describes a sequential semantics, that is, we assume at every time step t, a program issues a single instruction. The parallelism in this model arises from the fact that at a particular time, several nodes can be ready, and the runtime nondeterministically chooses which node to issue an instruction.

In the rest of this section, we give a detailed description of the OAT model. First, we describe the state information maintained by the OAT model and define the notation we use to refer to this state. Second, we describe how the OAT model constructs and traverses the computation tree as instructions are issued. Then, we describe how the OAT model handles memory operations (i.e., `read` and `write`), conflict detection, and transaction commits, and transaction aborts.

State information and notation

As the OAT model executes instructions, it dynamically constructs the computation tree $\mathcal{C}$. For each of the sets defined in Section 8.4 (e.g., $\texttt{nodes}(\mathcal{C})$, $\texttt{spNodes}(\mathcal{C})$, $\texttt{memOps}(\mathcal{C})$, $\texttt{xactions}(\mathcal{C})$, etc.), we define corresponding time-dependent versions of these sets by indexing them with an additional time argument. For example, we define the set $\texttt{nodes}(t, \mathcal{C})$ denotes the set of nodes in the computation tree after t time steps have passed. The generalized sets from Section 8.4 are monotonically increasing, i.e., once an element is added to the set, it is never removed at a later time t. Sometimes for shorthand, we omit the time argument when it is clear that we are referring to a particular fixed time t.

At any time t, each internal node $A \in \texttt{spNodes}(t, \mathcal{C})$ has a ***status*** field $\texttt{status}[A]$. These status fields change with time. If $A \in \texttt{xactions}(t, \mathcal{C})$, i.e., A is a transaction, then $\texttt{status}[A]$ can be one of `COMMITTED`, `ABORTED`, `PENDING`, or `PENDING_ABORT`. Otherwise, $A \in \texttt{spNodes}(t, \mathcal{C}) - \texttt{xactions}(t, \mathcal{C})$ is either a P-node or a nontransactional S-node; in this case, $\texttt{status}[A]$ can either

be WORKING or SYNCHED. We define several abstract sets for the tree based on this status field. The first 6 sets partition the $\texttt{spNodes}(t, \mathcal{C})$, the set of internal nodes of the computation tree. The last 4 sets categorize transactions and nodes as being either active or complete.

1. $\texttt{pending}(t, \mathcal{C}) = \{X \in \texttt{xactions}(t, \mathcal{C}) \ : \ \texttt{status}[Z] = \texttt{PENDING}\}$ (Pending transactions).
2. $\texttt{pendingAbort}(t, \mathcal{C}) = \{X \in \texttt{xactions}(t, \mathcal{C}) \ : \ \texttt{status}[Z] = \texttt{PENDING\_ABORT}\}$ (Aborting transactions).
3. $\texttt{committed}(t, \mathcal{C}) = \{X \in \texttt{xactions}(t, \mathcal{C}) \ : \ \texttt{status}[Z] = \texttt{COMMITTED}\}$ (Committed transactions).
4. $\texttt{aborted}(t, \mathcal{C}) = \{X \in \texttt{xactions}(t, \mathcal{C}) \ : \ \texttt{status}[Z] = \texttt{ABORTED}\}$ (Aborted transactions).
5. $\texttt{working}(t, \mathcal{C}) = \{Z \in \texttt{spNodes}(t, \mathcal{C}) - \texttt{xactions}(t, \mathcal{C}) \ : \ \texttt{status}[Z] = \texttt{WORKING}\}$ (Working nodes).
6. $\texttt{synched}(t, \mathcal{C}) = \{Z \in \texttt{spNodes}(t, \mathcal{C}) - \texttt{xactions}(t, \mathcal{C}) \ : \ \texttt{status}[Z] = \texttt{SYNCHED}\}$ (Synched nodes).
7. $\texttt{activeX}^{(t)}(\mathcal{C}) = \texttt{pending}(t, \mathcal{C}) \cup \texttt{pendingAbort}(t, \mathcal{C})$ (Active transactions).
8. $\texttt{activeN}(t, \mathcal{C}) = \texttt{activeX}^{(t)}(\mathcal{C}) \cup \texttt{working}(t, \mathcal{C})$. (Active nodes).
9. $\texttt{doneX}(t, \mathcal{C}) = \texttt{committed}(t, \mathcal{C}) \cup \texttt{aborted}(t, \mathcal{C})$ (Complete transactions).
10. $\texttt{doneN}(t, \mathcal{C}) = \texttt{doneX}(t, \mathcal{C}) \cup \texttt{synched}(t, \mathcal{C})$ (Complete nodes).

The OAT model maintains a set of ***ready*** S-nodes, denoted as $\texttt{ready}(t, \mathcal{C})$. We discuss the properties of ready nodes later, in Section 8.5. Note that $\texttt{ready}(t, \mathcal{C})$, and the sets defined above which are subsets of $\texttt{activeN}(t, \mathcal{C})$ are not monotonic, because completing nodes removes elements from these sets.

For the purposes of detecting conflicts, at any time t, for any active transaction T, i.e., $T \in \texttt{activeX}^{(t)}(\mathcal{C})$, the OAT model maintains a ***read set*** $\texttt{R}(t, T)$ and a ***write set*** $\texttt{W}(t, T)$ for T. The read set $\texttt{R}(t, T)$ is a set of pairs (ℓ, v), where $\ell \in \mathcal{M}$ is a memory location and $v \in \texttt{memOps}(t, \mathcal{C})$ is a memory operation that reads from ℓ. We define $\texttt{W}(t, T)$ similarly. We represent main memory as the read set/write set of $\texttt{root}(\mathcal{C})$. At time $t = 0$, we assume $\texttt{R}(0, \texttt{root}(\mathcal{C}))$ and $\texttt{W}(0, \texttt{root}(\mathcal{C}))$ initially contain a pair $(\ell, \perp)$ for all locations $\ell \in \mathcal{M}$.

In addition to the basic read and write sets, we also define ***module read set*** and ***module write set*** for all transactions $T \in \texttt{activeX}^{(t)}(\mathcal{C})$. Module read set is defined as

$$\texttt{modR}(t, T) = \{(\ell, v) \in \texttt{R}(t, T) \ : \ \texttt{owner}(\ell) = MT\} .$$

In other words, $\texttt{modR}(t, T)$ is the subset of $\texttt{R}(t, T)$ that accesses memory owned by T's Xmodule MT. Similarly, we define the ***module write set*** as

$$\texttt{modW}(t, T) = \{(\ell, v) \in \texttt{W}(t, T) \ : \ \texttt{owner}(\ell) = MT\} .$$

The OAT model maintains two invariants on $\texttt{R}(t, T)$ and $\texttt{W}(t, T)$. First, $\texttt{W}(t, T) \subseteq \texttt{R}(t, T)$ for every transaction $T \in \texttt{xactions}(t, \mathcal{C})$, i.e., a write also counts as a read. Second, $\texttt{R}(t, T)$ and

$\mathtt{W}(t,T)$ each contain at most one pair (ℓ,v) for any location ℓ. Thus, we use the shorthand $\ell \in \mathtt{R}(t,T)$ to mean that there exists a node u such that $(\ell,u) \in \mathtt{R}(t,T)$, and similarly for $\mathtt{W}(t,T)$. We also overload the union operator: at some time t, an operation $\mathtt{R}(T) \leftarrow \mathtt{R}(T) \cup \{(\ell,u)\}$ means we construct $\mathtt{R}(t+1,T)$ by

$$\mathtt{R}(t+1,T) = \{(\ell,u)\} \cup (\mathtt{R}(t,T) - \{(\ell,u') \in \mathtt{R}(t,T)\}) .$$

In other words, we add (ℓ,u) to $\mathtt{R}(T)$, replacing any $(\ell,u') \in \mathtt{R}(t,T)$ that existed previously.

Constructing the computation tree

In the OAT model, the runtime constructs the computation tree in a straightforward fashion as instructions are issued. For completeness, however, we give a detailed description of this construction.

Initially, at time $t = 0$, we begin with only the root node in the tree, i.e., $\mathtt{nodes}(0,\mathcal{C}) = \mathtt{xactions}(0,\mathcal{C}) = \{\mathtt{root}(\mathcal{C})\}$. This root node also begins as ready, i.e., $\mathtt{ready}(0,\mathcal{C}) = \{\mathtt{root}(\mathcal{C})\}$. Throughout the computation, the status of the root node of the tree is always PENDING.

A new internal node is created if the OAT model picks ready node X and X issues a `fork` or `xbegin` instruction. If X issues a `fork`, then the runtime creates a P-node P as a child of X, and two S-nodes S_1 and S_2 as children of P, all with status WORKING. The `fork` also removes X from $\mathtt{ready}(\mathcal{C})$ and adds S_1 and S_2 to $\mathtt{ready}(\mathcal{C})$. If X issues an `xbegin`, then the runtime creates a new transaction $Y \in \mathtt{xactions}(\mathcal{C})$ as a child of X, with $\mathtt{status}[Y] = \mathtt{PENDING}$, removes X from $\mathtt{ready}(\mathcal{C})$, and adds Y to $\mathtt{ready}(\mathcal{C})$.

The OAT model completes a nontransactional S-node $Z \in \mathtt{ready}(t,\mathcal{C}) - \mathtt{xactions}(t,\mathcal{C})$ (which must have $\mathtt{status}[Z] = \mathtt{WORKING}$) by having Z issue a `join` instruction. The `join` instruction first changes $\mathtt{status}[Z]$ to SYNCHED. In the tree, since $\mathtt{parent}[Z]$ is always a P-node, Z has exactly one sibling. If Z is the first child of $\mathtt{parent}[Z]$ to be SYNCHED, the OAT model removes Z from $\mathtt{ready}(\mathcal{C})$. Otherwise, Z is the last child of $\mathtt{parent}[Z]$ to be SYNCHED, and the OAT model removes Z and $\mathtt{parent}[Z]$ from $\mathtt{ready}(\mathcal{C})$, changes the status of both Z and $\mathtt{parent}[Z]$ to SYNCHED, and adds $\mathtt{parent}[\mathtt{parent}[Z]]$ to $\mathtt{ready}(\mathcal{C})$.

The OAT model can complete a transaction $X \in \mathtt{ready}(t,\mathcal{C})$ by having it issue either an `xend` or `xabort` instruction. If $\mathtt{status}[X] = \mathtt{PENDING}$, then X can issue an `xend` to change $\mathtt{status}[X]$ to COMMITTED. Otherwise, $\mathtt{status}[X] = \mathtt{PENDING\_ABORT}$, and X can issue an `xabort` to change its status to ABORTED. For both `xend` and `xabort`, the OAT model removes X from $\mathtt{ready}(\mathcal{C})$ and adds $\mathtt{parent}[X]$ back into $\mathtt{ready}(\mathcal{C})$. The `xend` instruction also performs an ownership-aware commit and changes read sets and write sets, which we describe later in Section 8.5.

Finally, a ready node X issues a `read` and `write` instruction, if the instruction does not generate a conflict, it adds a memory operation node v to $\mathtt{memOps}(t,\mathcal{C})$, with v as a child of X. If the instruction would create a conflict, the runtime may change the status of one PENDING transaction T to PENDING_ABORT to make progress in resolving the conflict. For shorthand, we refer to the status change of a transaction T from PENDING to PENDING_ABORT as a `sigabort` of T.

This construction of the tree guarantees a few properties.

First, the sequence of instructions $\mathcal{S}$ generated by the OAT model is a valid topological sort of the computation dag $G(\mathcal{C})$. Second, the OAT model generates a tree of a canonical form, where the

root node of the tree is a transaction, all transactions are S-nodes and every P-node has exactly two nontransactional S-node children. This canonical form is imposed for convenience of description; it is not important for any theoretical results. Finally, the OAT model maintains the invariant the active nodes form a tree, with the ready nodes at the leaves. This property is important for the correctness of the OAT model.

Memory operations and conflict detection

The OAT model performs eager conflict detection; before performing a memory operation that would create a new $v \in \texttt{memOps}(\mathcal{C})$, the OAT model first checks whether creating v would cause a conflict, according to Definition 21.

Definition 21 *Suppose at time t, the OAT model issues a* `read` *or* `write` *instruction that potentially creates a memory operation node v. We say that v generates a* ***memory conflict*** *if there exists a location $\ell \in \mathcal{M}$ and an active transaction $T_u \in \texttt{activeX}^{(t)}(\mathcal{C})$ such that*

1. $T_u \notin \texttt{xAnces}(v)$, *and*
2. *either* $R(v,\ell) \wedge ((\ell,u) \in \texttt{W}(t,T_u))$, *or* $W(v,\ell) \wedge ((\ell,u) \in \texttt{R}(t,T_u))$.

If a potential memory operation v would generate a conflict, then the memory operation v does not occur; instead, a `sigabort` of some transaction may occur. We describe the mechanism for aborts in Section 8.5. Otherwise, v does not generate a conflict and observes the value ℓ from $\texttt{R}(Y)$, where Y is the closest ancestor of v with ℓ in its readset (i.e., $(\ell,u) \in \texttt{R}(Y)$ and $\Phi(v) = u$). The `read` also adds v to X's readset. A successful `write` operation v sets the observer function $\Phi(v)$ in the same way as a `read`. The `write` adds (ℓ, v) to both $\texttt{R}(X)$ and $\texttt{W}(X)$.

Ownership-aware transaction commit

The ***ownership-aware commit mechanism*** employed by the OAT model 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 read and write sets which are owned by MY in an open-nested fashion to the root of the tree, while it commits locations owned by other Xmodules in a closed-nested fashion, merging those reads and writes into X's read and write sets.

We can describe the OAT model's commit mechanism more formally in terms of module readsets and writesets. Suppose at time t, $Y \in \texttt{xactions}(t,\mathcal{C})$ with $\texttt{status}[Y] = \texttt{PENDING}$ issues an `xend`. This `xend` changes readsets and writesets as follows.

$$\begin{aligned}
\texttt{R}(\texttt{root}(\mathcal{C})) &\leftarrow \texttt{R}(\texttt{root}(\mathcal{C})) \cup \texttt{modR}(Y) \\
\texttt{R}(\texttt{xparent}[Y]) &\leftarrow \texttt{R}(\texttt{xparent}[Y]) \cup (\texttt{R}(Y) - \texttt{modR}(Y)) \\
\texttt{W}(\texttt{root}(\mathcal{C})) &\leftarrow \texttt{W}(\texttt{root}(\mathcal{C})) \cup \texttt{modW}(Y) \\
\texttt{W}(\texttt{xparent}[Y]) &\leftarrow \texttt{W}(\texttt{xparent}[Y]) \cup (\texttt{W}(Y) - \texttt{modW}(Y))
\end{aligned}$$

Unique committer property

Definition 20 guarantees certain properties of the computation tree which are essential to the ownership-aware commit mechanism. Theorem 8.5 proves that every memory operation has one and only one transaction that is responsible for committing the memory operation. The proof of the theorem requires the following lemma.

Lemma 8.4 *Given a computation tree $\mathcal{C}$, for any $T \in \texttt{xactions}(\mathcal{C})$, let $S_T = \{MT' : T' \in \texttt{xAnces}(T)\}$. Then $\texttt{modAnces}(MT) \subseteq S_T$.*

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 $T = \texttt{root}(\mathcal{C})$, and $M\texttt{root}(\mathcal{C}) = \texttt{world}$. Thus, the fact holds trivially. For the inductive step, assume that $\texttt{modAnces}(MT) \subseteq S_T$ holds for any transaction T at depth d. We show that the fact holds for any $T^* \in \texttt{xactions}(\mathcal{C})$ at depth $d+1$.

For any such T^*, we know $T = \texttt{xparent}[T^*]$ is at depth d. Then, by Rule 2 of Definition 20, we have $\texttt{modParent}(MT^*) \in \texttt{modAnces}(MT)$. Thus, $\texttt{modAnces}(MT^*) \subseteq \texttt{modAnces}(MT) \cup \{MT^*\}$. By construction of the set S_T, we have $S_{T^*} = S_T \cup \{MT^*\}$. Therefore, using the inductive hypothesis, we have $\texttt{modAnces}(MT^*) \subseteq S_{T^*}$. □

Theorem 8.5 *If a memory operation v accesses a memory location ℓ, then there exists a unique transaction $T^* \in \texttt{xAnces}(v)$, such that*

1. $\texttt{owner}(\ell) = MT^*$, *and*

2. *For all transactions $X \in \texttt{pAnces}(T^*) \cap \texttt{xactions}(\mathcal{C})$, X can not directly access location ℓ.*

*This transaction T^* is the **committer** of memory operation v, denoted $\texttt{committer}(v)$.*

PROOF. This result follows from the properties of the module tree and computation tree stated in Definition 20.

Let $T = \texttt{xparent}[v]$. First, by Definition 20, Rule 1, we know $\texttt{owner}(\ell) \in \texttt{modAnces}(MT)$. We know $\texttt{modAnces}(MT) \subseteq S_T$ by Lemma 8.4. Thus, there exists some transaction $T^* \in \texttt{xAnces}(T)$ such that $\texttt{owner}(\ell) = MT^*$. We can use Rule 2 to show that the T^* is unique. Let X_i be the chain of ancestor transactions of T, i.e., let $X_0 = T$, and let $X_i = \texttt{xparent}[X_{i-1}]$, up until $X_k = \texttt{root}(\mathcal{C})$. By Rule 2, we know $\texttt{xid}(MX_i) < \texttt{xid}(MX_{i-1})$, that is, the xids strictly decrease walking up the tree from T. Thus, there can only be one ancestor transaction T^* of T with $\texttt{xid}(MT^*) = \texttt{xid}(\texttt{owner}(\ell))$.

To check the second condition of Theorem 8.5, consider any $X \in \texttt{pAnces}(T^*) \cap \texttt{xactions}(\mathcal{C})$. By Rule 1, X can access ℓ directly only if $\texttt{owner}(\ell) \in \texttt{modAnces}(MX)$ implying that $\texttt{xid}(\texttt{owner}(\ell)) \leq \texttt{xid}(MX)$. But we know that $\texttt{owner}(\ell) = MT^*$ and $\texttt{xid}(MT^*) > \texttt{xid}(MX)$. □

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 8.5 states that no ancestor transaction of T^* in the call stack can ever directly access ℓ; thus, it is "safe" for T^* to commit ℓ.

Transaction abort

When the OAT model detects a conflict, it aborts one of the conflicting transactions by changing its status from PENDING to PENDING_ABORT. In the OAT model, a transaction X might not abort immediately; instead, it might continue to issue more instructions after its status has changed to PENDING_ABORT. Later, it will be useful to refer to the set of operations a transaction X issues while its status is PENDING_ABORT.

Definition 22 *The set of operations issued by X or descendants of X after* status$[X]$ *changes to* PENDING_ABORT *are called X's* ***abort actions****. This set is denoted by* abortactions(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 T_u and T_u's status is PENDING, then the OAT model can nondeterministically choose to abort either xparent$[v]$, or T_u. In the latter case, v waits for T_u to finish aborting (i.e., change its status to ABORTED) before continuing. If T_u's status is PENDING_ABORT, then v just waits for T_u to finish aborting before trying to issue read or write again.

This operational model uses the same conflict detection algorithm as TM with ordinary closed-nested transactions does; the only subtleties are that v can generate a conflict with a PENDING_ABORT transaction T_u, and that transactions no longer abort instantaneously because they have abort actions. Some restrictions on the abort actions of a transaction may be necessary to avoid deadlock, as we describe later in Section 8.7.

8.6 Serializability by Modules

In this section, we define ***serializability by modules***, a definition inspired by the database notion of multilevel serializability (e.g., as described in [Wei86]). First, we describe the definition of serializability in the transactional computation framework, as given in [ALS06]. Next, we incorporate Xmodules into this definition and define serializability by modules. We then prove that the OAT model guarantees serializability by modules. Finally, we discuss the relationship between the definition of serializability by modules, and the notion of abstract serializability for the methodology of open nesting.

Transactional computations and serializability

In Chapter 6, we defined serializability formally for TM systems with closed and open nesting. In this section, we extend that definition to ownership-aware transactions. In order to do so, we first define content sets more generally than in Chapter 6, and extend the definition of hidden vertices. Once we have done so, the definitions of serializability and transactional serializabilty automatically generalize. Informally, a trace $(\mathcal{C}, \Phi)$ is serializable if there exists a topological sort order $\mathcal{S}$ of $G(\mathcal{C})$ such that $\mathcal{S}$ is "sequentially consistent with respect to Φ", and all transactions appear contiguous in the order $\mathcal{S}$.

Content sets

We first describe some notation needed to formally describe serializability by modules. All definitions in this section are *a posteriori*, i.e., they are defined on the computation tree after the program has finished executing.

We define "content" sets for every transaction T by partitioning $\texttt{memOps}(T)$ (all the memory operations enclosed inside T including those belonging to its nested transactions) into three sets: $\texttt{cContent}(T)$, $\texttt{oContent}(T)$ and $\texttt{aContent}(T)$. For any $u \in \texttt{memOps}(T)$, we define the content sets based on the final status of transactions in C that one visits when walking up the tree from u to T.

Definition 23 *For any transaction T and memory operation u, define the sets* $\texttt{cContent}(T)$, $\texttt{oContent}(T)$, *and* $\texttt{aContent}(T)$ *according the* $\texttt{ContentType}(u, T)$ *procedure:*

```
    ContentType(u, T)          ▷ For any u ∈ memOps(T)
1   X ← xparent[u]
2   while (X ≠ T)
3     if (X is ABORTED)        return u ∈ aContent(T)
4     if (X = committer(u))    return u ∈ oContent(T)
5     X ← xparent[X]
6   return u ∈ cContent(T)
```

Recall that in the OAT model, the safe-nested commit of T commits some memory operations in an open-nested fashion, to $\texttt{root}(C)$, and some operations in a closed-nested fashion, to $\texttt{xparent}[T]$. Informally, $\texttt{oContent}(T)$ is the set of memory operations that are committed in an "open" manner by T's subtransactions. Similarly, $\texttt{aContent}(T)$ is the set of operations that are discarded due to the abort of some subtransaction in T's subtree. Finally, $\texttt{cContent}(T)$ is the set of operations that are neither committed in an "open" manner, nor aborted.

Sequential consistency with transactions

For computations with transactions, we can modify the classic notion of sequential consistency to account for transactions which abort. Transactional semantics dictate that memory operations belonging to an aborted transaction T should not be observed by (i.e., are ***hidden*** from) memory operations outside of T.

Definition 24 *For $u \in \texttt{memOps}(C), v \in V(C)$, let $X = \texttt{xLCA}(u, v)$. We say that u is **hidden** from v if $u \in \texttt{aContent}(X)$.*

Our definition of serializability by modules requires that computations satisfy some notion of sequential consistency, generalized for the setting of TM. Here is the definition of transactional last writer function from Chapter 6.

Definition 25 *Consider a trace (C, Φ) and a topological sort S of $G(C)$. For all $v \in \texttt{memOps}(C)$ such that $R(v, \ell) \vee W(v, \ell)$, the **transactional last writer** of v according to S, denoted $\mathcal{X}_S(v)$, is the unique $u \in \texttt{memOps}(C) \cup \{\perp\}$ that satisfies four conditions:*

1. $W(u, \ell)$,

2. $u <_{\mathcal{S}} v$,

3. $\neg(uHv)$, *and*

4. $\forall w\,(W(w,\ell) \wedge (u <_{\mathcal{S}} w <_{\mathcal{S}} v)) \implies wHv$.

Definition 26 *A trace $(\mathcal{C}, \Phi)$ is **sequentially consistent** if there exists a topological sort $\mathcal{S}$ such that $\Phi = \mathcal{X}_{\mathcal{S}}$. We say that $\mathcal{S}$ is **sequentially consistent with respect to** Φ.*

In other words, the transactional last writer of a memory operation u which accesses location ℓ, is the last write v to location ℓ in the order $\mathcal{S}$, except we skip over writes w which are hidden from (i.e., aborted with respect to) u. Intuitively, Definition 12 requires that there exists an order $\mathcal{S}$ explaining all the memory operations of the computation.

Serializability

Definition 27 *A trace $(\mathcal{C}, \Phi)$ is **serializable** if there exists a topological sort $\mathcal{S}$ that satisfies two conditions:*

1. $\Phi = \mathcal{X}_{\mathcal{S}}$ *($\mathcal{S}$ is sequentially consistent with respect to Φ), and*

2. $\forall T \in \texttt{xactions}(\mathcal{C})$ *and* $\forall v \in V(\mathcal{C})$, *we have* $\texttt{xbegin}(T) \leq_{\mathcal{S}} v \leq_{\mathcal{S}} \texttt{xend}(T)$ *implies* $v \in V(T)$).

Ordinary serializability can be thought of as a strengthening of sequential consistency which also requires that the order $\mathcal{S}$ both explains all memory operations, and also has all transactions appearing contiguous.

Defining serializability by modules

In Chapter 6, a trace $(\mathcal{C}, \Phi)$ was said to be ***serializable*** if there exists a topological sort $\mathcal{S}$ of $G(\mathcal{C})$ such that $\mathcal{S}$ is sequentially consistent with respect to Φ, and all transactions appear contiguous in $\mathcal{S}$. Serializability in this context can be thought of as a sequential consistency plus the requirement that transactions are atomic. This definition of serializability is the "correct definition" for flat or closed-nested transactions. This definition of serializability is too strong, however, for ownership-aware transactions. A TM system that enforces this definition of serializability can not ignore lower-level memory accesses when detecting conflicts for higher-level transactions.

Instead, we describe a definition of serializability by modules which checks for correctness of one Xmodule at a time. Given a trace $(\mathcal{C}, \Phi)$, for each Xmodule A, we transform the tree $\mathcal{C}$ into a new tree $\texttt{mTree}(\mathcal{C}, A)$. The tree $\texttt{mTree}(\mathcal{C}, A)$ is constructed in such a way as to ignore memory operations of Xmodules which are lower-level than A, and also to ignore all operations which are hidden from transactions of A. For each Xmodule A, we check that the transactions of A in the trace $(\texttt{mTree}(\mathcal{C}, A), \Phi)$ is serializable. If the check holds for all Xmodules, then trace $(\mathcal{C}, \Phi)$ is said to be serializable by modules.

Definition 28 formalizes the construction of $\texttt{mTree}(\mathcal{C}, A)$.

Definition 28 *For any computation tree C, let* $\texttt{mTree}(C, A)$ *be the result of modifying C as follows:*

1. *For all memory operations $u \in \texttt{memOps}(C)$ with u accessing ℓ, if $\texttt{owner}(\ell) = B$ for some $\texttt{xid}(B) > \texttt{xid}(A)$, convert u into a nop.*

2. *For all transactions $T \in \texttt{modXactions}(A)$, convert all $u \in \texttt{aContent}(T)$ into nops.*

The intuition behind Condition 1 of Definition 28 is the following. When looking at Xmodule A, we throw away memory operations belonging to a lower-level Xmodule B, since by Theorem 8.5, transactions of A can never directly access the same memory as those operations anyway. In Condition 2, we ignore the content of any aborted transactions nested inside transactions of A; 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 A.

Lemma 8.6 argues that if a trace (C, Φ) is sequentially consistent, then $(\texttt{mTree}(C, A), \Phi)$ is a valid trace; an 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 28. In addition, the transformed trace is also sequentially consistent.

Lemma 8.6 *Let (C, Φ) be any sequentially consistent trace. Then for any Xmodule A, $(\texttt{mTree}(C, A), \Phi)$ is a valid trace. In other words, if $u \in \texttt{memOps}(\texttt{mTree}(C, A))$, then $\Phi(u) \in \texttt{memOps}(\texttt{mTree}(C, A))$. Furthermore, any S which is sequentially consistent with respect to Φ in (C, Φ) is also sequentially consistent with respect to Φ in $(\texttt{mTree}(C, A), \Phi)$.*

PROOF. In the new tree $\texttt{mTree}(C, A)$, pick any $u \in \texttt{memOps}(\texttt{mTree}(C, A))$ 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 28, then we know because v accessed a memory location ℓ where $\texttt{xid}(\texttt{owner}(\ell)) > \texttt{xid}(A)$. 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 28, then $v \in \texttt{aContent}(T)$ for some $MT = A$. 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 X is an ancestor of T or T is a proper ancestor of X.

1. First, suppose T is a proper ancestor of X. Consider the path of transactions $Y_0, Y_1, \ldots Y_k$, where $Y_0 = \texttt{xparent}[v]$, $\texttt{xparent}[Y_i] = Y_{i+1}$, and $\texttt{xparent}[Y_k] = T$. Since $v \in \texttt{aContent}(T)$, for some Y_j for $0 \leq j \leq k$ must have $\texttt{status}[Y_j] = \texttt{ABORTED}$. Since T is a proper ancestor of X, $X = Y_x$ for some x satisfying $0 \leq x \leq k$.

   (a) If $\texttt{status}[Y_j] = \texttt{ABORTED}$ for any j satisfying $0 \leq j < x$, then we know $v \in \texttt{aContent}(X)$, and thus vHu. Since we assumed (C, Φ) is sequentially consistent and $\Phi(u) = v$, by Definition 25, we know $\neg vHu$, leading to a contradiction.

   (b) If Y_j is ABORTED for any j satisfying $x \leq j \leq k$, then $\texttt{status}[Y_j] = \texttt{ABORTED}$ implies that $u \in \texttt{aContent}(X)$, 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.

Finally, if Φ is the transactional last writer according to $\mathcal{S}$ for $(\mathcal{C}, \Phi)$, it is still the transactional last writer for $(\texttt{mTree}(\mathcal{C}, A), \Phi)$ because the memory operations which are not turned into nops remain in the same relative order. Thus, the last condition is satisfied. □

Note that Lemma 8.6 *depends on* the restrictions on Xmodules described in Definition 20. Without this structure of modules and ownership, the construction of Definition 28 is not guaranteed to generate a valid trace.

Finally, we can define serializability by modules.

Definition 29 *A trace $(\mathcal{C}, \Phi)$ is **serializable by modules** if it is sequentially consistent, and if for all Xmodules A in $\mathcal{D}$, there exists a topological sort $\mathcal{S}$ of $\mathcal{C}_A = \texttt{mTree}(\mathcal{C}, A)$ such that:*

1. *$\mathcal{S}$ is sequentially consistent with respect to Φ, and*
2. *For the tree $\mathcal{C}_A$, $\forall T \in \texttt{modXactions}(A)$ and $\forall v \in V(\mathcal{C}_A)$, if we have $\texttt{xbegin}(T) \leq_{\mathcal{S}} v \leq_{\mathcal{S}} \texttt{xend}(T)$, then $v \in V(T)$.*

Informally, a trace $(\mathcal{C}, \Phi)$ is serializable by modules if it is sequentially consistent, and if for every Xmodule A, there exists a sequentially consistent order $\mathcal{S}$ for the trace $(\texttt{mTree}(\mathcal{C}, A), \Phi)$ such that all transactions of A are contiguous in $\mathcal{S}$.

OAT model guarantees serializability by modules

In this section, we show that the OAT model described in Section 8.5 generates traces $(\mathcal{C}, \Phi)$ that are serializable by modules, i.e., that satisfy Definition 29. The proof of this fact consists of three steps. First, we generalize the notion of "prefix race-freedom" described in [ALS06], to computations with Xmodules. Second, we prove that the OAT model guarantees that a program execution is prefix race-free. Finally, we argue that any trace which is prefix race-free is also serializable by modules.

Defining prefix race-freedom

First, we define prefix races. These definitions are essentially the same as those in [ALS06], except adapted for a system with an ownership-aware commit mechanism instead of an open-nested commit mechanism.

Definition 30 *For any execution order $\mathcal{S}$, for any transaction $T \in \texttt{xactions}(\mathcal{C})$, consider any $v \notin \texttt{memOps}(T)$ such that $\texttt{xbegin}(T) <_{\mathcal{S}} v <_{\mathcal{S}} \texttt{xend}(T)$. We say there exists a **prefix race** between T and v if there exists a memory operation $w \in \texttt{cContent}(T)$ s.t., $w <_{\mathcal{S}} v$, $\neg(vHw)$, v and w both access ℓ, and one of v, w writes to ℓ.*

Definition 31 *A trace $(\mathcal{C}, \Phi)$ is **prefix race-free** iff exists a topological sort $\mathcal{S}$ of $G(\mathcal{C})$ satisfying two conditions:*

1. *$\Phi = \mathcal{X}_{\mathcal{S}}$ ($\mathcal{S}$ is sequentially consistent with respect to Φ), and*
2. *$\forall v \in V(\mathcal{C})$ and $\forall T \in \texttt{xactions}(\mathcal{C})$ there is no prefix race between v and T.*

*$\mathcal{S}$ is called a **prefix race-free sort** of the trace.*

Properties of the OAT model

Second, we prove several invariants that OAT model preserves, and then use these invariants to prove that the OAT model generates only traces $(\mathcal{C}, \Phi)$ which are prefix race-free.

The sequence of instructions that the OAT model issues naturally generates a topological sort $\mathcal{S}$ of the computation dag $G(\mathcal{C})$: the `fork` and `xbegin` instructions correspond to the begin nodes of a parallel or series blocks in the dag, the `join`, `xend`, and `xabort` instructions correspond to end nodes of parallel or series blocks, and the `read` or `write` instructions correspond to memory operation nodes $v \in \texttt{memOps}(\mathcal{C})$.

Theorem 8.7 *Suppose the OAT model generates a trace $(\mathcal{C}, \Phi)$ and an execution order $\mathcal{S}$. Then, $\Phi = \mathcal{X}_{\mathcal{S}}$, i.e., $\mathcal{S}$ is sequentially consistent with respect to Φ.*

PROOF. This result is reasonably intuitive, but the proof is tedious and somewhat complicated. We defer the details of this proof to Appendix .2. □

Next, we describe an invariant on readsets and writesets that the OAT model maintains. Informally, Lemma 8.8 states that, if a memory operation u that reads (writes) location ℓ is in the `cContent`(T) for some transaction T, then ℓ belongs to the read set (write set) of some active transaction under T's subtree between the time when the memory operation is performed and the time when T ends.

Lemma 8.8 *Suppose the OAT model generates a trace $(\mathcal{C}, \Phi)$ with an execution order $\mathcal{S}$. For any transaction T, consider a memory operation $u \in \texttt{cContent}(T)$ which accesses memory location ℓ at step t_0. Let t_f be step when $\texttt{xend}(T)$ or $\texttt{xabort}(T)$ happens. At any time t such that $t_0 \leq t < t_f$ there exists some $T' \in \texttt{xDesc}(T) \cap \texttt{activeX}^{(t)}(\mathcal{C})$ (i.e., T' is an active transactional descendant of T) such that*

1. *If $R(u, \ell)$, then $\ell \in \texttt{R}(t, T')$.*
2. *If $W(u, \ell)$, then $\ell \in \texttt{W}(t, T')$.*

PROOF. Let $X_1, X_2, \ldots X_k$ be the chain of transactions from $\texttt{xparent}[u]$ up to, but not including T, i.e., $X_1 = \texttt{xparent}[u]$, $X_j = \texttt{xparent}[X_{j-1}]$, and $\texttt{xparent}[X_k] = T$. Since we assume that $u \in \texttt{cContent}(T)$ and since T completes at time t_f, for every j such that $1 \leq j < k$, there exists a unique time t_j (satisfying $t_0 \leq t_j < t_f$) when an `xend` changes $\texttt{status}[X_j]$ from `PENDING` to `COMMITTED`; otherwise, we would have $u \in \texttt{aContent}(T)$.

Also, by Theorem 8.5 and Definition 23, we know $\texttt{committer}(u) \in \texttt{xAnces}(T)$, i.e., none of the X_j's will commit location ℓ in an open-nested fashion to the world; otherwise, we would have $u \in \texttt{oContent}(T)$.

First, suppose $R(u, \ell)$. At time t_i, when the memory operation u completes, (ℓ, u) is added to $\texttt{R}(X_1)$. In general, at time t_j, the ownership-aware commit mechanism, as described in Section 8.5, will propagate ℓ from $\texttt{R}(X_j)$ to $\texttt{R}(X_{j+1})$. Therefore, for any time t in the interval $[t_{j-1}, t_j)$, we know $\ell \in \texttt{R}(t, X_j)$, i.e., for Lemma 8.8, $T' = X_j$. Similarly, for any time t in the interval $[t_k, t_f)$, we have $\ell \in \texttt{R}(t, T)$, i.e., we choose $T' = T$.

The case where $W(u, \ell)$ is completely analogous to the case of $R(u, \ell)$, except we have both $\ell \in \texttt{R}(t, T')$ and $\ell \in \texttt{W}(t, T')$. □

We use Theorem 8.7 and Lemma 8.8 to prove that the OAT model generates traces which are prefix race-free.

Theorem 8.9 *Suppose the OAT model generates a trace $(\mathcal{C}, \Phi)$ with an execution order $\mathcal{S}$. Then $\mathcal{S}$ is an prefix race-free sort of $(\mathcal{C}, \Phi)$.*

PROOF.

For the first condition of Definition 31, we know by Theorem 8.7 that the OAT model generates an order $\mathcal{S}$ which is sequentially consistent with respect to Φ.

To check the second condition, assume for contradiction that we have an order $\mathcal{S}$ generated by the OAT model, but there exists a prefix race between a transaction T and a memory operation $v \notin \texttt{memOps}(T)$. Let w be the memory operation from Definition 30, i.e., $w \in \texttt{cContent}(T)$, $w <_{\mathcal{S}} v <_{\mathcal{S}} \texttt{xend}T$, $\neg(vHw)$, w and v access the same location ℓ, with one of the accesses being a write. Let t_w and t_v be the time steps in which operations w and v occurred, respectively, and let t_{endT} be the time at which either $\texttt{xend}(T)$ or $\texttt{xabort}(T)$ occurs (i.e., either T commits or aborts). We argue that at time 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 $W(v, \ell)$ and $R(w, \ell)$. Since $t_w < t_v < t_{endT}$, by Lemma 8.8, at time t_v, ℓ is in the writeset of some active transaction $T' \in \texttt{desc}(T)$. Since $v \notin \texttt{memOps}(T)$, we know $T \notin \texttt{ances}(v)$. Thus, since T' is a descendant of T, we have $T' \notin \texttt{ances}(v)$. Since $T' \notin \texttt{ances}(v)$, by Definition 21, at time t_v, v generates a conflict with T'. The other two cases, where $R(v, \ell) \wedge W(w, \ell)$ or $W(v, \ell) \wedge W(w, \ell)$, are analogous.

□

Prefix race-freedom implies serializability by modules

Finally, we show that a trace $(\mathcal{C}, \Phi)$ which is prefix race-free is also serializable by modules.

Theorem 8.10 *Any trace $(\mathcal{C}, \Phi)$ which is prefix race-free is also serializable by modules.*

PROOF.

First, by Definition 28 and Lemma 8.6, it is easy to see that a prefix-race free sort $\mathcal{S}$ of a trace $(\mathcal{C}, \Phi)$ is also prefix-race free of the sort $(\texttt{mTree}(\mathcal{C}, A), \Phi)$ for any Xmodule A. Now we shall argue that for any Xmodule A, we can transform $\mathcal{S}$ into $\mathcal{S}_A$ such that all transactions in $\texttt{xactions}(A)$ appear contiguous in $\mathcal{S}_A$.

Consider a prefix-race free sort $\mathcal{S}$ of $(\texttt{mTree}(\mathcal{C}, A), \Phi)$ which has k nodes v which violate the second condition of Definition 29. We show how to construct a new order $\mathcal{S}'$ which is still a prefix race-free sort of $(\texttt{mTree}(\mathcal{C}, A), \Phi)$, but which has only $k - 1$ violations.

We reduce the number of violations according to the following procedure:

1. Of all transactions $T \in \texttt{modXactions}(A)$ such that there exists an operation v such that $\texttt{xbegin}(T) \leq_{\mathcal{S}} v \leq_{\mathcal{S}} \texttt{xend}(T)$ and $v \notin V(T)$, choose the $T = T^*$ which has the latest $\texttt{xend}(T)$ in the order $\mathcal{S}$.

2. In T^*, pick the first $v \notin V(T^*)$ which causes a violation.

3. Create a new sort $\mathcal{S}'$ by moving v to be immediately before $\texttt{xbegin}(T^*)$.

In order to argue that $\mathcal{S}'$ is still a prefix race-free sort of $(\texttt{mTree}(\mathcal{C}, A), \Phi)$, we need to show that moving v does not generate any new prefix races, and does not 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, an $\texttt{xbegin}(T')$, or an $\texttt{xend}(T')$.

1. Suppose v is a memory operation which accesses location ℓ. For all operations w such that $\texttt{xbegin}(T) <_{\mathcal{S}} w <_{\mathcal{S}} v$, we argue that w can not access the same location ℓ unless both w and v read from ℓ. Since we chose v to be the first memory operation such that $\texttt{xbegin}(T) <_{\mathcal{S}} v <_{\mathcal{S}} \texttt{xend}(T)$ such that $v \notin V(T)$, we know $w \in V(T)$. We know by construction of $\texttt{mTree}(\mathcal{C}, A)$, that $w \in \texttt{cContent}(T)$ (if $w \in \texttt{oContent}(T)$ or $w \in \texttt{aContent}(T)$, then steps 1 or 2, respectively, in Definition 28 will turn w into a nop). Therefore, by Definition 30, 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 $\texttt{xbegin}(T)$ can not generate any new prefix races or change the transactional last writer for any memory operation, and $\mathcal{S}'$ is still a prefix race-free sort of the trace.

2. Next, suppose $v = \texttt{xbegin}(T')$. Moving $\texttt{xbegin}(T')$ can not generate any new prefix races with T', because the only memory operations u which satisfy $\texttt{xbegin}(T) <_{\mathcal{S}} u <_{\mathcal{S}} \texttt{xbegin}(T')$ satisfy $u \notin \texttt{cContent}(T')$. Also, moving $\texttt{xbegin}(T')$ 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 $v = \texttt{xend}(T')$. By moving $\texttt{xend}(T')$ to be before $\texttt{xbegin}(T)$, we can only lose prefix races with T' that already existed in $\mathcal{S}$ because we are moving nodes out of the interval $[\texttt{xbegin}(T'), \texttt{xend}(T')]$. Also, as with $\texttt{xbegin}(T')$, moving $\texttt{xend}(T')$ 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 29 one at a time, we can construct a sort $\mathcal{S}_A$ which satisfies serializability by modules by eliminating all violations. □

Finally, we can prove the OAT model guarantees serializability by modules by putting the previous results together.

Theorem 8.11 *Any trace $(\mathcal{C}, \Phi)$ generated by the OAT model is serializable by modules.*

PROOF. By Theorem 8.9, the OAT model generates only trace $(\mathcal{C}, \Phi)$ which are prefix race-free. By Theorem 8.6, any trace $(\mathcal{C}, \Phi)$ which is prefix race-free is serializable by modules. □

Abstract serializability

By Theorem 8.11, the OAT model guarantees serializability by modules. We now relate this definition to the notion of ***abstract serializability*** used in multilevel database systems [Wei86]. As we mentioned in Section 8.1, the ownership-aware commit mechanism is a part of a methodology which includes abstract locks and compensating actions. In this section we argue 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 [Wei86] assumes that the program is divided into levels, and that a transaction at level i can only call a transaction at level $i + 1$.[9] In addition, transactions at a particular level have predefined commutativity rules, i.e., some transactions of the

[9]We assume level number increases as you go from a higher level to a lower-level to be consistent with our numbering of `xid`. In the literature (e.g. [Wei86]), levels typically go in the opposite direction.

same Xmodule can commute with each other and some can not. 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 [NMAT+07]: if two transactions can not commute, then they grab the same abstract lock in a conflicting manner. In the application described in Section 8.2, 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.[10] Locks associated with an Xmodule A are owned by $\texttt{modParent}(A)$. A module A is said to be ***properly locked*** if the following is true for all transactions T_1, T_2 with $MT_1 = MT_2 = A$: if T_1 and T_2 do not commute, then they access some $\ell \in \texttt{modMemory}(\texttt{modParent}(A))$ 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 non-leaf module has exactly one child). Let $\mathcal{S}_i$ be the sort $\mathcal{S}$ in Definition 29 for Xmodule A with $\texttt{xid}(A) = i$. 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 19, we know 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 have to 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 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.

8.7 Deadlock Freedom

In this section, we argue that the OAT model described in Section 8.5 can never enter a "semantic deadlock" if we impose suitable restrictions on the memory accessed by a transaction's abort actions. In particular, an abort action generated by transaction T from MT should read (write) from a memory location ℓ belonging to $\texttt{modAnces}(MT)$ only if ℓ is already in $\texttt{R}(T)$ ($\texttt{W}(T)$). Under these conditions, we 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; a scheduler in the TM runtime can abort all transactions, and then complete

[10] More complicated locks can be modeled by generalizing the definition of conflict.

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., when the status of T_2 becomes 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 want to 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 for noncompletion.

Recall that our abstract model has 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 $\mathcal{F}$ can not complete the program no matter what it does. To eliminate programs which have infinite loops, we only consider ***bounded programs***.

Definition 32 *We say that a program is **bounded** for an operational model $\mathcal{F}$ if any computation tree that $\mathcal{F}$ generates for that program is of a finite depth, and there exists a finite number K such that at any time t, every node $B \in \texttt{nodes}(t, \mathcal{C})$ 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 33 *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 time t, $\mathcal{F}$ nondeterministically chooses which ready node $X \in \texttt{ready}(\mathcal{C})$ 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 34 *Consider an $\mathcal{F}$ executing a bounded computation. We say that $\mathcal{F}$ does not exhibit a **semantic deadlock** if for all finite sequences of t_0 instructions that $\mathcal{F}$ can issue that generates some intermediate computation tree $\mathcal{C}_0$, there exists a finite schedule Γ on $[t_0, t_1]$ such that $\mathcal{F}$ brings the computation tree to a rest state $\mathcal{C}_1$, i.e., $\texttt{ready}(\mathcal{C}_1) = \{\texttt{root}(\mathcal{C}_1)\}$.*

This definition is sufficient, since once the computation tree is at the rest state, and only the root node is ready, $\mathcal{F}$ can execute each transaction serially and complete the computation.

Restrictions to avoid semantic deadlock

The general OAT model described in Section 8.5 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, where a PENDING_ABORT transaction T never accesses new memory belonging to Xmodules at MT's level or higher, however, we can show the OAT model is free of semantic deadlock.

More formally, for all transactions T, we restrict the memory footprint of $\texttt{abortactions}(T)$.

Definition 35 *An execution (represented by a computation tree $\mathcal{C}$) has **abort actions with limited footprint** if the following condition is true for all transactions $T \in \texttt{aborted}(\mathcal{C})$. At time t, if a memory operation $v \in \texttt{abortactions}(T)$ accesses location ℓ and $\texttt{owner}(\ell) \in \texttt{modAnces}(MT)$, then (1) if v is a read, then $\ell \in \texttt{R}(T)$, and (2) if v is a write then $\ell \in \texttt{W}(T)$.*

Intuitively, Definition 35 requires that once a transaction T's status becomes PENDING_ABORT, any memory operation v which T or a nested transaction inside T performs to finish aborting T can not read from (write to) any location ℓ which is owned by any Xmodules which are ancestors of MT (including MT itself), unless ℓ is already in the read (or write set) of T.

First, we show that the properties of Xmodules from Theorem 8.5 in combination with the ownership-aware commit mechanism imply that transaction read sets and write sets exhibit nice properties. In particular, we have Corollary 8.12, which states that a location ℓ can appear in the read set of a transaction T only if T's Xmodule is a descendant of $\texttt{owner}(\ell)$ in the module tree $\mathcal{D}$.

Corollary 8.12 *For any transaction T if $\ell \in \texttt{R}(T)$, then $MT \in \texttt{modDesc}(\texttt{owner}(\ell))$.*

PROOF. Follows from Definition 19 and Theorem 8.5, and induction on how a location ℓ can propagate into readsets and writsets using the ownership-aware commit mechanism. □

If all abort actions have a limited footprint, we can show that operations of an abort action of an Xmodule A can only generate conflicts with a "lower-level" Xmodule.

Lemma 8.13 *Suppose the OAT model generates an execution where 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', then $\texttt{xid}(MT') > \texttt{xid}(MT)$.*

PROOF. Suppose $v \in \texttt{abortactions}(T)$ accesses a memory location ℓ with $\texttt{owner}(\ell) = A$. Since $\texttt{abortactions}(T) \subseteq \texttt{memOps}(T)$, by the properties of Xmodules given in Definition 20, we know that either $A \in \texttt{modAnces}(MT)$, or $\texttt{xid}(A) > \texttt{xid}(MT)$. If $A \in \texttt{modAnces}(MT)$, then by Definition 35, T already had ℓ in its read or write set. Therefore, using Definition 21, v can not 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 $\texttt{xid}(A) > \texttt{xid}(MT)$. If v conflicts with some other transaction T', then T' has ℓ in its read or write set. Therefore, from Corollary 8.12, $MT' \in \texttt{modDesc}(A)$. Thus, we have $\texttt{xid}(MT') > \texttt{xid}(A) > \texttt{xid}(MT)$. □

Theorem 8.14 *In the case where aborted actions have limited footprint, the OAT model is free from semantic deadlock.*

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 `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 instances T with $\texttt{xid}(MT) > i$, i.e., no instances T from Xmodules with `xid` larger than i. During phase i, we finish aborting all active transaction instances T with $\texttt{xid}(MT) = i$. By Lemma 8.13, any abort action for a T, where $\texttt{xid}(MT) = i$, can only conflict with a transaction instance T' from a lower-level Xmodule, where $\texttt{xid}(MT') > i$. Since the schedule Γ executes serially, and since by the inductive hypothesis 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 transaction X and X aborts, typically abort actions of X simply consists of 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 MX or its ancestor Xmodules unless the memory location is already in X's read (write) set. Assuming locks are modeled as accesses to memory locations, the same restriction applies, meaning, a compensating action can not acquire new locks that were not already acquired by the transaction it is compensating for.

8.8 Related Work

In this chapter, we described ownership-aware transactions, which provide a disciplined methodology for open nesting while guaranteeing abstract serializability. In this section, we describe two other approaches for improving open-nested transactions, and distinguish them from ownership-aware transactions.

In [NMAT+07], Ni, et al. 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 can not 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 in [HK08] describe a technique of transactional boosting which allows transactions to call methods from a nontransactional module A. Roughly, as long as A 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 A writes to memory owned by the enclosing higher-level module, or when programs have more than two levels of modules.

Chapter 9

Nested Parallelism within Transactions

Most TM implementations do not allow parallelism inside transactions. Therefore, a function containing parallelism (and transactions to synchronize between its parallel threads) cannot be called from within another transaction. This limitation manifests itself in two ways. First, it breaks abstraction by disallowing certain method calls purely on the basis of their implementation. Second, in programs using a dynamic-multithreaded language such as MIT Cilk, it is unnatural to write code with no parallelism inside transactions; adding transactions to these languages in a natural manner generates code with parallelism inside transactions. This chapter describes the first provably efficient software transactional memory system that allows parallelism inside transactions, specifically for languages that use Cilk-like work-stealing schedulers. Although this work is primarily theoretical, it provides a basis for concurrency platforms that allow transactions in dynamic multithreaded languages.

The remainder of this chapter is organized as follows. Section 9.1 provides the motivation of this work and the results. We use a computation tree described in Chapter 6 to model a computation with nested parallel transactions; Section 9.2 describes the computation tree and how the CWSTM runtime system maintains this computation tree online. Section 9.3 defines our CWSTM semantics. We show in Section 9.4 that a naive conflict-detection algorithm has poor worst-case performance. Section 9.5 describes the high-level design of CWSTM and its use of XConflict for conflict-detection. Section 9.6 gives an overview of the XConflict algorithm. Sections 9.7 and 9.10 provide details on data structures used by XConflict. Finally, Section 9.11 claims that XConflict, and hence CWSTM, is efficient for programs that experience no conflicts or contention.

9.1 Motivation and Results

Most work on transactional memory focuses exclusively on supporting transactions in programs that use persistent threads (e.g., pthreads). TM systems for such an environment are designed assuming that transactions execute serially, since the overhead of creating or destroying a pthread naturally discourages programmers from having nested parallelism inside a transaction. Furthermore, the special case of serial transactions greatly simplifies conflict detection for TM. Typically, the TM runtime detects a ***conflict*** between two distinct active transactions if they both access the same object ℓ, at least one transaction tries to write to ℓ, and both transactions are executed on

This is joint work with Jeremy Fineman and Jim Sukha [AFS08].

```
PARALLELINCREMENT()
1  x ← 0
2  parallel
3     { x ← x + 1                }
4     { x ← x + 10
5       parallel
6             { x ← x + 100  }
7             { x ← x + 1000 }   }
8  print x
```

Figure 9.1: A simple fork-join program that does several parallel increment of a shared variable. The **parallel** statement, similar to Dijkstra's "cobegin," allows the two following code blocks (each contained in $\{.\}$) to run in parallel. The subsequent line (line 8) executes after *both* parallel blocks complete. This program contains several races—assuming sequential consistency, valid outputs are $x \in \{1, 11, 101, 110, 111, 1001, 1010, 1011, 1101, 1110, 1111\}$.

Figure 9.2: The series-parallel dag for the sample program given in Figure 9.1. Edges correspond to instructions in the program. Diamonds and squares correspond to the start and end, respectively, of parallel constructs.

different threads. This last condition is relevant for TM that supports ***nested transactions***. Conceptually, when transactions execute serially, two active transactions executing on the same thread are allowed to access the same object because one must be nested inside the other.

In dyanamic multithreaded languages such as Cilk [BJK+96, Sup06] or NESL [BG96] or multithreaded libraries like Hood [BP99], a programmer specifies dynamic parallelism using linguistic constructs such as "fork" and "join," "spawn" and "sync," or "parallel" blocks. Dynamic multithreaded languages allow the programmer to specify a program's parallel structure abstractly, that is, permitting (but not requiring) parallelism in specific locations of the program. A runtime system (e.g., for Cilk) dynamically schedules the program on the number of processors, P, specified at execution time. If the language also permits only "properly nested" parallelism, i.e., any program execution can be represented as a "series-parallel dag" or "series-parallel parse tree" [FL97], then a Cilk-like work-stealing scheduler executes the program in a provably efficient manner [BJK+96]. A natural question arises: how can transactions be added to a dynamic multithreaded language such as Cilk?

To pose the problem more concretely, consider the series-parallel program shown in Figure 9.1, which performs parallel increments to a shared variable. Figure 9.2 gives the corresponding series-parallel dag for the program. One natural way to add transactions to a series-parallel program is by wrapping segments of the program in atomic blocks, as illustrated by Figure 9.3. As shown, it is easy to generate transactions (e.g., X_3) with nested parallelism and nested transactions (e.g., X_4). How does a TM system execute the program in Figure 9.3?

This chapter describes a way of adding transactions to a dynamic multithreaded language that generates only series-parallel programs. We focus on a provably efficient TM system that supports

```
XPARALLELINCREMENT()
 1  atomic {                                              ▷ Transaction X1
 2     x ← 0
 3     parallel
 4          { atomic {x ← x + 1}                  }       ▷ X2
 5          { atomic {                                    ▷ X3
 6               x ← x + 10
 7               parallel
 8                 { x ← x + 100                 }
 9                 { atomic {x ← x + 1000}       }       ▷ X4
10            }                                  }
11  }
12  print x
```

Figure 9.3: The program from Figure 9.1 with the addition of some transactions, denoted by **atomic**{.} blocks. The triangle denotes a comment. 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.

unbounded nesting and parallelism. That is, we want a TM system with a bound on a program's completion time that is independent of the maximum nesting depth of transactions. It turns out that TMs that perform work on every transaction commit proportional to the size of the transaction's "readset" or "writeset" cannot support an unbounded nesting depth efficiently. Generally, TM with lazy conflict detection requires work proportional to the size of the transactions readset, and TM with lazy updates require work proportional to the size of the transaction's writeset. Thus, we focus on TM with eager conflict detection and eager updates, since both require only a constant amount of work on every commit.

A key component of a TM system with nested parallelism is the conflict-detection scheme. We describe the semantics for TM with eager conflict detection for series-parallel computations with transactions. We present XConflict, a data structure that a software TM system can use to query for conflicts when implementing these semantics. For Cilk-like work-stealing schedulers, the XConflict answers concurrent queries in $O(1)$ time and can be maintained efficiently. In particular, consider a program with T_1 work and a span (or critical-path length) of T_∞, The running time on P processors of the program augmented with XConflict is only $O(T_1/P + PT_\infty)$. In comparison, with high probability, Cilk executes the program without XConflict in the asymptotically optimal time $O(T_1/P + T_\infty)$. These two bounds imply that maintaining the XConflict data structure does not asymptotically increase the running time of the program, compared to Cilk, when $\sqrt{T_1/T_\infty} \gg P$.

We describe CWSTM, a design for a software TM system with eager updates that uses the XConflict data structure. CWSTM provides strong atomicity and supports lazy cleanup on aborts (i.e., when a transaction X aborts, other transactions can help roll back the objects modified by X). The XConflict bounds translate to CWSTM in a restricted case when there are no concurrent readers (all memory accesses are treated as writes) and there are no transaction abort. If the underlying transaction-free program has T_1 work and T_∞ span, then the CWSTM executes the transactional

program in time $O(T_1/P + PT_\infty)$ when run on P processors. At first glance, these bounds might seem uninteresting due to the restrictions. It is difficult, however, for any TM system to provide any nontrivial bounds on completion time in the presence of aborts, since the system might redo an arbitrary amount of work. Moreover, TM with eager conflict detection that allows more than a constant number of shared readers to an object can potentially lead to memory contention; thus, even if there are no conflicts on that object, it seems difficult to provide efficient worst-case theoretical bounds.

9.2 CWSTM Framework

We use the computation tree framework, described in Chapter 6 to model CWSTM program executions. For simplicity in explanation, however, we use a canonical computation tree. In addition, instead of using the tree for *a posteriori* analysis, as in the previous chapter, CWSTM runtime system explicitly maintains the computation tree. The computation tree is not given *a priori* (i.e., from a static analysis of the program); rather, it unfolds dynamically as the program executes. Moreover, nondeterminism in the program may result in different computation trees. Constructing the computation tree on the fly as the program executes is not difficult and thus not described in full in this paper. A partial program execution corresponds to partial traversal of the computation tree.

In our canonical computation tree, all P-nodes have exactly 2 nontransactional S-nodes as children, while S-nodes can have an arbitrary number of children. In addition, we require that no nontransactional S-node has a child nontransactional S-node. Thus, it follows that all nontransactional S-nodes are children of P-nodes (or the root of the tree). For convenience, we treat the root of the computation tree as both a transactional and a nontransactional S-node.

For any node $B \neq \texttt{root}(\mathcal{C})$, we define the transactional parent $\texttt{xparent}[B]$ as $Z = \texttt{parent}[B]$ if Z is a transaction or $\texttt{root}(\mathcal{C})$, and as $\texttt{xparent}[Z]$ otherwise. Similarly, we define nontransactional S-node parent $\texttt{nsParent}[B]$ as $Z = \texttt{parent}[B]$ if Z is a nontransactional S-node or $\texttt{root}(\mathcal{C})$, or $\texttt{nsParent}[Z]$ otherwise.

At any point during the computation-tree traversal, each node B in the computation tree has a ***status***, denoted by $\texttt{status}[B]$. The status can be one of PENDING, PENDING_ABORT, COMMITTED, or ABORTED. A leaf is ***complete*** if the corresponding operation has been executed. An internal node can complete only if all nodes in its subtree are complete. Thus, a complete node corresponds to the root of a subtree that will not unfold further, and hence a node is complete if and only if its status is COMMITTED or ABORTED. A node is ***active*** (having status PENDING or PENDING_ABORT) if it has any unexecuted descendants. Once a node is complete, it can never become active again.

Any execution of the computation tree has the invariant that at any time, the set of active nodes in the computation tree also forms a tree, with the leaves of this active tree being the set of ***ready*** nodes. Only a node that is ready can be traversed to "discover" a new child node. (Discovering a new child node corresponds to executing an instruction in the program: a read or write creates new leaf below the current S-node, a transactional begin creates a new transactional S-node, and a fork statement creates a new P-node along with its two S-node children.) When a ready transactional S-node completes, its parent becomes ready. When a ready nontransactional S-node Z completes, if Z's sibling nontransactional S-node is already complete, then Z's parent (which is a P-node) completes, and Z's grandparent becomes ready.

Figure 9.5 shows the structure of the computation tree after an execution of the code from

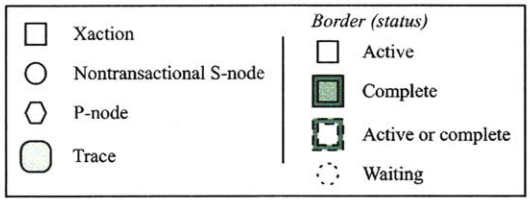

Figure 9.4: A legend for computation-tree figures.

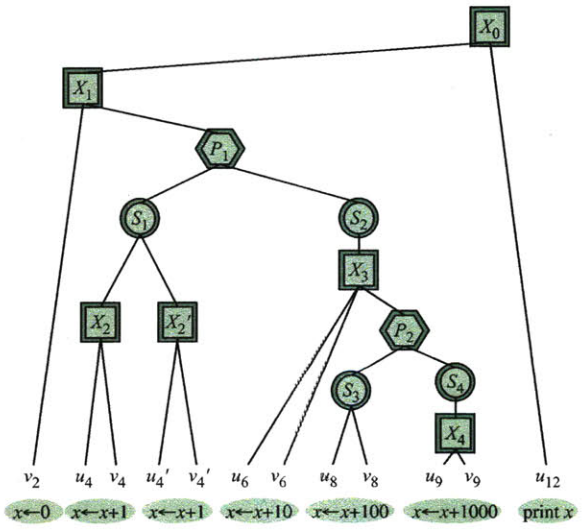

Figure 9.5: A computation tree for an execution of the program given by Figure 9.1, in which transaction X_2 aborted once and was retried as the transaction X_2'. The root X_0 does not correspond to any transaction in the program—it is just the S-node root of the tree. Each increment to x on line j of the program decomposes into two atomic memory operations: a `read` u_j, and a `write` v_j. The corresponding code is shown in a gray oval under the accesses.

Figure 9.1 in which transaction X_2 aborts once. If other aborts (and retries) occur, the computation tree would have additional subtrees.

9.3 CWSTM Semantics

This section describes CWSTM semantics, a semantics for a generic transactional memory system with nested parallel transactions and eager conflict detection. We describe these semantics operationally, in terms of a "readset" and "writeset" for each transaction. In particular, we define conflicts and describe transaction commits and aborts abstractly using readsets and writesets. Later, in Section 9.4, we give a simple design for a TM that provides these semantics. In Section 9.5, we improve the simple TM and present the the CWSTM design.

At any point during the program execution, the ***readset*** of a transaction X is the set of objects ℓ that X has "accessed". Similarly, the ***writeset*** is a set of objects ℓ that X has written to. Operationally, readsets and writesets change as follows. A transaction begins with an empty readset and empty writeset. Whenever a successful `read` of ℓ_1 occurs in a memory-operation (leaf) node u, ℓ_1 is added to $\texttt{xparent}[u]$'s readset. Similarly, whenever a successful `write` of ℓ_2 occurs in a memory-operation node u, ℓ_2 is added to $\texttt{xparent}[u]$'s readset and writeset. A `read` or a `write` to ℓ by an operation u "observes" the value associated with the write stored in the writeset of Z, where Z is the nearest transactional ancestor of u that contains ℓ in its writeset. For consistency, the writeset of the computation-tree's root contains all objects. When a transaction X commits, its readset and writeset are merged into $\texttt{xparent}[X]$'s readset and writeset, respectively.

A transactional memory with eager conflict detection must test for conflict before performing each `read` or `write`. 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 active transaction is allowed to contain ℓ in its writeset at one time. For CWSTM semantics, we generalize this definition of conflict in a straightforward manner. At any point in time, let $\texttt{readers}(\ell)$ and $\texttt{writers}(\ell)$ be the sets of *active* transactions that have object ℓ in their readsets or writesets, respectively. Then, we define conflicts as follows:

Definition 36 *At any point in time, a memory operation v generates a **conflict** if*

1. *v reads object ℓ, and $\exists X \in \texttt{writers}(\ell)$ such that $X \notin \texttt{ances}(v)$, or*
2. *v writes to object ℓ, and $\exists X \in \texttt{readers}(\ell)$ such that $X \notin \texttt{ances}(v)$.*

If there is such a transaction X, then we say that v conflicts with X. If v belongs to the transaction X', then we say that X and X' conflict with each other.

If a memory operation v would cause a conflict between $X = \texttt{xparent}[v]$ and another transaction X', then v triggers an abort of either X or X' (or both). Say X is aborted. An abort of a transaction X changes $\texttt{status}[X]$ 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, when a transaction X is `ABORTED`, CWSTM semantics discards X's writeset and readset. Since X is no longer active after this action occurs, the action also conceptually removes X from $\texttt{readers}(\ell)$ and $\texttt{writers}(\ell)$ for all objects ℓ. Note that in CWSTM, if v

causes a conflict, and the runtime chooses to abort $X' \neq \texttt{xparent}[v]$, then the conflict is not fully resolved until $\texttt{status}[X']$ has changed to ABORTED.

Consider a computation subtree rooted at a transaction X with $\texttt{status}[X] = \texttt{PENDING}$. Since we allow only closed-nested transactions, if every child of X has completed, CWSTM can commit X, i.e., change X's status from PENDING to COMMITTED, and merge X's readset and writeset into those of $\texttt{xparent}[X]$.

Code example

We can now describe how the CWSTM semantics constrain the possible outputs of the program in Figure 9.3. Since parallelism is allowed in transactions, we must consider the scoping of atomicity. In particular, the $x \leftarrow x + 1$ 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 36 naturally enforces strong atomicity [BLM06]. Strong atomicity implies that although line 8 is not atomic, it cannot perform its write between line 9's read and write. In terms of the computation tree in Figure 9.5, after u_9 performs a `read` of x, it adds x to the readset of X_4; thus, after u_9 occurs but before X_4 commits, if v_8 tries to write to x, it will cause a conflict with X_4. We can, however, have line 8 read x, line 9 read and write x and commit, and then line 8 write x. This interleaving can occur because when u_8 happens, it adds x to the readset of X_3, and u_9 and v_9 can subsequently happen because they are both descendants of X_3 in the computation tree. This behavior means that 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 a transaction.

Semantic guarantees

The CWSTM semantics maintains the invariant that a program execution is always conflict-free, according to Definition 36. One can show that when transactions have nested parallel transactions, TM with eager conflict detection according to Definition 36 satisfies the transactional-memory model of prefix race-freedom defined in [ALS06].[1] As shown in [ALS06], prefix race-freedom and serializability are equivalent if one can safely "ignore" the effects aborted transactions. Note that this equivalence may not hold in TM systems with explicit `retry` constructs that are visible to the programmer.

Definition 36 directly implies the following lemma about a conflict-free execution.

Lemma 9.1 *For a conflict-free execution, the following invariants hold for any object ℓ:*

1. *All transactions $X \in \texttt{writers}(\ell)$ fall along a single root-to-leaf path in C. Let $\texttt{lowest}(\texttt{writers}(\ell))$ denote the unique transaction $Y \in \texttt{writers}(\ell)$ such that $\texttt{writers}(\ell) \subseteq \texttt{ances}(Y)$.*

[1] The proof is a special case of the proof for the operational model described in Chapter 6, without any open-nested transactions.

2. *All transactions $X \in \texttt{readers}(\ell)$ are either along the root-to-leaf path induced by the writers or are descendants of the* $\texttt{lowest}(\texttt{writers}(\ell))$.

We use Lemma 9.1 to argue that one can check for conflicts for a memory operation u by looking at one writer and only a small number of readers. Since all the transactions fall on a single root-to-leaf path, by Lemma 9.1, Invariant 1, the transaction $\texttt{lowest}(\texttt{writers}(\ell))$ belongs to $\texttt{writers}(\ell)$ and is a descendant of all transactions in $\texttt{writers}(\ell)$. Similarly, let $Q = \texttt{lastReaders}(\ell) \subseteq \texttt{desc}(\texttt{lowest}(\texttt{writers}(\ell)))$ denote the set of readers implied by Invariant 2. If a memory operation u tries to read ℓ, abstractly, there is no conflict exactly if and only if $\texttt{lowest}(\texttt{writers}(\ell))$ is an ancestor of u. Similarly, when u tries to write to ℓ, by Invariant 2, there is no conflict if for all $Z \in \texttt{lastReaders}(\ell)$, Z is an ancestor of u.

9.4 A Naive TM

The CWSTM semantics described in Section 9.3 suggest a design for a TM system that supports transactions with nested parallelism. In particular, Lemma 9.1 suggests that for each object ℓ, the TM can maintain an active writing transaction $\texttt{lowest}(\texttt{writers}(\ell))$ and some active reading transactions $\texttt{lastReaders}(\ell)$. This scheme allows transactions accessing ℓ to test for conflicts against these transactions. This section focuses on a straightforward data structure, called an "access stack," used to maintain these values. We show that an access stack yields a TM with poor worst-case performance, even assuming the rest of the TM system incurs no overhead. The CWSTM design uses a lazy variant of the access stack, described in Section 9.5, that has much better performance.

The ***access stack*** for an object ℓ is a stack containing the active transactions that have written to ℓ and sets of active 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 $\texttt{lowest}(\texttt{writers}(\ell))$ 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 $\texttt{lastReaders}(\ell)$ is. No two consecutive elements are sets of readers.

The access stack is maintained as follows, locking the relevant stack on all memory access 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 $\texttt{xparent}[X]$ (in a fashion that ensures there are no duplicated transactions). This action mimics the commit semantics from Section 9.3: when a transaction X commits, the objects in its readset and writeset are moved to $\texttt{xparent}[X]$'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.[2]

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

[2]This value can either be stored in the stack itself, or in a log per transaction.

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 following code snippet:

```
void f(int i) {
  if (i >= 1) {  atomic { x[i]++; f(i-1); }   }
}
```

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 due to lazy object updates also has this issue.

9.5 CWSTM Overview

This section describes our CWSTM design for a transactional-memory system with nested parallel transactions and eager updates and eager conflict detection. We first describe how CWSTM updates the computation-tree-node statuses on commits and aborts. We then give an overview of the conflict-detection mechanism, deferring details of the XConflict data structure to later sections. The conflict-detection mechanism includes a "lazy access stack," improving on the shortcoming of the access stack from Section 9.4. Finally, we describe properties of the Cilk-like work-stealing scheduler that CWSTM uses. The XConflict data structure requires such a scheduler for its performance and correctness.

CWSTM explicitly builds the internal nodes of the computation tree (i.e., leaf nodes for memory operations are omitted). Each node maintains a status field which in most cases, explicitly represents the node's status (`PENDING_ABORT`, `PENDING`, `COMMITTED`, or `ABORTED`), and changes in a straightforward fashion. 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`$[X] \leftarrow$ `ABORTED`. We refer to this type of update as an `xabort`. Alternatively, suppose a processor p_i wishes to abort X even though p_i is not currently executing X. First, p_i atomically changes `status`$[X]$ from `PENDING` to `PENDING_ABORT`. Then p_i walks X's active subtree, changing `status`$[Y] \leftarrow$ `PENDING_ABORT` atomically for each active $Y \in$ `desc`(X). Notice that p_i never changes any status to `ABORTED`—only the processor running a transaction Y is allowed to perform that update. When X "discovers" that its status has changed to `PENDING_ABORT`, it has no active descendants (otherwise, X cannot be ready, and hence X cannot be executing). Then, X simply performs an `xabort` on itself.

For reasons specific to XConflict, the data structure the CWSTM design uses for conflict detection, during an abort of X, some of X's *COMMITTED* descendants Y also have their status

```
XCONFLICT-ORACLE(X, u)
    ▷ For any node X and active memory operation u
1   if ∃Z ∈ ances(X) such that status[Z] = ABORTED
2       then return "no conflict: X aborted"

3   Y ← closest active transactional ancestor of X
4   if Y ∈ ances(u)
5       then 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 9.6: Pseudocode for a conflict-detection query suggested by Definition 37. Many subroutine (e.g., line 3) details are omitted (and in fact do not have efficient implementations). The LCA function returns the least common ancestor of two nodes in the computation tree.

field changed to ABORTED. Our conflict-detection algorithm uses these updates to more quickly determine that a memory operation does not conflict with Y, since Y has an ABORTED ancestor X. Section 9.9 describes when these updates occur.

In CWSTM, the rollback of objects on abort occur 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 aazy access stack

We now discuss conflict detection. The key observation that allows us to avoid explicit maintenance of active readers and writers (or transaction readsets and writesets) is the following alternate conflict definition.

Definition 37 *Consider a (possibly inactive) transaction X that has written to ℓ and a new memory operation v that reads from or writes to ℓ. Then v does* not *conflict with X if and only if*

1. *some transactional ancestor of X has aborted, or*
2. *X's nearest active transactional ancestor is an ancestor of v.*

The case when X has read from ℓ and v writes to ℓ is analogous.

This definition is equivalent to Definition 36 because X's nearest active transactional ancestor logically belongs to writers(ℓ) if X doesn't have an aborted ancestor.

Definition 37 suggests a conflict-detection algorithm that does not require maintaining lowest(writers(ℓ)) and the normal access stack. In particular, let X be the last node that has successfully written to ℓ. Then 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 9.6 gives pseudocode for this test. Note that CWSTM does not actually implement this query as given—instead, it uses an equivalent, but more efficient query, described in Section 9.6.

To maintain the most recent successful write (and reads), facilitating the necessary conflict queries, CWSTM uses a ***lazy access stack***. The structure of the lazy access stack is somewhat different from the simple access stack given in Section 9.4. An object ℓ's lazy stack stores (possibly complete) transactions that have written to ℓ and sets of transactions that have read from ℓ, but now these stack entries are ordered chronologically by access. The top of the stack holds the last writer or the last readers. We have 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 represent deprecated values. The main difference in maintenance is that the lazy access stack is not updated on transactional commit (thus ignoring the merge of a transaction's readset and writeset into its parent's). On memory operations, new transactions are added to the access stack in the same way as described in Section 9.4.

Figure 9.7 gives pseudocode for an instrumentation of each memory access, assuming for simplicity that all memory accesses behave as `write` instructions.[3] Incorporating readers into the access stacks is more complicated, but conceptually similar. If a memory access u does not belong to an aborting transaction, then it is allowed to proceed. First, we test for conflict with the last writer in lines 4–5. If the last writer has aborted (or has an aborted ancestor), handled in lines 6–9, then the access stack should be cleaned up by calling CLEANUP. (This auxiliary procedure, given in Figure 9.8, rolls back the value of the topmost aborted transaction on ℓ's access stack.) Since there is no new conflict, after CLEANUP, the access should be retried. If, on the other hand, there is a conflict between u and an active transaction (lines 10–16), then either $\texttt{xparent}[u]$ must abort or the conflicting transaction (B) must abort. Finally, if there are no conflicts, then the access is successful. The access stack is updated as necessary (lines 18–20), and the access is performed.

Note that 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 never create a new conflict with u. Thus, concurrent changes may introduce spurious aborts, but do not affect correctness.

The CWSTM scheduler

XConflict relies on a Cilk-like work-stealing scheduler for efficiency and correctness. The main idea of a work stealing is that when a processor completes its own work, it "steals" work from a different ***victim*** processor. Conceptually, the entire (unexpanded) computation tree is initially "owned" by a single processor. A processor traverses the current subtree that it owns, and only that subtree that it owns. As this processor "discovers" P-nodes it executes one of its nontransactional S-node children, and the other child can subsequently be stolen by a thief processor. Whenever a processor p_i has no work (does not own a subtree), it steals a subtree T rooted at such a nontransactional S-node. Thus, p_i now owns and traverses the subtree T.

When we say a work-stealing scheduler is 'Cilk-like," as required by XConflict, we mean that it has the following two properties. First, a processor executes its computation subtree in a left-to-right fashion. Second, whenever a thief processor steals work from a victim processor, it steals the right subtree from the highest P-node in the victim's subtree that has work available.

[3] It is possible to reduce locking on the access stack, but we do not describe that optimization in this paper.

```
ACCESS(u, ℓ)
 1  Z ← xparent[u]
 2  if status[Z] = PENDING_ABORT return XABORT
    ▷ Otherwise Z is active
 3  accessStack(ℓ).LOCK()

    ▷ Set X to be the last writer.
 4  X ← accessStack(ℓ).TOP()
 5  result ← XCONFLICT-ORACLE(X, u)

 6  if result is "no conflict: X aborted"
 7     then accessStack(ℓ).UNLOCK()
 8          CLEANUP(ℓ)                          ▷ Rollback some values
 9          return RETRY                        ▷ The access should be retried

10  if result indicates a conflict with transaction B
11     then if choose to abort self
12             then accessStack(ℓ).UNLOCK()
13                  return XABORT
14             else accessStack(ℓ).UNLOCK()
15                  signal an abort of B
16                  return RETRY

    ▷ Otherwise, there is no conflict: X is an ancestor of Z
17  if Z ≠ X                                      ▷ Z's first access to ℓ
18     then ▷ Log the access
19          LOGVALUE(Z, ℓ)
20          accessStack(ℓ).PUSH(Z)

    ▷ Actually perform the write operation
21  Perform the write
22  accessStack(ℓ).UNLOCK()
23  return SUCCESS
```

Figure 9.7: Pseudocode instrumenting an access by u to 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(ℓ)
1  accessStack(ℓ).LOCK()
2  X ← accessStack(ℓ).TOP()
3  if ∃Z ∈ ances(X) such that status[Z] = ABORTED
4     then RESTOREVALUE(X, ℓ) ▷ Restore ℓ from X's log
5          accessStack(ℓ).POP()
6  accessStack(ℓ).UNLOCK()
```

Figure 9.8: Code for cleaning up an aborted transaction from the top of accessStack(ℓ), assuming all accesses are writes. If the last writer has an aborted ancestor, it should be rolled back.

9.6 CWSTM Conflict Detection

This section describes the high-level ***XConflict*** scheme for conflict detection in CWSTM. As the computation tree dynamically unfolds during an execution, our algorithm dynamically divides the computation tree into "traces," where each trace consists of memory operations (and internal nodes) that execute on the same processor. Our 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.

By dividing the computation tree into traces, we reduce the cost of locking on shared data structures. Updates and queries on a data structure whose elements belong to a single trace are also performed without locks because these updates are performed by a single processor. 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 traces are created only on steals, however, we can bound the number of traces by $O(pT_\infty)$—the number of steals performed by the Cilk-like work-stealing runtime system. Therefore, the number of updates on these data structure can be bounded similarly.

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 Bender et al.'s [BFGL04] SP-hybrid algorithm for series-parallel maintenance (later improved in [Fin05]). Our traces differ slightly, and our 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 [BFGL04, Fin05]. We briefly describe the structure of traces here. Since our computation tree 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.

Formally, each trace U is a disjoint subset of nodes of the (*a posteriori*) computation tree. We let $\mathcal{Q}$ denote the set of all traces. $\mathcal{Q}$ partitions the nodes of the computation tree C. Initially, the entire computation belongs to a single trace. As the program executes, traces dynamically split into

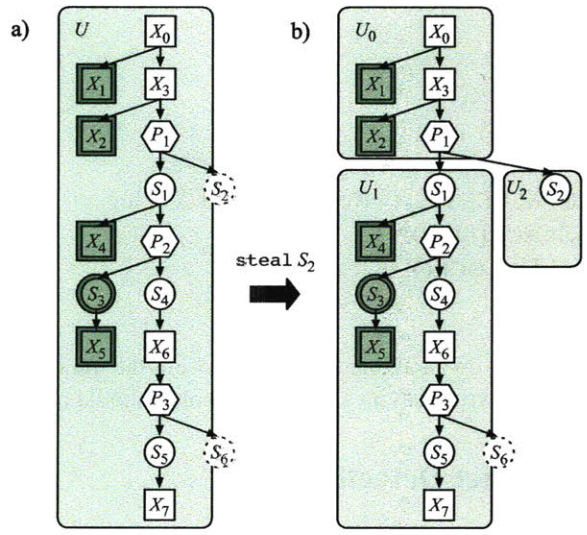

Figure 9.9: Traces of a computation tree (a) before and (b) after a `steal` action. Before the `steal`, only one processor is executing the subtree, but S_2 and S_6 are ready. After the steal, the subtree rooted at the highest ready S-node (S_2) is executed by the thief. The subtree rooted at S_1, on the other hand, is still owned and executed by the victim processor.

multiple traces whenever steals occur.

A trace itself executes on a single processor in a depth-first manner. Whenever a steal occurs and a processor steals the right subtree of a P-node $P \in U$, the trace U splits into three traces U_0, U_1, and U_2 (i.e., $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{U_0, U_1, U_2\} - \{U\}$). 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. Notice that although the processor performing the steal begins work on *only the right subtree* of P, both subtrees become new traces. Figure 9.9 gives an example of traces resulting from a steal. The left and right children of the *highest uncompleted* P-node P_1 (both these nodes are nontransactional S-nodes in our canonical tree) are the roots of two new traces, U_1 and U_2.

Traces in CWSTM satisfy the following properties.

Property 1 *Every trace $U \in \mathcal{Q}$ has a well-defined **head** nontransactional S-node $S = \texttt{head}[U] \in U$ such that for all nodes $B \in U$, we have $S \in \texttt{ances}(B)$.*

For a trace $U \in \mathcal{Q}$, we use $\texttt{xparent}[U]$ as a shorthand for $\texttt{xparent}[\texttt{head}[U]]$. We similarly define $\texttt{nsParent}[U]$.

Property 2 *The computation-tree nodes of a trace $U \in \mathcal{Q}$ form a tree rooted at $S = \texttt{head}[U]$.*

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 a P-node P has a stolen child (i.e., P and its children belong to different traces). Consider all ancestor P-nodes P' of P such that P is in the left subtree of P'. Then P' must have a stolen child (i.e., P and ancestor P' belong to different traces).*

The last property follows from the Cilk-like work stealing.

The partition $\mathcal{Q}$ of nodes in the computation tree $\mathcal{C}$ induces a tree of traces $J(\mathcal{C})$ as follows. For any traces $U, U' \in \mathcal{Q}$, there is an edge $(U, U') \in J(\mathcal{C})$ if and only if $\texttt{parent}[\texttt{head}[U']] \in U$.[4] The properties of traces and the fact that traces partition $\mathcal{C}$ into disjoint subtrees together imply that $J(\mathcal{C})$ is also a tree.

We say that a trace U is ***active*** if and only if $\texttt{head}[U]$ is active. The following lemma states that if a descendant trace U' is active, then U' is a descendant of *all* active nodes in U. The proof relies on the fact that traces execute serially in a depth-first (or equivalently, left-to-right) manner.

Lemma 9.2 *Consider active traces $U, U' \in \mathcal{Q}$, with $U \neq U'$. Let $D \in U'$ be an active node, and suppose $D \in \texttt{desc}(\texttt{head}[U])$ (i.e., U' is a descendant trace of U). Then for any active node $B \in U$, we have $B \in \texttt{ances}(D)$.*

PROOF. Since traces execute on a single processor in a depth-first manner, only a single head-to-leaf path of each trace can be active. Thus, if a descendant trace U' is active, it must be the descendant of some node along that path. In particular, we claim that U' is a descendant of the leaf, and hence it is a descendant of *all* active nodes as the lemma states. This claim follows from Property 3, because both children of the *active* P-node on the trace boundary must belong to different traces. □

XConflict algorithm

Recall that CWSTM instruments memory accesses, testing for conflicts on each memory access by performing queries of XConflict data structures. In particular, XConflict must test whether a recorded access by node B conflicts with the current access by node u. Suppose that B does not have an aborted ancestor. Then recall Definition 37 states that a conflict occurs if only if the nearest uncommitted transactional ancestor of B is *not* an ancestor of u.

A straightforward algorithm (given in Figure 9.6) for conflict detection finds the nearest uncommitted transactional ancestor of B and determines whether this node is an ancestor of u. Maintaining such a data structure subject to parallel updates is costly (in terms of locking overheads).

XConflict performs a slightly simpler query that takes advantages of traces. XConflict does not explicitly find the nearest uncommitted transactional ancestor of B; it does, however, still determine whether that transaction is an ancestor of u. In particular, let Z be the nearest uncommitted transactional ancestor of B, and let U_Z be the trace that contains Z. Then XConflict finds U_Z (without necessarily finding Z). Testing whether U_Z is an ancestor of u is sufficient to determine whether Z is an ancestor of u. Note that XConflict does not lock on any queries. Many of the subroutines (described in later sections) need only perform simple ABA tests to see if anything changed between the start and end of the query.

[4]The function $\texttt{parent}[]$ refers to the parent in the computation tree $\mathcal{C}$, not in the trace tree $J(\mathcal{C})$.

The XCONFLICT algorithm is given by pseudocode in Figure 9.10. lines 1–4 handle the simple base cases. If B and u belong to the same trace, they are executed by a single processor, so there is no conflict. If B is aborted, there is also no conflict.

Suppose B is not aborted and that B and u belong to different traces. XCONFLICT first finds X, the nearest transactional ancestor of A that belongs to an active trace, in line 5. The possible locations of X in the computation tree are shown in Figure 9.12. Let $U_X = \mathsf{trace}(X)$. Notice that U_X is active, but X may be active or inactive. For cases (a) or (b), we find X with a simple lookup of $\texttt{xparent}[B]$. Case (c) involves first finding U, the highest completed ancestor trace of $\mathsf{trace}(B)$, then performing a simple lookup of $\texttt{xparent}[U]$. Section 9.9 describes how to find the highest completed ancestor trace.

Line 9 finds Y, the highest active transaction in U_X. If Y exists and is an ancestor of X, as shown in the left of Figure 9.13, then XCONFLICT is in the case given by lines 11–13. If U_X is an ancestor of u, we conclude that A has committed to an ancestor of u. Figure 9.13 (a) and (b) show the possible scenarios where U_X is an ancestor of u: either X is an ancestor u, or X has committed to some transaction Z that is an ancestor of u.

Suppose instead that Y is not an ancestor of X (or that Y does not exist), as shown in the left of Figure 9.14. Then XCONFLICT follows the case given in lines 15–17. Let Z be the transactional parent of U_X. Since X has no active transactional ancestor in U_X, it follows that X has committed to Z. Thus, if $\mathsf{trace}(Z)$ is an ancestor of u, we conclude that A has committed to an ancestor of u, as shown in Figure 9.14.

Section 9.7 describes how to find the trace containing a particular computation-tree node (i.e., computing $\mathsf{trace}(B)$). Section 9.8 describes how to maintain the highest active transaction of any trace (used in line 9). Section 9.9 describes how to find the highest completed ancestor trace of a trace (used for line 5), or find an aborted ancestor trace (line 3). Computing the transactional parent of any node in the computation tree ($\texttt{xparent}[B]$) is trivial. Section 9.10 describes a data structure for performing ancestor queries within a trace (line 10), and a data structure for performing ancestor queries between traces (lines 11 and 15).

The following theorem states that XConflict is correct.

Theorem 9.3 *Let B be a node in the computation tree, and let u be a currently executing memory access. Suppose that B does not have an aborted ancestor. Then* XCONFLICT(B, u) *reports a conflict if and only if the nearest (deepest) active transactional ancestor of B is an ancestor of u.*

PROOF. If B has an aborted ancestor, then XCONFLICT properly returns no conflict.

Let Z be the nearest active transactional ancestor of B. Let U_Z be the trace containing Z; since Z is active, U_Z is active. Lemma 9.2 states that U_Z is an ancestor of u if and only if Z is an ancestor of u. It remains to show that XConflict finds U_Z.

XConflict first finds X, the nearest transactional ancestor of B belonging to an active trace (line 5). The nearest active ancestor of B must be X or an ancestor of X. Let U_X be the trace containing X, and let Y be the highest active transaction in U_X. If Y is an ancestor of X, then either $Z = X$, or Z is an ancestor of X and a descendant of Y (as shown in Figure 9.13). Thus, XConflict performs the correct test in lines 11–13.

Suppose instead that Y, the *highest* active transaction in U_X, is not an ancestor of X. Then no active transaction in U_X is an ancestor of X. Let Z be the transactional ancestor of U_X. Since U_X is active, Z must be active. Thus, Z is the nearest uncommitted transactional ancestor of B, and XConflict performs the correct test in lines 15–17.

```
XCONFLICT(B, u)
    ▷ For any computation-tree node B and any
      active memory-operation u

    ▷ Test for simple base cases
 1  if trace(B) = trace(u)
 2     then return "no conflict"
 3  if some ancestor transaction of B is aborted
 4     then return "no conflict: B aborted"

 5  Let X be the nearest transactional ancestor of B
       belonging to an active trace.
 6  if X = null                                  ▷ committed at top level
 7     then return "no conflict: B committed to root"
 8  U_X ← trace(X)

 9  Let Y be the highest active transaction in U_X

10  if Y ≠ null and Y is an ancestor of X
11     then if U_X is an ancestor of u
12             then return "no conflict: B committed
                                  to u's ancestor"
13             else return "conflict with Y"
14     else Z ← xparent[U_X]
15          if Z = null or trace(Z) is an ancestor of u
16             then return "no conflict: B committed
                                  to u's ancestor"
17             else return "conflict with Z"
```

Figure 9.10: Pseudocode for the XConflict algorithm.

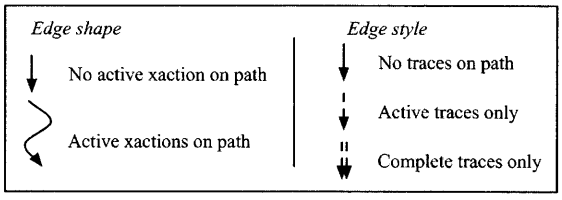

Figure 9.11: The definition of arrows used to represent paths in Figures 9.12, 9.13 and 9.14.

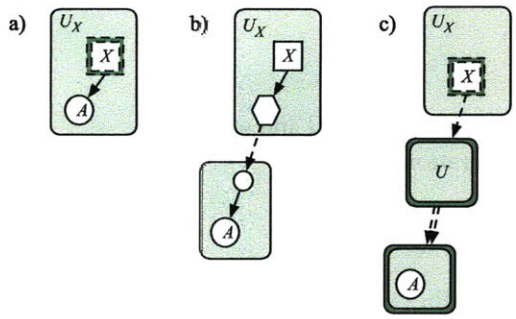

Figure 9.12: The three possible scenarios in which X is the nearest transactional ancestor of B that belongs to an active trace. Arrows represent paths between nodes (i.e., many nodes are omitted): see Figures 9.4 and 9.11 for definitions. In both (a) and (b), B belongs to an active trace. In (a), $\texttt{xparent}[B]$ belongs to the same active trace as B. In (b), $\texttt{xparent}[B]$ belongs to an ancestor trace of $\mathsf{trace}(B)$. In (c), B belongs to a complete trace, U is the highest completed ancestor trace of B, and X is the $\texttt{xparent}[U]$.

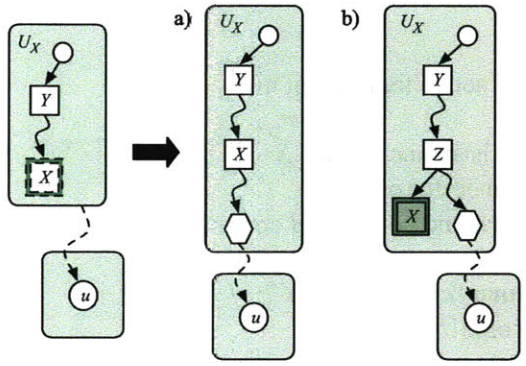

Figure 9.13: The possible scenarios in which the highest active transaction Y in U_X is an ancestor of X, and U_X is an ancestor of u (i.e., line 11 of Figure 9.10 returns true). Arrows represent paths between nodes (i.e., many nodes are omitted): see Figures 9.4 and 9.11 for definitions. The block arrow shows implication from the left side to either (a) or (b).

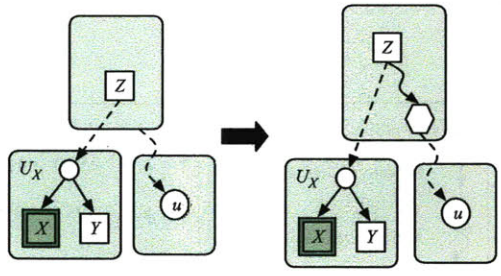

Figure 9.14: The scenario in which the highest active transaction Y in U_X is not an ancestor of X, and $Z = \texttt{xparent}[U_X]$ is an ancestor of u (i.e., line 15 of Figure 9.10 returns true). The block arrow shows implication from the left side to the situation on the right.

In the above explanation, we assume that no XConflict data-structural changes occur concurrently with a query. The case of concurrent updates is a bit more complicated and omitted from this proof. The main idea for proving correctness subject to concurrent updates is as follows. Even when trace splits occur, if a conflict exists, XCONFLICT has pointers to traces that exhibit the conflict. Similarly, if XCONFLICT acquires pointers to a transaction (Y or Z) deemed to be active, that transaction was active at the start of the XCONFLICT execution.

□

Note that XCONFLICT may return some spurious conflicts if transactions complete during the course of a query.

9.7 Trace Maintenance

This section describes how to maintain trace membership for each node B in the computation tree, subject to queries $\mathsf{trace}(B)$. The queries take $O(1)$ time in the worst case. We give the main idea of the scheme here for completeness, but we omit details as they are similar to the local-tier of SP-hybrid [BFGL04, Fin05].

To support trace membership queries, XConflict organizes computation-tree nodes belonging to a single trace as follows. Nodes are associated with their nearest nontransactional S-node ancestor. These S-nodes are grouped into sets, called "trace bags." To be more precise, for each nontransactional S-node $S \in \mathtt{nsNodes}(\mathcal{C})$, XConflict creates bag called $\mathsf{FBag}(S)$ with the following property. Let $\mathsf{nsSet}(S) = \{S' \in \mathtt{nsNodes}(S) : \mathtt{nsParent}[S'] = S\}$ be the set of all nearest nontransactional S-node descendants. Then for any $S' \in \mathsf{nsSet}(S) \cap \mathsf{trace}(U)$, we have $\mathsf{FBag}(S) = \mathsf{FBag}(S')$ if and only if S' has completed. In other words, $\mathsf{FBag}(S)$ contains all the descendant S-nodes of procedures that have returned to S.

Each bag b has a pointer to a trace, denoted $\mathtt{traceField}[b]$, which must be maintained efficiently. A trace may contain many trace bags. Bags are merged dynamically in a way similar to the SP-bags [FL97] in the local tier of SP-hybrid [BFGL04, Fin05] using a disjoint-sets data structure [CLRS01, Chapter 21]. XConflict UNIONs bags when nontransactional S-nodes complete. That is, when S' completes, we perform $\textsc{Union}(S, S')$ if and only S and S' still belong to the same trace (i.e., if a steal has not occurred). Since traces execute on a single processor, we do not lock the data structure on update (UNION) operations. The difference in our setting is that we use only one kind of bag (instead of two in SP-bags).

When steals occur, a global lock is acquired, and then a trace is split into multiple traces, as in the global tier of SP-hybrid [BFGL04, Fin05]. The difference in our setting is that traces split into three traces (instead of five in SP-hybrid). 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 (including the split work in Sections 9.8 and 9.10), the global lock is released. Consider the bags at the time a node S_2 is stolen from the trace U with $\mathtt{head}[U] = S$, and let S_1 be S_2's left sibling (the parent is a P-node). There are three new traces U_0, U_1, and U_2 created, rooted at S, S_1, and S_2, respectively. Since CWSTM executes in a depth-first manner and performs steals from the *highest* P-node, it follows that all descendant nodes of S belonging to U that are not descendants of S_1 (or S_2) still belong to the bag $\mathsf{FBag}(S)$. Moreover, these are exactly the nodes that belong to the resulting trace U_0. Thus, moving nodes to U_0 is simply a matter of updating $\mathtt{traceField}[\mathsf{FBag}(S)] \leftarrow U_0$. Since S_2 is only now beginning to execute, its trace is initially

empty, and we simply perform MAKE-SET(FBag(S_2)) and traceField[FBag(S_2)] $\leftarrow U_2$. There may be many bags residing in U_1, but these bags all previously belonged to U, so we can simply rename U as U_1. Thus, updating trace bags after a steal require only $O(1)$ pointer updates.

To query what trace a node B belongs to, we perform the operations traceField[FIND-BAG(nsParent[B] These queries (in particular, FIND-BAG) take $O(1)$ worst-case time as in SP-hybrid [BFGL04, Fin05]. Merging bags uses an UNION operation and takes $O(1)$ amortized time, but an optimization [Fin05] gives a technique that improves UNIONs to worst-case $O(1)$ time whenever the amortization might adversely increase the program's critical path.

9.8 Highest Active Transaction

This section describes how XConflict finds the highest active transaction in a trace, used in line 9 of Figure 9.10 in $O(1)$ time.

For each nontransactional S-node S, we have 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, so instead we maintain it only for some S-nodes as follows. Let $S \in U$ be an active nontransactional S-node such that either $S =$ head[U], or S is the left child of a P-node and S's nearest S-node ancestor (which is always a grandparent) is a transaction. Then $nextx[S]$ is defined to be the nearest, active descendant transaction of S in U. Otherwise, $nextx[S] =$ null.

Finding the highest active transaction simply entails a call to $nextx$[head[U]], which takes $O(1)$ time. The complication is maintaining the $nextx$ values, especially subject to dynamic trace splits.

To maintain $nextx$, we keep 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 $nextx[S] \leftarrow X$. Otherwise, do nothing. Whenever completing a transaction X, check $nextx[S]$. If $nextx[S] = X$, then set $nextx[S] \leftarrow$ null. Otherwise, do nothing. Whenever encountering a nontransactional S-node S'. If $nextx[S] =$ null, do nothing. Otherwise, push S' onto the stack. Whenever completing a nontransactional 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, nontransactional S-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 we leave 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 $nextx[S_1] \leftarrow nextx[S]$ and $nextx[S] \leftarrow$ 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 splitting occurs while holding the global lock acquired during the steal (as in Section 9.7).

9.9 Supertraces

This section describes XConflict's data structure to find the highest completed ancestor trace of a given trace (used as a subprocedure for line 5 in Figure 9.10, illustrated by U in Figure 9.12 (c)). To facilitate these queries, XConflict groups traces together into "supertraces." Grouping traces into supertraces also facilitates faster aborts—when aborting a transaction in trace U, we need only abort some of the supertrace children of U, not the entire subtree in $\mathcal{C}$. This section also provides some details on performing the abort.

All update operations on supertraces take place while holding the same global lock acquired during the steal (as in Sections 9.7, 9.8, and 9.10). Note that unlike the data structures in Sections 9.7, 9.8, and 9.10, the updates to supertraces do not occur when steals occur. To prove good performance (in Section 9.11), we use the fact that the number of supertrace-update operations is asymptotically identical to the number of steals. This amortization is similar to the "global tier" of SP-hybrid [BFGL04].

At any point during program execution, a completed trace $U \in \mathcal{Q}$ belongs to a ***supertrace*** $K = \mathsf{strace}(U) \subseteq \mathcal{Q}$. In particular, the traces in K form a tree rooted at some ***representative trace*** $rep[K]$, which is an ancestor of all traces in K. Our structure of supertraces is such that either $rep[\mathsf{strace}(U)]$ is the highest completed ancestor trace of U (i.e., as used by line 5 in Figure 9.10), or U has an aborted ancestor. We prove this claim in Lemma 9.4 after describing how to maintain supertraces.

Supertraces are implemented using a disjoint-sets data structure [CLRS01, Chapter 21]. In particular, we use Gabow and Tarjan's data structure 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).

When a trace U is created, we create an empty supertrace for U (so $\mathsf{strace}(U) = \emptyset$). When the trace completes (i.e., at a `join` operation), we acquire the global lock. We then add U to U's supertrace (giving $\mathsf{strace}(U) = \{U\}$). Next, we consider all child traces U' of U (in the tree of traces $J(\mathcal{C})$).[5] If $\texttt{head}[U']$ is `ABORTED`, then we skip U'. If $\texttt{head}[U']$ is `COMMITTED`, we merge the two supertraces with $\textsc{Union}(\mathsf{strace}(U), \mathsf{strace}(U'))$. Thus, for U' (and all relevant descendants), $rep[\mathsf{strace}(U')] = rep[\mathsf{strace}(U)] = U$. Once these updates complete, the global lock is released. Later, U's supertrace may be merged with its parents, thereby updating $rep[\mathsf{strace}(U)]$.

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`. Instead, we only walk the subtree rooted at X in U, not $\mathcal{C}$. Whenever hitting a trace boundary (i.e., $B \in U$, $D \in \texttt{children}(B)$, $D \in U' \neq U$), we set that root of the child trace (D) to be aborted and do not continue into its descendants. Thus, we enforce all descendants of B have a supertrace with an aborted representative.

The following lemma states that either the representative of U's supertrace is the highest completed ancestor trace of U, or U has an aborted ancestor.

Lemma 9.4 *For any completed trace $U \in \mathcal{Q}$, let $K = \mathsf{strace}(U)$, and let $U' = rep[K]$. Exactly one of the following cases holds.*

1. *Either* $\texttt{head}[U']$ *is* `ABORTED`*, or*

[5] 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.

2. head[U'] *is* COMMITTED, trace(parent[head[U']]) *is active, and there is no* ABORTED *transaction between* head[U] *and* head[U'].

PROOF. We claim also that U' is an ancestor of all traces in K. We prove this claim and the lemma inductively on the (tree of traces) height of the highest committed ancestor of U.

For a base case, consider the moment when U completes. Then K contains only descendants of U, and $rep[K] = U$ satisfying our claims.

Suppose that U's highest committed ancestor occurs at height h in the tree of traces, and assume that $U' = rep[K]$ satisfies the lemma. Let V be height-$(h+1)$ ancestor of U. Then we show that the lemma is maintained when V commits. We divide the proof into the two cases.

Suppose head[U'] is ABORTED. Since U' is an ancestor of all traces in K, the only time U' can be merged with another trace is when its parent completes. ABORTED children, however, are not unioned. Thus, once the representative U' of a supertrace is aborted, the supertrace does not change.

Suppose instead that head[U'] is COMMITTED. Then when V commits, XConflict unions the two supertraces for U' and V, and then sets that supertrace's representative to V. Thus, rep[strace(U)] is the highest completed ancestor trace of U. This trace may be COMMITTED or ABORTED, but either satisfies the claim. □

9.10 Ancestor Queries

This section describes how XConflict performs ancestor queries. XConflict performs a "local" ancestor query of two nodes belonging to the same trace (line 10 of Figure 9.10) and a "global" ancestor query of two different traces (lines 11 and 15 of Figure 9.10). Both of these queries can be performed in $O(1)$ worst-case time. The global lock is acquired only on updates to the global data structure, which occurs on trace splits (i.e., steals).

Local ancestor queries

CWSTM executes a trace on a single processor, and each trace is 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 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 [CLRS01, Section 22.3].) These timestamps are sufficient to perform ancestor queries in constant time. To mark these times, we increment a counter each time a node is labeled.

To query whether a node u is an ancestor of a node v, we simply need to compare the discovery and finish times. The following lemma states a well known fact about these timestamps induced by depth-first search

Lemma 9.5 *Consider a depth-first tree walk, and let $d[u]$ and $f[u]$ be the discovery and finish times, respectively, of a node u. Then u is an ancestor of v if and only if $d[u] < d[v]$ and $f[u] > f[v]$.* □

In the setting of XConflict, we need to perform these local ancestor queries while traces split dynamically. This dynamic split of traces, however, does not significantly affect the timestamps associated with each node *within a trace*. In particular, consider when a trace U splits into three trace U_0, U_1, and U_2. There is initially a (time) counter associated with U. After the split, we copy the value of this counter to the counter associated with each of the resulting traces U_0, U_1, and U_2. Thus, the relative discovery and finish times between nodes in a single trace remain correct.

Global ancestor queries

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, we keep two total orders on the traces dynamically using order-maintenance data structures [DS87, BCD+02]. These two orders give us enough information to query the ancestor-descendant relationship between two nodes in the tree of traces. These total orders are updated while holding the global lock acquired during the steal, as in Sections 9.7 and 9.8. Since our global ancestor-query data structure resembles the global series-parallel-maintenance data structure in SP-hybrid [BFGL04], we omit 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 $O(1)$.

The two orders used are ***left-to-right*** order and ***right-to-left*** order. In our ***left-to-right order***, a node U precedes (all the nodes in) the left subtree of U, which precedes (all the nodes in) the right subtree of U. In other words, we order the nodes (traces) in the order they are output in a preorder tree walk that visits the children of a node in left-to-right order. In our ***right-to-left order***, a node U precedes its descendants, but now U's *right* subtree precedes U's *left* subtree. This ordering corresponds to a preorder tree walk that visits the children of a node in right-to-left order. The left-to-right order here matches the "English" ordering of SP-hybrid. SP-hybrid's "Hebrew" ordering, however, only matches the right-to-left order at P-nodes.

Let $<_L$ denote precedence in the left-to-right order. That is, $u <_L v$ means that u precedes v in the left-to-right order. Similarly, $>_R$ denotes precedence in the right-to-left order. Then the following lemma states how to perform ancestor queries.

Lemma 9.6 *Let U and V be two nodes in a tree. Then U is an ancestor of V if and only $U <_L V$ and $U <_R V$.* □

To maintain these orders dynamically, we use two order-maintenance data structures [DS87, BCD+02], as in SP-hybrid. An order maintenance data structure supports the following operations:

1. OM-PRECEDES(O, x, y): Return TRUE if x precedes y in the ordering O. Both x and y must already exist in the ordering.

2. OM-INSERT-BEFORE$(O, x, y_1, y_2, \ldots, y_k)$: In the ordering O, insert new elements $y_1, y_2, \ldots, y_k$, in that order, immediately before existing element x.

3. OM-INSERT-AFTER$(O, x, y_1, y_2, \ldots, y_k)$: In the ordering O, insert new elements $y_1, y_2, \ldots, y_k$, in that order, immediately after existing element x.

The OM-PRECEDES operation can be supported in $O(1)$ worst case time. The OM-INSERT operations can be inserted in $O(1)$ worst-case time for each node inserted.[6]

XConflict uses order-maintenance data structures O_L and O_R to maintain the left-to-right and right-to-left orderings, respectively, subject to dynamically created traces. Since these traces can be created in parallel by multiple processors, we obtain a global lock during steals. Whenever a trace U splits into three traces U_0, U_1, and U_2, we obtain the lock and perform insert operations into each ordering. (The traces U_0 and U_2 are added, whereas U_1 holds everything else that used to be part of U.) Notice that the traces U_0 and U_2 are the parent and right sibling, respectively, of the existing trace $U = U_1$.

One added difficulty is that at the time of the split, the resulting U_1 may have several descendant traces.[7] To easily deal with this fact that U_1 may have descendant's, at the time a trace is created, we insert placeholder traces to help with future trace inserts. In particular, for each trace U, we have placeholders $U^{(\ell)}$ and $U^{(r)}$ (surrounding the region of the orderings containing U and its descendant's). Thus, all descendant's of U are inserted between $U^{(\ell)}$ and $U^{(r)}$, but all other nodes are inserted outside these boundaries. These placeholders only increase the number of inserted elements by a constant factor (3), so they have no affect on the asymptotic analysis.

When U splits, we perform the following insert operations. First, to insert the parent and new parent placeholders, we perform OM-INSERT-BEFORE$(O_L, U^{(\ell)}, U_0^{(\ell)}, U_0, U_0^{(r)})$ and OM-INSERT-BEFORE$(O$ To insert the new right sibling and placeholders, we perform insert operations OM-INSERT-AFTER$(O_L, U^{(r)}, U$ and OM-INSERT-BEFORE$(O_L, U^{(\ell)}, U_2^{(\ell)}, U_2, U_2^{(r)})$. Collectively, these inserts insert 12 elements, for an amortized cost of $O(1)$.

Note that correctness of the global ancestor queries relies on Property 4—the Cilk-like work-stealing property that a thief processor steals a subtree from the highest available P-node owned by the victim.

9.11 Performance Claims

The section bounds the running time of an CWSTM program in the absence of conflicts. The bound includes the time to check for conflicts *assuming that all accesses are writes* and to maintain the relevant data structures. Checking for conflicts with multiple readers, however, increases the runtime. Additionally, aborts add more work to the computation. Those slowdowns are not included in the analysis.

The following theorem states the running time of an CWSTM program under nice conditions. We give bounds for both Cilk's normal randomized work-stealing scheduler, and for a round-robin work-stealing scheduler (as in [Fin05]).

Theorem 9.7 *Consider an CWSTM program with T_1 work and critical-path length T_∞ in which all memory accesses are writes. Suppose the program, augmented with XConflict, is executed on P and that no transaction aborts occur.*

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 $1 - \varepsilon$, for any $\varepsilon > 0$,*

[6] For our purposes, $O(1)$ amortized is sufficient for inserts. The amortized data structure [DS87, BCD+02] is easily implementable and still supports $O(1)$ worst-case queries.

[7] In contrast, the SP-hybrid algorithm essentially splits traces only at leaves in the tree of traces, so there are no descendant traces to worry about. The main reason for this difference is our different definition of traces.

2. *When using a round-robin work-stealing scheduler, the program runs in $O(T_1/P + PT_\infty)$ worst-case time.*

PROOF. The proof technique here is similar to the proof of performance of SP-hybrid in [Fin05]. The key insight in this analysis technique is to amortize the cost of updates of global-lock-protected data structures against the number of steals. One important feature of XConflict's "global" data structures is that they have $O(\#$ of traces$)$ total update cost. Another is that whenever a steal attempt occurs, the processor being stolen from is making progress on the original computation. (That is, whenever stealable, a processor performs only $O(1)$ additional work for each step of the original computation.) The proof makes the pessimistic assumption that while the global lock is held, only the processor holding the lock makes any progress.

In XConflict, an insertion into the global ancestor datastructure requires a lock, that blocks all queries to that datastructure. However, an insertion into the global ancestor datastructure happens only when traces split, which occurs only on steals. Randomized work stealing analysis guarantees that the number of steal attempts for a job with work T_1 and span T_∞ is $O(P(T_\infty + \lg(1/\epsilon))$ time with probability $1 - \epsilon$. Therefore, the total number of trace splits is $O(P(T_\infty + \lg(1/\epsilon))$ time with probability $1 - \epsilon$. We assume worst case contention and stop all processors when any steal occurs. Since the total work is T_1, time to complete this work when no processor is stealing is at most T_1/P. Therefore, the total completion time is $O(T_1/P + P(T_\infty + \lg(1/\epsilon))$. Note that all other operations on XConflict take a constant amount of time, and do not require any locks. □

One way of viewing these bounds is as 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, we expect $pT_\infty \ll T_1/P$, so these bound represents only constant-factor overheads beyond optimal. We would also expect the first bound (using the randomized scheduler) to have better constants hidden in the big-O.

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. 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.

The CWSTM design we describe does not provide any reasonable performance guarantees when we allow multiple readers. 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 proportional to the number of parallel readers to an object. It is part of future work to improve the CWSTM design and analysis in the presence of multiple readers.

9.12 Discussion

The CWSTM model presented in Section 9.5 describes one approach for implementing a software transactional memory system that supports transactions with nested fork-join parallelism. CWSTM design was guided by a few major goals.

- Supporting nested transactions of unbounded depth.
- Small overhead when there are no aborts.
- Avoid asymptotically increasing the work or the critical path of the computation too much.

We believe that we have achieved these goals to some extent, since the CWSTM guarantees provably good completion time in the case when there are no aborts, and there are no concurrent readers (or all accesses are treated as writes).

We believe that supporting unbounded nesting depth in transactions is important for composability of programs. If a function is called from inside a transaction, the caller should not have to worry about how many transactions are nested inside the function call. However, it may be true that the common case is transactions of small depth. In this case, a simpler design like the one described in Section 9.4 might be sufficient in the common case, at the expense of a slowdown in the case when the nesting depth is large. It is difficult, however, to conclude what the common case is, since there are currently few examples of programs with nested parallel transactions. An important part of future research would be to write series-parallel programs with nested transactions to understand what the common case is, and what one should optimize for.

CWSTM is a TM system design and has not been implemented yet. As described in this paper, each memory access may potentially require multiple data structure queries. CWSTM also may have a large memory footprint. Due to lazy cleanup on aborts, and fast commits, the access stack for an object may grow and require space proportional to the number of accesses to that object. Also, access stacks may contain pointers to transaction logs that persist long after the transactions are committed or aborted. Thus, a computation's memory footprint can become quite large. In practice, implementing a separate, concurrent thread for "garbage-collection" of metadata may help. As part of future work, we would like to implement the system in the Cilk runtime system to evaluate its practical performance and explore ways to optimize the implementation.

It would be interesting to see if CWSTM-like mechanisms are useful for high-performance languages like Fortress [ACH+07] and X10 [ESS05]. Both these languages support transactions and fork-join parallelism. The language specification for Fortress also permits nested parallel transactions. These are richer languages than Cilk, however, and may require more complicated mechanisms to support nested parallel transactions.

9.13 Related Work

Chapter 10

Conclusions and Future Work

Properly designed concurrency platforms can simplify parallel programming by freeing programmers from worrying about the low-level details of parallel programming. The work presented in this dissertation advances the technology in the design of concurrency platforms, primarily with respect to scheduling and synchronization for dynamic multithreaded languages. A brief summary of the contributions is as follows:

- ***Adaptive Scheduling***: Chapters 2 and 3 present automatic schedulers that adapt to jobs' changes in parallelism. This work provides a basis for building concurrency platforms where programmers need not specify how many processors must be used to run their program, thereby unburdening programmers from analyzing the parallelism of the program. In addition, these schedulers are more effective than nonadaptive schedulers for programs with irregular parallelism.

- ***Dag Evaluation***: Chapter 4 presents a concurrency platform for dag evaluations in the form of a Cilk++ library. This library allows the programmer to specify dag evaluations easily and automatically schedules these evaluations to take full advantage of both the inter-node and inter-node parallelism of these computations.

- ***Helper Locks***: Chapter 5 presents helper locks that allow programmers to express and exploit parallelism inside large critical sections. In the context of this thesis, this work relates to both scheduling and synchronization in concurrency platforms.

- ***Memory Models for TM and Open Nesting***: Chapter 6open present work on understanding the exact semantics of the synchronization mechanism of transactional memory. In particular, Chapter 6 presents the transactional computation framework and Chapter 7 presents the description of the exact semantics of open-nested transactions.

- ***Ownership-Aware Transactions***: Chapter 8 presents ownership-aware transactions, which allow programmers to get the concurrency of open-nesting without its headaches. Therefore, this work can be used as a basis of a TM concurrency platform that provides high concurrency while also providing understandable semantic guarantees.

- ***Nested Parallelism in Transactions***: Chapter 9 presents the first TM design that allows parallelism within transactions. This feature is important if TM is to be integrated into the concurrency platforms that support dynamic multithreaded languages. Again, this work is

related to both scheduling and synchronization in concurrency platforms, since the design is particularly suited for concurrency platforms that use work-stealing schedulers.

If multicores follow Moore's Law, the number of cores per chip will double every 18 months. The primary challenge that multicores face is the difficulty of parallel programming. Concurrency platforms can ease parallel programming. However, in order to gain wide acceptance, they must also provide performance competitive with hand-tuned programs. This dissertation aims to provide strong theoretical underpinnings for the design of such concurrency platforms.

Future Work

There are many aspects of parallel programming on multicores, such as ***locality*** and ***heterogeneity***, this dissertation *does not* consider. Both sequential computers and multicores have caches and accessing data from the local cache is much cheaper than accessing data from main memory. For sequential algorithms, researchers have developed both ***cache-aware*** [AV88, ACS87, BM72] and cache-oblivious [FLPR99] strategies to minimize cache misses. However, this problem is even more crucial for multicores since the memory latency may be even higher in multicores than serial computers. There has been some recent work in understanding the cache complexity of parallel algorithms [Val08, BCG+08]. However, it is just a drop in the bucket. In particular, we do not yet completely understand how scheduling decisions taken by concurrency platforms impact the cache performance.

In contrast to cache locality, which effects both sequential and parallel machine performance, heterogeneity is an issue that primarily affects parallel machines. Multicores may exhibit heterogeneity at many levels. For example, they may exhibit heterogeneous communication: some processors may be physically closer, and may therefore communicate faster with each other than with others. How a program exploits this processor locality may be an important factor in its performance. Multicores may also exhibit processor heterogeneity: different cores may have different characteristics. Ideally, concurrency platforms should hide this heterogeneity from the programmer and provide good performance by handling heterogeneity itself.

Appendices

.1 ON Model and Sequential Consistency

The transactional computation framework described in Section 6.2 admits only an a posteriori analysis of a program execution. In this section, we describe how to extend the framework to dynamically model a program execution. This dynamic model is used in Chapters 7, 8, and 9 in order to model the particular design we model in those chapters. This chapter describes the general modeling. We abstractly model the behavior of a generic transactional memory implementation as a nondeterministic state machine which transitions between states as the program executes *instructions*.

.2 The OAT Model and Sequential Consistency

This appendix contains the details of the proof of Theorem 8.7: if the OAT model generates a trace $(\mathcal{C}, \Phi)$ and a topological sort order $\mathcal{S}$, then $\mathcal{S}$ satisfies Definition 12, i.e., $\mathcal{S}$ is sequentially consistent with respect to Φ.

In this appendix, we first define some useful notation for the proof. Next, we prove that the OAT model preserves several invariants about memory operations, read set, and write sets. Finally, we use these invariants to prove Theorem 8.7.

.2.1 Notation

We define some notation that is useful later for stating operational invariants of the OAT model.

For any subset S of nodes in the computation tree $\mathcal{C}$, i.e., $S \subseteq \texttt{nodes}(\mathcal{C})$, define

- $\texttt{low}(S) = \{X \in S \ : \ \texttt{pDesc}(X) \cap S = \emptyset\}$.
- $\texttt{high}(S) = \{X \in S \ : \ \texttt{pAnces}(X) \cap S = \emptyset\}$.

Intuitively, $\texttt{low}(S)$ represents the nodes in S closest to the leaves of the tree. Similarly, $\texttt{high}(S)$ represents the nodes in S closest to the root of the tree. In cases where the set S is guaranteed to fall along one root-to-leaf path in the tree, we define $\texttt{lowest}(S)$ as the only element $X \in \texttt{low}(S)$. Similarly, we define $\texttt{highest}(S)$ as the only element in $\texttt{high}(S)$.

We also define two time-dependent sets of transactions.

- The ***reader set*** $\texttt{readers}^{(t)}(\ell) = \left\{T \in \texttt{activeX}^{(t)}(\mathcal{C}) : \ell \in \texttt{R}(t,T)\right\}$.
- The ***writer set***, $\texttt{writers}^{(t)}(\ell) = \left\{T \in \texttt{activeX}^{(t)}(\mathcal{C}) : \ell \in \texttt{W}(t,T)\right\}$.

Said differently, $\texttt{readers}^{(t)}(\ell)$ is the set of active transactions at time t which have location ℓ in their read set. Similarly, $\texttt{writers}^{(t)}(\ell)$ is the set of active transactions at time t with $\ell \in \texttt{W}(T)$.

Next, we generalize the content sets from Definition 23 and define a set of dynamic content sets.

Definition 38 *At any time t, for any transaction $T \in \texttt{xactions}(t,\mathcal{C})$ and a memory operation $u \in \texttt{memOps}(t,\mathcal{C})$, define the sets cContent(t,T), $\texttt{oContent}^{(t)}(T)$, $\texttt{aContent}(t,T)$, and $\texttt{vContent}(t,T)$ according the $\texttt{ContentType}(t,u,T)$ procedure:*

```
   ContentType(t,u,T)          ▷ For any u ∈ memOps(t,T)
1  X ← xparent[u]
2  while (X ≠ T)
3     if X ∈ activeX^(t)(C),        return u ∈ vContent(t,T)
4     if X ∈ aborted(t,C),          return u ∈ aContent(t,T)
5     if (X = committer(u))         return u ∈ oContent^(t)(T)
6     X ← xparent[X]
7  return u ∈ cContent(t,T)
```

The difference between Definition 38 and the previous statement in Definition 23 is that for dynamic content sets, if we encounter a PENDING or PENDING_ABORT transaction when walking up the tree from a memory operation u to a transaction T, we place u in the *active content* of T, i.e., $u \in \texttt{vContent}(t, T)$. If a transaction T completes at time t^*, it is not hard to see that the dynamic classification $\texttt{ContentType}(t, u, T)$ gives the same answer as the static classification $\texttt{ContentType}(u, T)$ for all times $t \geq t^*$.

.2.2 OAT Model Invariants

Because the OAT model performs eager conflict detection according to Definition 21, it is not hard to prove the following invariant about the readers and writers to a particular memory location ℓ.

Theorem .1 *At all times t, for all memory locations $\ell \in \mathcal{M}$, the OAT maintains the following invariants on the sets* $\texttt{readers}(\ell)$ *and* $\texttt{writers}(\ell)$*:*

1. *For all* $\ell \in \mathcal{M}$, $\left|\texttt{low}(\texttt{writers}^{(t)}(\ell))\right| = 1$, *i.e.,* $\texttt{lowest}(\texttt{writers}^{(t)}(\ell))$ *exists.*

2. *For any* $T \in \texttt{readers}^{(t)}(\ell)$, *either* $\texttt{lowest}(\texttt{writers}^{(t)}(\ell)) \in \texttt{desc}(T)$ *or* $T \in \texttt{desc}(\texttt{lowest}(\texttt{writers}^{(t)}$

PROOF. The proof is by induction on the instructions that the OAT model issues.

In the base case, for all locations $\ell \in \mathcal{M}$, we begin with $\texttt{readers}^{(0)}(\ell) = \texttt{writers}^{(0)}(\ell) = \{\texttt{root}(\mathcal{C})\}$, and no other nodes in the computation tree $\mathcal{C}$ except $\texttt{root}(\mathcal{C})$. Thus, Invariants 1 and 2 are satisfied.

In the inductive step, suppose at time $t-1$, Invariants 1 and 2 are satisfied. A `read` or `write` instruction at time t can not break the invariants without causing a conflict according to Definition 21. 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` instruction that commits a transaction T can only add the transaction $\texttt{xparent}[T]$ to $\texttt{readers}(\ell)$ or $\texttt{writers}(\ell)$. Since $\texttt{xparent}[T]$ is an ancestor of T, it can not break either of the two invariants.

The remaining instructions preserve Invariants 1 and 2 trivially. A `fork` or `join` instruction at time t preserves the invariants because they do not change the set active transactions or any transaction read sets or write sets. An `xbegin` preserves the invariants because it creates new transactions T with empty read sets and write sets. The `xabort` instruction preserves the invariants because it can only remove transactions from $\texttt{readers}^{(t)}(\ell)$ or $\texttt{W}(t, \ell)$. □

The following invariant shows that, informally, the read sets of transactions act as caches for pairs (ℓ, u) stored in write sets.

Lemma .2 *At any time t, for any $T \in \texttt{readers}^{(t)}(\ell)$, suppose $(\ell, u) \in \texttt{R}(t, T)$. Let $T' = \texttt{lowest}(\texttt{xAnces}(T) \cap \texttt{writers}^{(t)}(\ell))$. Then $(\ell, u) \in \texttt{W}(t, T')$.*

PROOF. The proof is by induction on the instructions issued by the OAT model. In the base case, we know for all memory locations $\ell \in \mathcal{M}$,, we start with $\texttt{readers}^{(0)}(\ell) = \texttt{writers}^{(0)}(\ell) = \{\texttt{root}(\mathcal{C})\}$ and $\texttt{R}(\texttt{root}(\mathcal{C})) = \texttt{W}(\texttt{root}(\mathcal{C}))$. Since $T' = T = \texttt{root}(\mathcal{C})$, Lemma .2 is satisfied in the base case.

For the inductive step, assume the lemma is satisfied at time $t-1$. We show after any S-node X issues an instruction at time t, the lemma is still satisfied.

For any $T \in \texttt{xactions}(t-1, \mathcal{C})$, after a `fork`, `join`, or `xbegin` instruction in step t, we have $\texttt{R}(t,T) = \texttt{R}(t-1,T)$ and $\texttt{W}(t,T) = \texttt{W}(t-1,T)$. Thus, the lemma is satisfied after these instructions. An `xbegin` which creates a new transaction X at time step t starts with $\texttt{R}(t,X) = \texttt{W}(t,X) = \emptyset$; thus, the lemma is satisfied.

Next, consider an `xabort` issued by $X \in \texttt{xactions}(t-1, \mathcal{C})$. Suppose, before the `xabort` of X there exists a transaction $T \in \texttt{readers}^{(t-1)}(\ell)$ with $(\ell, u) \in \texttt{R}(t-1, T)$. Let $T' = \texttt{lowest}(\texttt{xAnces}(T) \cap \texttt{writers}^{(t-1)}(\ell))$. Then before the `xabort`, $(\ell, u) \in \texttt{W}(t-1, T')$. Assume for contradiction after the `xabort` of X, that there exists some transaction $T \in \texttt{xactions}(t, \mathcal{C})$ such that the invariant no longer holds for T, i.e., we no longer have $(\ell, u) \in \texttt{W}(t, T')$. Since an `xabort` does not change the contents of any transaction's write set, but removes X from $\texttt{writers}(\ell)$, the only way to violate the invariant is if $X = T'$. Consider two cases: either $X = T' = T$, or $X = T' \neq T$. In the first case, we can not violate the invariant for T because T is aborted and removed from $\texttt{readers}(\ell)$. In the second case, we must have $T \in \texttt{pDesc}(X)$. But then, before the `xabort`, we have $T \in \texttt{pDesc}(X) \cap \texttt{activeN}(t-1, \mathcal{C})$ and $X \in \texttt{ready}(t-1, \mathcal{C})$, contradicting the property that the ready nodes are the leaves of tree of active nodes. Thus, the `xabort` must preserve the invariant.

A successful `read` operation v observes the value from the closest transactional ancestor X which has location ℓ in its read set. The only transaction whose read set changes is $\texttt{xparent}[v]$. The invariant is preserved because $\texttt{xAnces}(\texttt{xparent}[v]) \supseteq \texttt{xAnces}(X)$, and since the `read` does not change any write sets.

A successful `write` operation v only changes the write set of $\texttt{xparent}[v]$; this `write` can not break the invariant without generating a conflict.

Finally, suppose at time t, a ready node X issues an `xend`. Consider two cases:

1. $X \neq \texttt{owner}(\ell)$. The only transaction Y which has its read set or write set change after the `xend` (i.e., for which we could have $\texttt{R}(t,Y) \neq \texttt{R}(t-1,Y)$ or $\texttt{W}(t,Y) \neq \texttt{W}(t-1,Y)$) is $Y = \texttt{xparent}[X]$. The `xend` merges X's read and write sets into Y's read and write sets, respectively; using Theorem .1, it is straightforward to show that the invariant is preserved for Y.

   For all other transactions $T \in \texttt{readers}^{(t)}(\ell)$ with $T \neq Y$, since the read set or write set of T or any transaction in $\texttt{xAnces}(T)$ remains the same, the invariant is still preserved for T.

2. Suppose $X = \texttt{owner}(\ell)$. Then, the only transaction whose read set or write set can change is $Y = \texttt{root}(\mathcal{C})$. But the only way to break the invariant is if X commits a pair (ℓ, v) from $\texttt{W}(t-1, X)$ to $\texttt{root}(\mathcal{C})$, which corrupts the version $(\ell, u) \in \texttt{R}(t-1, T)$, for some transaction T parallel to X. But then, we would violate Theorem .1, and should have had a conflict earlier.

Since all possible choices for action $k+1$ preserve the invariant, the lemma holds by induction. □

Theorem .3 characterizes when a transaction should have a location in its write set.

Theorem .3 *At any time t, consider any transaction $T \in \texttt{activeX}^{(t)}(\mathcal{C})$ and any memory location ℓ such that $\texttt{xid}(\texttt{owner}(\ell)) \leq MT$. Let $S_\ell(t) = \{u \in \texttt{memOps}(t, \mathcal{C}) : W(u, \ell)\}$. Exactly one of the following cases holds:*

1. $T = \texttt{root}(\mathcal{C})$, $(\ell, \bot) \in \texttt{W}(t, T)$, *and two conditions are satisfied:*
   (a) *cContent*$(t, T) \cap S_\ell = \emptyset$.
   (b) *For all* $v \in S_\ell(t)$, *we have* $v \in \texttt{aContent}(t, T) \cup \texttt{vContent}(t, T)$.
2. *There exists an* $(\ell, u) \in \texttt{W}(t, T)$ *which happens at time* t_u, *and two conditions are satisfied:*
   (a) $u \in$ *cContent*$(t, T) \cap S_\ell(t)$
   (b) *For any operation* $v \in (S_\ell(t) - \{u\})$ *which happens at time* t_v, *where* $t_u < t_v \leq t$, *we have* $v \in \texttt{aContent}(t, T) \cup \texttt{vContent}(t, T)$.
3. *We have* $\ell \notin \texttt{W}(t, T)$, *and* *cContent*$(t, T) \cap S_\ell(t) = \emptyset$.

PROOF.

This theorem can be proved by a straighforward, albeit tedious, induction on time.

Note that because we assume $\texttt{xid}(\texttt{owner}(\ell)) \leq MT$, $S_\ell(t) \cap \texttt{oContent}^{(t)}(T) = \emptyset$, i.e., the theorem is only concerned with memory locations ℓ which belong to T's open content. Because of the properties of ownership and Xmodules, any location ℓ with $\texttt{xid}(\texttt{owner}(\ell)) > MT$ can never propagate into T's write set anyway. □

The intution for Theorem .3 lies mostly in Case 2; if at time t a pair (ℓ, u) is the write set of a transaction T, then u is the last write to ℓ in T's subtree which is "committed with respect to" T. Any v which writes to ℓ after t_u (the time u occurs) must belong to T's subtree; otherwise, there would have been a conflict. Furthermore, any v which happens after t_u must still be aborted or pending with respect to T (i.e., $v \in \texttt{aContent}(t, T) \cup \texttt{vContent}(t, T)$); otherwise, v should replace u in T's write set.

Case 3 says the write set of T does not contain a location ℓ if no memory operation in T's subtree commits ℓ to T. Case 1 of Theorem .3 handles the special case of the root.

.2.3 Proof of Sequential Consistency

Finally, we can use the invariants from Lemma .2 and Theorem .3 to prove Theorem 8.7.

PROOF. [Theorem 8.7]

The first condition and second conditions are true by construction, since the OAT 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 and fourth conditions, we require some setup. Suppose at time t_v, memory operation v happens and the OAT model sets $\Phi(v) = u$. Let $A = \texttt{lowest}(\texttt{readers}^{(t)}(\ell) \cap \texttt{ances}(v))$. Because the OAT model sets $\Phi(v) = u$, we must have $(\ell, u) \in \texttt{R}(t, A)$. Let $T = \texttt{lowest}(\texttt{xAnces}(A) \cap \texttt{writers}^{(t)}(\ell))$. By Lemma .2, we know $(\ell, u) \in \texttt{W}(t, T)$. By Theorem .3, since $(\ell, u) \in \texttt{W}(t, T)$, we know $u \in$ *cContent*(t, T). Let $X = \texttt{xLCA}(u, v)$. We must have $T \in \texttt{ances}(X)$; otherwise, we could not have $\{u, v\} \subseteq \texttt{memOps}(t, T)$.

Since $u \in$ *cContent*(t, T), we know $u \in$ *cContent*$(t, X) \cup \texttt{oContent}^{(t)}(X)$. Therefore, we have $\neg(uHv)$, satisfying the third condition.

To check the fourth condition, assume for contradiction that there exists a w such that $W(w, \ell)$, and $u <_S w <_S v$. Let t_v be the time that v happens. Then, since $\Phi(v) = u$, we know $u \in \texttt{W}(t_v, T)$. Therefore, by Theorem .3 we know $w \in \texttt{memOps}(t_v, T)$, $w \in \texttt{aContent}(t_v, T) \cup \texttt{vContent}(t_v, T)$.

Let $Y = \texttt{xLCA}(w, v)$. Since $w \in \texttt{memOps}(t_v, T)$, we know $T \in \texttt{ances}(Y)$. Consider the two cases for w:

1. Suppose $w \in \texttt{aContent}(t_v, T)$. Since $T \in \texttt{ances}(Y)$, we know $w \in \textit{cContent}(t_v, Y) \cup \texttt{aContent}(t_v, Y)$.

   We can show by contradiction that we must have $w \in \texttt{aContent}(t_v, Y)$. If $Y = T$, then we already know $w \in \texttt{aContent}(t_v, Y)$. Otherwise, assume $T \in \texttt{pAnces}(Y)$. If we had $w \in \textit{cContent}(t_v, Y)$, then by Theorem .3, we must have $(\ell, y) \in \texttt{W}(t_v, Y)$. This statement contradicts the fact that OAT model found (ℓ, u) from transaction T, since a closer transaction Y had ℓ in its read set.

   But then, since $w \in \texttt{aContent}(t_v, Y)$, we have wHv.

2. Suppose $w \in \texttt{vContent}(t_v, T)$:

   Then, we know $w \in \textit{cContent}(t_v, Y) \cup \texttt{vContent}(t_v, Y)$. As in the previous case, we can show $w \notin \textit{cContent}(t_v, Y)$.

   If $w \in \texttt{vContent}(t_v, Y)$, then there exists some transaction $Z \in \texttt{activeX}^{(t_v)}(Y) - \{Y\}$ such that $\ell \in \texttt{W}(t_v, Z)$.

   Since $w \in \texttt{memOps}(t_v, Z)$, we can strengthen this condition to $Z \in \texttt{activeX}^{(t_v)}(\texttt{LCA}(w, v)) - \{\texttt{LCA}(w, v)\}$. This statement leads to a contradiction, however, because $w \in \texttt{W}(t_v, Z)$ must conflict with v.

   More formally, by Invariant 2 of Theorem .1, any new read operation v at time t_v must satisfy $v \in \texttt{desc}(\texttt{low}(\texttt{writers}^{(t_v)}(\ell)))$ (i.e., v is a descendant of the base of the spine for ℓ). At time t_v, however, we must have $\texttt{low}(\texttt{writers}^{(t_v)}(\ell)) \in \texttt{desc}(Z)$.

□

.3 Rules for Type Checking the OAT Type System

This appendix contains the type rules for the OAT type system. The grammar for the type system is shown below.

$$
\begin{array}{rcl}
P & ::= & defn^*\ e \\
defn & ::= & \textsf{class}\ ocn\langle formal+\rangle\ \textsf{extends}\ oc \\
 & & \textsf{where}\ constr^*\ \{field^*\ meth^*\}\ \mid \\
 & & \textsf{class}\ xcn\langle formal+\rangle\ \textsf{extends}\ xc \\
 & & \textsf{where}\ constr^*\ \{xfield^*\ meth^*\} \\
oc & ::= & ocn\langle owner+\rangle\ \mid\ \textsf{Object}\langle owner\rangle \\
xc & ::= & xcn\langle owner+\rangle\ \mid\ \textsf{Xmodule}\langle owner\rangle \\
owner & ::= & \textsf{world}[i]\ \mid\ formal\ \mid\ \textsf{this}[i] \\
constr & ::= & (owner \rhd owner)\ \mid\ (owner \not\rhd owner)\ \mid \\
 & & (owner = owner)\ \mid\ (owner \neq owner) \\
meth & ::= & t\ mn\langle formal^*\rangle(arg^*)\ \textsf{where}\ constr^*\{e\} \\
field & ::= & t\ fd \\
xfield & ::= & c\ fd \\
arg & ::= & t\ x \\
t & ::= & c\ \mid\ \textsf{int} \\
formal & ::= & f \\
e & ::= & \textsf{new}\ c\ \mid\ x\ \mid\ x = e\ \mid \\
 & & \textsf{let}\ (arg = e)\ \textsf{in}\ \{e\}\ \mid \\
 & & x.fd\ \mid\ x.fd = y\ \mid\ x.mn\langle owner^*\rangle(y^*)
\end{array}
$$

$$
\begin{array}{rcl}
ocn & \in & \text{class names that are not subtype of Xmodule} \\
xcn & \in & \text{class names that are subtype of Xmodule} \\
fd & \in & \text{field names} \\
mn & \in & \text{method names} \\
x, y & \in & \text{variable names} \\
f & \in & \text{owner names} \\
i, j & \in & \text{type int literals}
\end{array}
$$

We define a number of predicates used in the type system. These predicates are adapted from [BLS03], but our type system does not handle inner classes for now.

Predicate	Meaning
WFClasses(P)	There are no cycles in the class hierarchy
ClassOnce(P)	No class is declared twice in P
FieldsOnce(P)	No class contains two fields, decalred or inherited with the same name
MethodsOnce(P)	No class contains two methods with the same name
OverridesOK(P)	Overriding methods have the same return type and parameter types as the methods being overridden.
WorldInMainOnly(P)	Only the main method uses the world tag to initialize owner.
ThisInXcOnly(P)	Only classes that are subtype of Xmodule use this tag to initialize owner.

Our typing judgment follows the form adapted from [BLS03]: $P;\ E\ \vdash\ e : t$, where P is the program being checked to provide information about class definitions; E is an environment providing type information for the free variables in e; finally, t is the type of e.

The typing environment is defined as

$E ::= \emptyset\ \mid\ E,\ t\ x\ \mid\ E,\ \text{owner}\ f\ \mid\ E,\ constr$

The typing environtment contains the the declared types of variables, the decalred owner parameters, the declared constraints among owners, and certain inferred constraints, such as this[i] = this[j] when they are used in a Xmodule class definition.

The typing system uses the following judgments.

Judgment	Meaning
$\vdash\ P : t$	program P yields type t
$P \vdash\ defn$	$defn$ is a well-formed class
$P;\ E\ \vdash\ constr$	constraint $constr$ is satisfied
$P;\ E\ \vdash\ (o_1 = o_2)$	o_1 and o_2 represent the same owner instance
$P;\ E\ \vdash_{\text{owner}}\ o$	o is an owner
$P;\ E\ \vdash\ wf$	typing environment E is well-formed
$P;\ E\ \vdash\ t$	t is a well-formed type
$P;\ E\ \vdash\ t_1 <: t_2$	t_1 is a subtype of t_2
$P;\ E\ \vdash\ t_1 <:= t_2$	t_2 is assignable to t_1
$P\ \vdash\ xfield \in xc$	Xmodule class xc declares/inherits $xfield$
$P\ \vdash\ field \in oc$	non-Xmodule class oc declares/inherits $field$
$P;\ E\ \vdash\ field$	$field$ is a well-formed field
$P\ \vdash\ meth \in xc$	Xmodule class xc declares/inherits $meth$
$P\ \vdash\ meth \in oc$	non-Xmodule class oc declares/inherits $meth$
$P;\ E\ \vdash\ meth$	$meth$ is a well-formed method
$P;\ E\ \vdash\ e : t$	expression e has type t

We present the type rules for these judgments in the following pages.

$\boxed{\vdash P : t}$

[PROG]

$$\frac{\begin{array}{c} WFClasses(P) \\ ClassOnce(P) \quad FieldsOnce(P) \quad MethodsOnce(P) \quad OverridesOK(P) \\ WorldInMainOnly(P) \\ ThisInXcOnly(P) \quad P = defn_{1..n}\ e \quad P \vdash defn_i \quad P;\ \emptyset \vdash e : t \end{array}}{\vdash P : t}$$

$\boxed{P \vdash defn}$

[CLASS]

$$\frac{\begin{array}{c} E = ocn\langle f_{1..n}\rangle \text{ this, owner } f_{1..n},\ f_1 \rhd f_i,\ constr^* \\ P;\ E \vdash wf \quad P;\ E \vdash oc' \quad P;\ E \vdash field_i \quad P;\ E \vdash meth_i \end{array}}{P \vdash \textsf{class}\ ocn\langle f_{1..n}\rangle\ \textsf{extends}\ oc'\ \textsf{where}\ constr^*\ \{field^*\ meth^*\}}$$

[XMODULE CLASS]

$$\frac{\begin{array}{c} E = xcn\langle f_{1..n}\rangle \text{ this, owner } f_{1..n},\ f_1 \rhd f_i,\ constr^* \\ P;\ E \vdash wf \quad P;\ E \vdash xc' \quad P;\ E \vdash field_i \quad P;\ E \vdash meth_i \end{array}}{P \vdash \textsf{class}\ xcn\langle f_{1..n}\rangle\ \textsf{extends}\ xc'\ \textsf{where}\ constr^*\ \{xfield^*\ meth^*\}}$$

$\boxed{P;\ E \vdash constr}$

[CONSTR ENV]

$$\frac{E = E_1,\ constr,\ E_2}{P;\ E \vdash constr}$$

[$\rhd$ WORLD]

$$\frac{P;\ E \vdash_{\text{owner}} o}{P;\ E \vdash (o \rhd \textsf{world})}$$

[$\rhd$ OWNER]

$$\frac{P;\ E \vdash e : xn\langle o_{1..n}\rangle}{P;\ E \vdash (e \rhd o_1)}$$

[$\rhd$ REFL]

$$\frac{P;\ E \vdash_{\text{owner}} o}{P;\ E \vdash (o \rhd o)}$$

[$\rhd$ TRAN

$$\frac{P;\ E \vdash}{P;\ E \vdash}$$

$\boxed{P;\ E \vdash (o_1 = o_2)}$

[= OWNER]

$$\frac{E = E_1,\ xc\textsf{this},\ E_2}{P;\ E \vdash (this[i] = this[j])}$$

[= REFL]

$$\frac{P;\ E \vdash_{\text{owner}} o}{P;\ E \vdash (o = o)}$$

[= TRANS]

$$\frac{\begin{array}{c} P;\ E \vdash (o_1 = o_2) \\ P;\ E \vdash (o_2 = o_3) \end{array}}{P;\ E \vdash (o_1 = o_3)}$$

$\boxed{P;\ E \vdash_{\text{owner}} o}$

[OWNER WORLD] [OWNER FORMAL] [OWNER THIS]

$$\frac{}{P;\ E \vdash_{\text{owner}} \textsf{world}} \qquad \frac{E = E_1,\ \text{owner}\ f,\ E_2}{P;\ E \vdash_{\text{owner}} f} \qquad \frac{E = E_1,\ xc\ \textsf{this},\ E_2}{P;\ E \vdash_{\text{owner}} \textsf{this[i]}}$$

$\boxed{P;\ E \vdash wf}$

[ENV ∅] [ENV X] [ENV OWNER] [ENV CONSTR]

$$\frac{}{P;\ \emptyset \vdash wf} \qquad \frac{\begin{array}{c}P;\ E \vdash t \\ x \notin \text{Dom}(E) \\ P;\ E \vdash wf\end{array}}{P;\ E,\ t\ x \vdash wf} \qquad \frac{\begin{array}{c}f \notin \text{Dom}(E) \\ P;\ E \vdash wf\end{array}}{P;\ E,\ \text{owner}\ f \vdash wf} \qquad \frac{\begin{array}{c}constr = (o \rhd o') \ \vee\ constr = \\ P;\ E \vdash wf \qquad P;\ E \vdash_{\text{owner}} o,\ o' \\ \nexists_{x,y}\ (P;\ E' \vdash x \rhd y)\ \wedge\ (P;\ E'\end{array}}{P;\ E,\ constr \vdash wf}$$

$\boxed{P;\ E \vdash t}$

[TYPE INT] [TYPE OBJECT] [TYPE OC]

$$\frac{}{P;\ E \vdash \textsf{int}} \qquad \frac{P;\ E \vdash_{\text{owner}} o}{P;\ E \vdash \textsf{Object}\langle o\rangle} \qquad \frac{\begin{array}{c}P \vdash \textsf{class}\ ocn\langle f_{1..n}\rangle \ldots \textsf{where}\ constr^* \ldots \\ P;\ E \vdash_{\text{owner}} o_i \qquad P;\ E \vdash o_1 \rhd o_i \qquad P;\ E \vdash constr\ [o_1/f_1\end{array}}{P;\ E \vdash ocn\langle o_{1..n}\rangle}$$

[TYPE XMODULE] [TYPE XC]

$$\frac{P;\ E \vdash_{\text{owner}} o}{P;\ E \vdash \textsf{Object}\langle o\rangle} \qquad \frac{\begin{array}{c}P \vdash \textsf{class}\ xcn\langle f_{1..n}\rangle \ldots \textsf{where}\ constr^* \ldots \\ P;\ E \vdash_{\text{owner}} o_i \qquad P;\ E \vdash o_1 \rhd o_i \qquad P;\ E \vdash constr\ [o_1/f_1]..[o_n/f_n]\end{array}}{P;\ E \vdash xcn\langle o_{1..n}\rangle}$$

$\boxed{P;\ E \vdash t_1 <: t_2}$

[SUBTYPE REFL] [SUBTYPE TRANS]

$$\frac{P;\ E \vdash t}{P;\ E \vdash t <: t} \qquad \frac{\begin{array}{c}P;\ E \vdash t_1 <: t_2 \\ P;\ E \vdash t_2 <: t_3\end{array}}{P;\ E \vdash t_1 <: t_3}$$

[SUBTYPE XC]

$$\frac{P;\ E \vdash xcn\langle o_{1..k..n}\rangle \qquad P \vdash \textsf{class}\ xcn\langle f_{1..k..n}\rangle\ \textsf{extends}\ xcn'\langle f_1\ o_{2..k}\rangle\ \ldots}{P;\ E \vdash xcn\langle o_{1..k..n}\rangle\ <:\ xcn'\langle f_1\ o_{2..k}\rangle\ [o_1/f_1]..[o_n/f_n]}$$

[SUBTYPE OC]

$$\frac{P;\ E \vdash ocn\langle o_{1..k..n}\rangle \qquad P \vdash \textsf{class}\ ocn\langle f_{1..k..n}\rangle\ \textsf{extends}\ ocn'\langle f_1\ }{P;\ E \vdash ocn\langle o_{1..k..n}\rangle\ <:\ ocn'\langle f_1\ o_{2..k}\rangle\ [o_1/f_1}$$

$\boxed{P;\ E \vdash t_1\ <:=\ t_2}$

[ASSIGNABILITY REFL]

$$\frac{P;\ E \vdash t}{P;\ E \vdash t\ <:=\ t}$$

[ASSIGNABILITY TRANS]

$$\frac{P;\ E \vdash t_1\ <:=\ t_2 \qquad P;\ E \vdash t_2\ <:=\ t_3}{P;\ E \vdash t_1\ <:=\ t_3}$$

[ASSIGNABILITY FOR XC]

$$\frac{P;\ E \vdash xcn\langle o_{1..n}\rangle \qquad P;\ E \vdash xcn\langle o'_{1..n}\rangle \qquad P;\ E \vdash (o_i = o'_i)^{i\in 1..n} \qquad P;\ E \vdash (o_i \rhd o'_i)^{i\in 1..n}}{P;\ E \vdash xcn\langle o_{1..n}\rangle\ <:=\ xcn\langle o'_{1..n}\rangle}$$

[ASSIGNABILITY FOR OC]

$$\frac{P;\ E \vdash ocn\langle o_{1..n}\rangle \qquad P;\ E \vdash ocn\langle o'_{1..n}\rangle \qquad P;\ E \vdash (o_i = o'_i)^{i\in 1..n} \qquad P;\ E \vdash (o_i \rhd o'_i)^{i\in 1..n}}{P;\ E \vdash ocn\langle o_{1..n}\rangle\ <:=\ ocn\langle o'_{1..n}\rangle}$$

$\boxed{P \vdash xfield \in xc}$

[XFIELD DECLARED]

$$\frac{P \vdash \textsf{class}\ xcn\langle f_{1..n}\rangle\ldots\ \{\ldots xfield\ldots\}}{P \vdash xfield \in xcn\langle f_{1..n}\rangle}$$

[XFIELD INHERITED]

$$\frac{P \vdash xfield \in xcn\langle f_{1..n}\rangle \qquad P \vdash \textsf{class}\ xcn'\langle g_{1..m}\rangle\ \textsf{extends}\ xcn\langle o_{1..n}\rangle\ldots}{P \vdash xfield\,[o_1/f_1]..[o_n/f_n] \in xcn'\langle g_{1..m}\rangle}$$

$\boxed{P \vdash field \in oc}$

[FIELD DECLARED]

$$\frac{P \vdash \textsf{class}\ ocn\langle f_{1..n}\rangle\ldots\ \{\ldots field\ldots\}}{P \vdash field \in ocn\langle f_{1..n}\rangle}$$

[FIELD INHERITED]

$$\frac{P \vdash field \in ocn\langle f_{1..n}\rangle \qquad P \vdash \textsf{class}\ ocn'\langle g_{1..m}\rangle\ \textsf{extends}\ ocn\langle o_{1..n}\rangle\ldots}{P \vdash field\,[o_1/f_1]..[o_n/f_n] \in ocn'\langle g_{1..m}\rangle}$$

$\boxed{P;\ E \vdash}$

[FIELD]

$$\frac{P;\ E \vdash}{P;\ E \vdash t}$$

$\boxed{P \vdash meth \in xc}$

[METHOD DECLARED IN XC]

$$\frac{P \vdash \textsf{class}\ xcn\langle f_{1..n}\rangle\ldots\ \{\ldots meth\ldots\}}{P \vdash meth \in xcn\langle f_{1..n}\rangle}$$

[METHOD INHERITED BY XC]

$$\frac{P \vdash meth \in xcn\langle f_{1..n}\rangle \qquad P \vdash \textsf{class}\ xcn'\langle g_{1..m}\rangle\ \textsf{extend}}{P \vdash meth\,[o_1/f_1]..[o_n/f_n] \in xcn'\langle g_{1..m}\rangle}$$

$\boxed{P \vdash meth \in oc}$

[METHOD DECLARED IN OC]

$$\frac{P \vdash \textsf{class}\ ocn\langle f_{1..n}\rangle ... \{... meth ...\}}{P \vdash meth \in ocn\langle f_{1..n}\rangle}$$

[METHOD INHERITED BY OC]

$$\frac{P \vdash meth \in ocn\langle f_{1..n}\rangle \qquad P \vdash \textsf{class}\ ocn'\langle g_{1..m}\rangle\ \textsf{extends}}{P \vdash meth\,[o_1/f_1]..[o_n/f_n] \in ocn'\langle g_{1..m}\rangle}$$

$\boxed{P;\ E \vdash meth}$

[METHOD]

$$\frac{\begin{array}{c}E' = E, \textsf{owner}\ f_{1..n},\ constr^*,\ arg^* \\ P;\ E' \vdash wf \qquad P;\ E' \vdash e : t\end{array}}{P;\ E \vdash t\ mn\langle f_{1..n}\rangle(arg^*)\ \textsf{where}\ constr^*\ \{e\}}$$

$\boxed{P;\ E \vdash e : t}$

[EXP TYPE]

$$\frac{P;\ E \vdash t}{P;\ E \vdash e : t}$$

[EXP SUB]

$$\frac{P;\ E \vdash e : t' \qquad P;\ E \vdash t' <: t}{P;\ E \vdash e : t}$$

[EXP NEW]

$$\frac{P;\ E \vdash c}{P;\ E \vdash c : c}$$

$\boxed{P;\ E \vdash e : t}$

[EXP LET]

$$\frac{\begin{array}{c}arg = t_1\,x \qquad P;\ E \vdash e_1 : t_1' \\ P;\ E \vdash t_1 <:= t_1' \qquad P;\ E,\ arg \vdash e_2 : t_2\end{array}}{P;\ E \vdash \textsf{let}\ (arg = e_1)\ \textsf{in}\ \{e_2\} : t_2}$$

[EXP VAR]

$$\frac{E = E_1,\ t\,x,\ E_2}{P;\ E \vdash x : t}$$

[EXP VAR ASSIGN]

$$\frac{\begin{array}{c}P;\ E \vdash x : t \qquad P;\ E \vdash e : t' \\ P;\ E \vdash t <:= t'\end{array}}{P;\ E \vdash x = e : t}$$

[EXP REF]

$$\frac{\begin{array}{c}P;\ E \vdash x : cn\langle o_{1..n}\rangle \\ P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle\end{array}}{P;\ E \vdash x.fd : t\,[o_1/f_1]..[o_n/f_n]}$$

[EXP REF ASSIGN]

$$\frac{\begin{array}{c}P;\ E \vdash x : cn\langle o_{1..n}\rangle \qquad P \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \\ P;\ E \vdash y : t'\,[o_1/f_1]..[o_n/f_n] \qquad P;\ E \vdash t <:= t'\end{array}}{P;\ E \vdash x.fd = y : t\,[o_1/f_1]..[o_n/f_n]}$$

[EXP INVOKE]

$$\frac{\begin{array}{c}P \vdash (t\ mn\langle f_{(n+1)..m}\rangle(t_j\ y_j\ ^{j\in 1..k})\ \textsf{where}\ constr^*\ ...) \in cn\langle f_{1..n}\rangle \\ P;\ E \vdash x : cn\langle o_{1..n}\rangle \qquad P;\ E \vdash x_j : t_j'\,[o_1/f_1]..[o_m/f_m] \\ P;\ E \vdash o_1 \rhd o_i \qquad P;\ E \vdash constr\,[o_1/f_1]..[o_m/f_m] \qquad P;\ E \vdash (t_j <:= t_j')^{j\in 1..k}\end{array}}{P;\ E \vdash x.mn\langle o_{(n+1)..m}\rangle(x_{1..k}) : t\,[o_1/f_1]..[o_m/f_m]}$$

Bibliography

[AAK+05] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie, *Unbounded transactional memory*, Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA) (San Francisco, California), February 2005, pp. 316–327.

[ABB00] Umar A. Acar, Guy E. Blelloch, and Robert D. Blumofe, *The data locality of work stealing*, Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA) (Bar Harbour, Maine), 2000, pp. 1–12.

[ABP98] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton, *Thread scheduling for multiprogrammed multiprocessors*, Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA) (Puerto Vallarta, Mexico), June 1998, pp. 119–129.

[ACH+07] Eric Allen, David Chase, Joe Hllett, Victor Luchango, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt, *The Fortress language specification, version 1.0 β*, Tech. report, Sun Microsystems, Inc., March 2007.

[ACS87] Alok Aggarwal, Ashok K. Chandra, and Marc Snir, *Hierarchical memory with block transfer*, Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS) (Los Angeles, California), IEEE, 12–14 October 1987, pp. 204–216.

[AFS08] Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha, *Nested parallelism in transactional memory*, Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) (Salt Lake City, Utah, USA), February 2008.

[AHHL06] Kunal Agrawal, Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson, *Adaptive task scheduling with parallelism feedback*, Proceedings of the Annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) (New York City, NY, USA), March 2006.

[AHHL08] ———, *Adaptive work stealing with parallelism feedback*, ACM Transactions on Computer Systems **26** (2008), no. 3.

[AHL06] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson, *An empirical evaluation of work stealing with parallelism feedback*, Proceedings of the International Conference on Distributed Computing Systems (ICDCS) (Lisboa, Portugal), July 2006.

[AHL07] ———, *Adaptive work stealing with parallelism feedback*, Proceedings of the Annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) (San Jose, CA), March 2007.

[AHS94] James Aspnes, Maurice Herlihy, and Nir Shavit, *Counting networks*, Journal of the ACM **41** (1994), no. 5, 1020–1048.

[ALS06] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha, *Memory models for open-nested transactions*, Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC), October 2006, In conjunction with International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).

[ALS08] Kunal Agrawal, I-Ting Angelina Lee, and Jim Sukha, *Safe open-nested transactions through ownership*, Tech. Report MIT-CSAIL-TR-2008-038, MIT CSAIL, June 2008, Available online at `http://supertech.csail.mit.edu/~angelee/safeTech.pdf`.

[ALS09] ———, *Safe open-nested transactions through ownership*, Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) (Raleigh, NC, USA), February 2009.

[Art09] Cilk Arts, *Cilk++ programmer's guide*, `http://www.cilk.com`, April 2009.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter, *The input/output complexity of sorting and related problems*, Communications of the ACM **31** (1988), no. 9, 1116–1127.

[Bar93] Greg Barnes, *A method for implementing lock-free shared data structures*, Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA) (Velen, Germany), June 1993, pp. 261–270.

[BBG89] Catriel Beeri, Philip A. Bernstein, and Nathan Goodman, *A model for concurrency in nested transactions systems*, Journal of the ACM **36** (1989), no. 2, 230–269.

[BCD+02] Michael A. Bender, Richard Cole, Erik Demaine, Martin Farach-Colton, and Jack Zito, *Two simplified algorithms for maintaining order in a list*, Proceedings of the European Symposium on Algorithms (ESA) (Rome, Italy), 2002, pp. 152–164.

[BCG+08] Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch, *Provably good multicore cache performance for divide-and-conquer algorithms*, Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA) (Philadelphia, PA, USA), 2008, pp. 501–510.

[BCH+94] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha, *Implementation of a portable nested data-parallel language*, Journal of Parallel and Distributed Computing **21** (1994), no. 1, 4–14.

[BDKS04] Nikhil Bansal, Kedar Dhamdhere, Jochen Konemann, and Amitabh Sinha, *Non-clairvoyant scheduling for minimizing mean slowdown*, Algorithmica **40** (2004), no. 4, 305–318.

[BEGCS74] John L. Bruno, Jr. Edward G. Coffman, and Ravi Sethi, *Scheduling independent tasks to reduce mean finishing time*, Communications of the ACM **17** (1974), no. 7, 382–387.

[BFGL04] 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*, Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA) (Barcelona, Spain), June 2004, pp. 133–144.

[BFJ+95] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Rob Miller, Keith H. Randall, and Yuli Zhou, *Cilk 2.0 reference manual*, Massachusetts Institute of Technology Laboratory for Computer Science, 545 Technology Square, Cambridge, Massachusetts 02139, June 1995.

[BG96] Guy E. Blelloch and John Greiner, *A provable time and space efficient implementation of NESL*, International Conference on Functional Programming (ICFP) (Philadelphia, Pennsylvania), 1996, pp. 213–225.

[BGM95] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias, *Provably efficient scheduling for languages with fine-grained parallelism*, Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA) (Santa Barbara, California), July 1995, pp. 1–12.

[BGM99] Guy Blelloch, Phil Gibbons, and Yossi Matias, *Provably efficient scheduling for languages with fine-grained parallelism*, Journal of the ACM **46** (1999), no. 2, 281–321.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou, *Cilk: An efficient multithreaded runtime system*, Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) (Santa Barbara, California), July 1995, pp. 207–216.

[BJK+96] _____, *Cilk: An efficient multithreaded runtime system*, Journal of Parallel and Distributed Computing **37** (1996), no. 1, 55–69.

[BL97] Robert D. Blumofe and Philip A. Lisiecki, *Adaptive and reliable parallel computing on networks of workstations*, Proceedings of the USENIX Annual Technical Conference on UNIX and Advanced Computing Systems (Anaheim, California), January 1997, pp. 133–147.

[BL98] Robert D. Blumofe and Charles E. Leiserson, *Space-efficient scheduling of multithreaded computations*, SIAM Journal on Computing **27** (1998), no. 1, 202–229.

[BL99] _____, *Scheduling multithreaded computations by work stealing*, Journal of the ACM **46** (1999), no. 5, 720–748.

[BLM05] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin, *Deconstructing transactions: The subtleties of atomicity*, Proceedings of the Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD), Jun 2005.

[BLM06] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin, *Subtleties of transactional memory atomicity semantics*, Computer Architecture Letters **5** (2006), no. 2.

[BLS] Robert D. Blumofe, Charles E. Leiserson, and Bin Song, *Automatic processor allocation for work-stealing jobs*, Unpublished Manuscript.

[BLS03] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira, *Ownership types for object encapsulation*, Proceedings of the ACM Symposium on Principles of Programming Languages (POPL) (New Orleans, Louisiana), January 2003.

[Blu95] Robert D. Blumofe, *Executing multithreaded programs efficiently*, Ph.D. 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.

[BM72] Rudolf Bayer and Edward M. McCreight, *Organization and maintenance of large ordered indexes*, Acta Informatica **1** (1972), no. 3, 173–189.

[Boa08] OpenMP Architecture Review Board, *OpenMP specification and features*, `http://openmp.org/wp/`, May 2008.

[BP94] Robert D. Blumofe and David S. Park, *Scheduling large-scale parallel computations on networks of workstations*, Proceedings of the International Symposium on High Performance Distributed Computing (HPDC) (San Francisco, California), August 1994, pp. 96–105.

[BP98a] Robert D. Blumofe and Dionisios Papadopoulos, *The performance of work stealing in multiprogrammed environments*, Tech. Report TR-98-13, The University of Texas at Austin, Department of Computer Sciences, May 1998.

[BP98b] ———, *The performance of work stealing in multiprogrammed environments*, SIGMETRICS, 1998, pp. 266–267.

[BP99] ———, *Hood: A user-level threads library for multiprogrammed multiprocessors*, Technical Report, University of Texas at Austin, 1999.

[Bre74] Richard P. Brent, *The parallel evaluation of general arithmetic expressions*, Journal of the ACM **21** (1974), no. 2, 201–206.

[BS81] F. Warren Burton and M. Ronan Sleep, *Executing functional programs on a virtual tree of processors*, Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA) (Portsmouth, New Hampshire), October 1981, pp. 187–194.

[CB01] Walfredo Cirne and Francine Berman, *A model for moldable supercomputer jobs*, Proceedings of the International Symposium on Parallel and Distributed Systems (IPDPS) (Washington, DC, USA), 2001, pp. 50–59.

[CCL+00] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby, *ZPL: A machine independent programming language for parallel computers*, IEEE Transactions on Software Engineering **26** (2000), no. 3, 197–211.

[CL05] David Chase and Yossi Lev, *Dynamic circular work-stealing deque*, Proceedings of the ACM symposium on Parallelism in Algorithms and Architectures (SPAA) (New York, NY, USA), 2005, pp. 21–28.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms*, second ed., The MIT Press and McGraw-Hill, 2001.

[CMC+07] Brian D. Carlstrom, Austen McDonald, Michael Carbin, Christos Kozyrakis, and Kunle Olukotun, *Transactional collection classes*, Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP) (New York, NY, USA), 2007, pp. 56–67.

[Dav73] C.T. Davies, *Recovery semantics for a DB/DC system*, ACM National Conference, 1973, pp. 136–141.

[DD96] Xiaotie Deng and Patrick Dymond, *On multiprocessor system scheduling*, Proceedings of the ACM Symposium on Pararallel Algorithms and Architectures (SPAA), 1996, pp. 82–88.

[DES99] *DESMO-J: A framework for discrete-event modelling and simulation*, `http://asi-www.informatik.uni-hamburg.de/desmoj/`, 1999.

[DFL+06] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum, *Hybrid transactional memory*, Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 2006.

[DGBL96] Xiaotie Deng, Nian Gu, Tim Brecht, and KaiCheng Lu, *Preemptive scheduling of parallel jobs on multiprocessors*, Proceedings of the Symposium on Discrete Algorithms (SODA), 1996, pp. 159–167.

[DLL05] John Danaher, I-Ting Angelina Lee, and Charles E. Leiserson, *Exception handling in JCilk*, Synchronization and Concurrency in Object-Oriented Languages (SCOOL), October 2005, Available at `http://hdl.handle.net/1802/2095`.

[Dow98] Allen B. Downey, *A parallel workload model and its implications for processor allocation*, Cluster Computing **1** (1998), no. 1, 133–145.

[DS87] P. Dietz and D. Sleator, *Two algorithms for maintaining order in a list*, Proceedings of the Annual ACM Symposium on Theory of Computing (STOC) (New York City), May 1987, pp. 365–372.

[DS09] Luke Dalessandro and Michael Scott, *Strong isolation is a weak idea*, Proceedings of the Workshop on Transactional Memory (TRANSACT) (Raleigh, North Carolina), 2009.

[EAS+95] Guy Edjlali, Gagan Agrawal, Alan Sussman, Jim Humphries, and Joel Saltz, *Compiler and runtime support for programming in adaptive parallel environments*, Tech. Report CS-TR-3510, University of Maryland, 1995.

[EASS94] Guy Edjlali, Gagan Agrawal, Alan Sussman, and Joel Saltz, *Data parallel programming in an adaptive environment*, Tech. Report CS-TR-3350, University of Maryland, 1994.

[ECBD03] Jeff Edmonds, Donald D. Chinn, Timothy Brecht, and Xiaotie Deng, *Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics*, Journal of Scheduling **6** (2003), no. 3, 231–250.

[Edm99] Jeff Edmonds, *Scheduling in the dark*, Proceedings of Symposium on Theory of Computing (STOC), 1999, pp. 179–188.

[ESS05] Kemal Ebcioglu, Vijay Saraswat, and Vivek Sarkar, *X10: An experimental language for high productivity programming of scalable systems.*, Proceedings of Workshop on Productivity and Performance in High-End Computing (P-PHEC), 2005, In conjunction with *Symposium on High Performance Computer Architecture (HPCA)*.

[EZL89] Derek L. Eager, John Zahorjan, and Edward D. Lozowska, *Speedup versus efficiency in parallel systems*, IEEE Transactions on Computers **38** (1989), no. 3, 408–423.

[Fei] Dror Feitelson, *Parallel workloads archive*, `http://www.cs.huji.ac.il/labs/paral`

[Fei96] Dror G. Feitelson, *Packing schemes for gang scheduling*, Proceedings of the Workshop on Job Scheduling Strategies on Parallel Processors (JSSPP) (Dror G. Feitelson and Larry Rudolph, eds.), vol. 1162, Springer, 1996, pp. 89–110.

[Fei97] ———, *Job scheduling in multiprogrammed parallel systems (extended version)*, Tech. report, IBM Research Report RC 19790 (87657) 2nd Revision, 1997.

[Fin05] Jeremy T. Fineman, *Provably good race detection that runs in parallel*, Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA, August 2005.

[FL97] Mingdong Feng and Charles E. Leiserson, *Efficient detection of determinacy races in Cilk programs*, Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA) (Newport, Rhode Island), June22–25 1997, pp. 1–11.

[FL98] Matteo Frigo and Victor Luchangco, *Computation-centric memory models*, Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA) (Puerto Vallarta, Mexico), 1998, pp. 240–249.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran, *Cache-oblivious algorithms*, Proceedings of the Annual Symposium on Foundations of Computer Science (FOCS) (New York, New York), October 17–19 1999, pp. 285–297.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall, *The implementation of the Cilk-5 multithreaded language*, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (Montreal, Canada), 1998.

[FM87] Raphael Finkel and Udi Manber, *DIB — A distributed implementation of backtracking*, ACM Transactions on Programming Languages and Systems **9** (1987), no. 2, 235–256.

[For93] High Performance Fortran Forum, *High performance fortran language specification version 1.0*, Tech. report, Rice University, 1993.

[Fri98] Matteo Frigo, *The weakest reasonable memory model*, Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, January 1998.

[FTYZ90] Zhixi Fang, Peiyi Tang, Pen-Chung Yew, and Chuan-Qi Zhu, *Dynamic processor self-scheduling for general parallel nested loops*, IEEE Transactions on Computers **39** (1990), no. 7, 919–929.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The java language specification*, second ed., Addison Wesley, 2000.

[GK08] Rachid Guerraoui and Michal Kapalka, *On the correctness of transactional memory*, Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP) (New York, NY, USA), ACM, 2008, pp. 175–184.

[GR93] Jim Gray and Andreas Reuter, *Transaction processing: Concepts and techniques*, Morgan Kaufmann, 1993.

[Gra69] R. L. Graham, *Bounds on multiprocessing timing anomalies*, SIAM Journal on Applied Mathematics **17** (1969), no. 2, 416–429.

[Gra81] Jim Gray, *The transaction concept: Virtues and limitations*, Proceedings of the International Conference of Very Large Databases (VLDB), September 1981, pp. 144–154.

[Gu95] Nian Gu, *Competitive analysis of dynamic processor allocation strategies*, Master's thesis, York University, 1995.

[Hal84] Robert H. Halstead, Jr., *Implementation of Multilisp: Lisp on a multiprocessor*, Proceedings of the ACM Symposium on Lisp and Functional Programming (Austin, Texas), August 1984, pp. 9–17.

[HB99] Mor Harchol-Balter, *The effect of heavy-tailed job size. distributions on computer system design*, Conference on Applications of Heavy Tailed Distributions in Economics, 1999.

[HBD97] Mor Harchol-Balter and Allen B. Downey, *Exploiting process lifetime distributions for dynamic load balancing*, ACM Transactions on Computer Systems **15** (1997), no. 3, 253–285.

[HHL06] Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson, *Provably efficient two-level adaptive scheduling*, Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) (Saint-Malo, France), 2006.

[HHL07] ———, *Provably efficient online non-clairvoyant adaptive scheduling*, Proceedings of the International Conference on Parallel and Distribued Systems(IPDPS) (Long Beach, California, USA), March 2007.

[HK08] Maurice Herlihy and Eric Koskinen, *Transactional boosting: a methodology for highly-concurrent transactional objects*, Proceedings of ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP) (New York, NY, USA), ACM, Feb 2008, pp. 207–216.

[HLM03] Maurice P. Herlihy, Victor Luchangco, and Mark Moir, *Obstruction-free synchronization: Double-ended queues as an example*, Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS) (Providence, Rhode Island), May 2003, pp. 522–529.

[HLMS06] Danny Hendler, Yossi Lev, Mark Moir, and Nir Shavit, *A dynamic-sized nonblocking work stealing deque*, Distributed Computing **18** (2006), no. 3, 189–207.

[HM] Maurice Herlihy and J. Eliot B. Moss, *Transactional memory: Architectural support for lock-free data structures*, Proceedings of the International Conference on Computer Architecture (ISCA) (San Diego, CA), pp. 289–300.

[HS91] S. F. Hummel and E. Schonberg, *Low-overhead scheduling of nested parallelism*, IBM Journal of Research and Development **35** (1991), no. 5-6, 743–765.

[HS02] Danny Hendler and Nir Shavit, *Non-blocking steal-half work queues*, Proceedings of the Annual Symposium on Principles of Distributed Computing (PODC) (New York, NY, USA), ACM, 2002, pp. 280–289.

[HWC+04] 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*, Proceedings of the Annual International Symposium on Computer Architecture (ISCA), 2004, pp. 102–113.

[HZJ94] Michael Halbherr, Yuli Zhou, and Chris F. Joerg, *MIMD-style parallel programming with continuation-passing threads*, Proceedings of the International Workshop on Massive Parallelism: Hardware, Software, and Applications (Capri, Italy), September 1994.

[Ins] 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.

[IR94] Amos Israeli and Lihu Rappoport, *Disjoint-access-parallel implementations of strong shared memory primitives*, Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (PODC) (New York, NY, USA), ACM, 1994, pp. 151–160.

[KCJ+06] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen, *Hybrid transactional memory*, Proceedings of the ACM SIGPLAM Symposium on Principles and Practices of Parallel Programming (PPoPP) (New York, NY), March 2006.

[KKP+81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, *Dependence graphs and compiler optimizations*, Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (New York, NY, USA), ACM Press, 1981, pp. 207–218.

[KR03] Matthias Korch and Thomas Rauber, *A comparison of task pools for dynamic load balancing of irregular algorithms*, Concurrency and Computation: Practice & Experience **16** (2003), no. 1, 1–47.

[KZ88] Richard M. Karp and Yanjun Zhang, *A randomized parallel branch-and-bound procedure*, Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing (Chicago, Illinois), May 1988, pp. 290–300.

[LAK03] Sung-Chae Lim, Joonseon Ahn, and Myoung Ho Kim, *A concurrent B^{link}-tree algorithm using a cooperative locking protocol*, Lecture Notes in Computer Science, vol. 2712, Springer Berlin / Heidelberg, 2003, pp. 253–260.

[Lam79] Leslie Lamport, *How to make a multiprocessor computer that correctly executes multiprocess programs*, IEEE Transactions on Computers **C-28** (1979), no. 9, 690–691.

[LF03] Uri Lublin and Dror G. Feitelson, *The workload on parallel supercomputers: Modeling the characteristics of rigid jobs*, Journal of Parallel and Distributed Computing **63** (2003), no. 11, 1105–1122.

[Lib06] Ben Liblit, *An operational semantics for LogTM*, Tech. report, Department of Computer Sciences, University of Wisconsin-Madison, August 2006.

[LLS07] Malcolm Yoke Hean Low, Weiguo Liu, and Bertil Schmidt, *A parallel BSP algorithm for irregular dynamic programming*, Proceedings of the International Symposium on Advanced Parallel Processing Technologies, 2007, pp. 151–160.

[LO86] Will Leland and Teunis J. Ott, *Load-balancing heuristics and process behavior*, SIGMETRICS (New York, NY, USA), 1986, pp. 54–69.

[LV90] Scott T. Leutenegger and Mary K. Vernon, *The performance of multiprogrammed multiprocessor scheduling policies*, Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (Boulder, Colorado), May 1990.

[MBM+06] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood, *LogTM: Log-based transactional memory*, Proceedings of the Symposium on High Performance Computer Architecture (HPCA), Feb 2006.

[MBS+08] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc, *Practical weak-atomicity semantics for java stm*, Proceedings of the Annual Symposium on Parallelism in Algorithms and Architectures (SPAA) (New York, NY, USA), ACM, 2008, pp. 314–325.

[MCC+06] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun, *Architectural semantics for practical transactional memory*, Proceedings of the International Symposium on Computer Architecture (ISCA), June 2006.

[MCN+00] Xavier Martorell, Julita Corbalán, Dimitrios S. Nikolopoulos, Nacho Navarro, Eleftherios D. Polychronopoulos, Theodore S. Papatheodorou, and Jesús Labarta, *A tool to schedule parallel applications on multiprocessors: the NANOS CPU manager*, Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) (Cancun, Mexico) (Dror G. Feitelson and Larry Rudolph, eds.), Springer-Verlag, May 2000, Lecture Notes in Computer Science Vol. 1911, pp. 87–112.

[MEB88] Shikharesh Majumdar, Derek L. Eager, and Richard B. Bunt, *Scheduling in multiprogrammed parallel systems*, Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (Santa Fe, New Mexico, United States), May 1988, pp. 104–113.

[MG08] Katherine F. Moore and Dan Grossman, *High-level small-step operational semantics for transactions*, Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL) (New York, NY, USA), ACM, 2008.

[MH05] J. Eliot B. Moss and Antony L. Hosking, *Nested transactional memory: Model and preliminary architecture sketches*, Proceedings of the Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL) (San Diego, California), October 2005.

[MH06] J. Eliot B. Moss and Antony L. Hosking, *Nested transactional memory: Model and architecture sketches*, Science of Computer Programming, vol. 63, Elsevier, Dec 2006, pp. 186–201.

[MKHJ90] Eric Mohr, David A. Kranz, Robert H. Halstead, and Jr., *Lazy task creation: A technique for increasing the granularity of parallel programs*, IEEE Transactions on Parallel and Distributed Systems 2 (1990), 185–197.

[Mos85] J. Eliot B. Moss, *Nested transactions: An approach to reliable distributed computing*, MIT Press, Cambridge, MA, USA, 1985.

[Mos06] J. Eliot B Moss, *Open nested transactions: Semantics and support*, Proceedings of the Workshop on Memory Performance Issues (Austin, Texas), Feb 2006.

[MPT93] Rajeev Motwani, Steven Phillips, and Eric Torng, *Non-clairvoyant scheduling*, Proceedings of the Symposium on Discrete Algorithms (SODA), 1993, pp. 422–431.

[MR95] Rajeev Motwani and Prabhakar Raghavan, *Randomized algorithms*, Cambridge University Press, Cambridge, England, June 1995.

[MSH+06] 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*, Proceedings of the Workshop of Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT), June 2006.

[MSS05] Virendra J. Marathe, William N. Scherer III, and Michael L. Scott, *Adaptive software transactional memory*, Proceedings of the International Symposium on Distributed Computing (DISC) (Cracow, Poland), Sep 2005, Earlier but expanded version available as TR 868, University of Rochester Computer Science Dept., May2005.

[MVZ93] Cathy McCann, Raj Vaswani, and John Zahorjan, *A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors*, ACM Transactions on Computer Systems **11** (1993), no. 2, 146–178.

[NB99] Girija J. Narlikar and Guy E. Blelloch, *Space-efficient scheduling of nested parallelism*, ACM Transactions on Programming Languages and Systems (TOPLAS) **21** (1999), no. 1, 138–173.

[NMAT+07] 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*, Proceedings of ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP), March 2007.

[NSS93] V. K. Naik, M. S. Squillante, and S. K. Setia, *Performance analysis of job scheduling policies in parallel supercomputing environments*, Proceedings of the ACM/IEEE conference on Supercomputing (Portland, Oregon), November 1993, pp. 824–833.

[Pap79] Christos H. Papadimitriou, *The serializability of concurrent database updates*, Journal of the ACM **26** (1979), no. 4, 631–653.

[Ree78] David Patrick Reed, *Naming and synchronization in a decentralized computer system*, Tech. Report MIT/LCS/TR-205, Massachusetts Institute of Technology Laboratory for Computer Science, September 1978.

[Rei07] James Reinders, *Intel threading building blocks: Outfitting c++ for multi-core processor parallelism*, O'Reilly, 2007.

[RSAU91] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal, *A simple load balancing scheme for task allocation in parallel machines*, Proceedings of the Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA) (Hilton Head, South Carolina), July 1991, pp. 237–245.

[RSD+94] Emilia Rosti, Evgenia Smirni, Lawrence W. Dowdy, Giuseppe Serazzi, and Brian M. Carlson, *Robust partitioning schemes of multiprocessor systems*, Performance Evaluation **19** (1994), no. 2-3, 141–165.

[RSSD95] Emilia Rosti, Evgenia Smirni, Giuseppe Serazzi, and Lawrence W. Dowdy, *Analysis of non-work-conserving processor partitioning policies*, Proceedings of the International Parallel Processing Symposium (IPPS), 1995, pp. 165–181.

[RW08] R. Raman and D.S. Wise, *Converting to and from dilated integers*, IEEE Transactions on Computers **57** (2008), no. 4, 567–573.

[SATG+09] Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Robert Geva, Yang ni, and Adam Welc, *Towards transactional memory semantics for C++*, Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA) (Calgary, Canada), 2009.

[SATH+06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg, *McRT-STM: A high performance software transactional memory system for a multi-core runtime*, Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), March 2006, pp. 187–197.

[Sco06] Michael L. Scott, *Sequential specification of transactional memory semantics*, Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT), June 2006.

[SDMS08] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott, *Ordering-based semantics for software transactional memory*, Proceedings of the International Conference on Principles of Distributed Systems (OPODIS) (Berlin, Heidelberg), Springer-Verlag, 2008, pp. 275–294.

[Sen04] Siddhartha Sen, *Dynamic processor allocation for adaptively parallel work-stealing jobs*, Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, September 2004.

[Sev94] K. C. Sevcik, *Application scheduling and processor allocation in multiprogrammed parallel processing systems*, Performance Evaluation **19** (1994), no. 2–3, 107–140.

[Son98] Bin Song, *Scheduling adaptively parallel jobs*, Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts, January 1998.

[Squ95] Mark S. Squillante, *On the benefits and limitations of dynamic partitioning in parallel computer systems*, Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) (Santa Barbara, California) (Dror G. Feitelson and Larry

Rudolph, eds.), Springer-Verlag, April 1995, Lecture Notes in Computer Science Vol. 949, pp. 219–238.

[ST94] Dan Suciu and Val Tannen, *Efficient compilation of high-level data parallel algorithms*, Proceedings of the ACM Symposium of Parallel Algorithms and Architecrures (SPAA), 1994, pp. 57–66.

[Sup03] Supercomputing Technologies Group, MIT Laboratory for Computer Science, *Cilk 5.4.6 reference manual*, `http://supertech.csail.mit.edu/cilk/`, 2003.

[Sup06] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science, *Cilk 5.4.2.3 reference manual*, April 2006.

[SW81] T. F. Smith and M. S. Waterman, *Identification of common molecular subsequences.*, Journal of Molecular Biology **147** (1981), 195–197.

[TBB96] Kaushik Guha Timothy B. Brecht, *Using parallel program characteristics in dynamic processor allocation policies*, Performance Evaluation **27-28** (1996), 519–539.

[TG89] Andrew Tucker and Anoop Gupta, *Process control and scheduling issues for multiprogrammed shared-memory multiprocessors*, Proceedings of the ACM Symposium on Operating Systems Principles (SOSP) (Litchfield Park, Arizona), December 1989, pp. 159–166.

[TLW+94] John Turek, Walter Ludwig, Joel L. Wolf, Lisa Fleischer, Prasoon Tiwari, Jason Glasgow, Uwe Schwiegelshohn, and Philip S. Yu, *Scheduling parallelizable tasks to minimize average response time*, Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA), 1994, pp. 200–209.

[Tra83] I.L. Traiger, *Trends in systems aspects of database management*, International Conference on Databases, Wiley Heyden Ltd, 1983, pp. 1–21.

[TSP92] John Turek, Dennis Shasha, and Sundeep Prakash, *Locking without blocking: making lock based concurrent data structure algorithms nonblocking*, Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems (PODS) (New York, NY, USA), ACM, 1992, pp. 212–222.

[Val08] Leslie G. Valiant, *A bridging model for multi-core computing*, Proceedings of the European Symposium on Algorithms (ESA) (Berlin, Heidelberg), 2008, pp. 13–28.

[Wei86] Gerhard Weikum, *A theoretical foundation of multi-level concurrency control*, Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of database systems (PODS) (New York, NY, USA), ACM Press, 1986, pp. 31–43.

[WF99] David S. Wise and Jeremy D. Frens, *Morton-order matrices deserve compilers' support*, Tech. Report TR533, Indiana University, 1999.

[WS92] Gerhard Weikum and Hans-Jorg Schek, *Concepts and applications of multilevel transactions and open nested transactions*, Database Transaction Models for Advanced Applications, Morgan Kaufmann, San Francisco, CA, USA, 1992, pp. 515–553.

[YL01] K. K. Yue and D. J. Lilja, *Implementing a dynamic processor allocation policy for multiprogrammed parallel applications in the Solaris™ operating system*, Concurrency and Computation-Practice and Experience **13** (2001), no. 6, 449–464.

[Zip49] George K. Zipf, *Human behavior and the principle of least effort*, Addison-Wesley, 1949.