Memory Abstractions for Parallel Programming

by

I-Ting Angelina Lee

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

Author . . . .
Department of Electrical Engineering and Computer Science
March 7, 2012

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

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

Memory Abstractions for Parallel Programming
by
I-Ting Angelina Lee

Submitted to the Department of Electrical Engineering and Computer Science
on March 7, 2012, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

A ***memory abstraction*** is an abstraction layer between the program execution and the memory that provides a different "view" of a memory location depending on the execution context in which the memory access is made. Properly designed memory abstractions help ease the task of parallel programming by mitigating the complexity of synchronization or admitting more efficient use of resources. This dissertation describes five memory abstractions for parallel programming: *(i)* cactus stacks that interoperate with linear stacks, *(ii)* efficient reducers, *(iii)* reducer arrays, *(iv)* ownership-aware transactions, and *(v)* location-based memory fences. To demonstrate the utility of memory abstractions, my collaborators and I developed ***Cilk-M***, a dynamically multithreaded concurrency platform which embodies the first three memory abstractions.

Many dynamic multithreaded concurrency platforms incorporate ***cactus stacks*** to support multiple stack views for all the active children simultaneously. The use of cactus stacks, albeit essential, forces concurrency platforms to trade off between performance, memory consumption, and interoperability with serial code due to its incompatibility with linear stacks. This dissertation proposes a new strategy to build a cactus stack using ***thread-local memory mapping*** (or ***TLMM***), which enables Cilk-M to satisfy all three criteria *simultaneously*.

A ***reducer hyperobject*** allows different branches of a dynamic multithreaded program to maintain coordinated local views of the same nonlocal variable. With reducers, one can use nonlocal variables in a parallel computation without restructuring the code or introducing races. This dissertation introduces ***memory-mapped reducers***, which admits a much more efficient access compared to existing implementations.

When used in large quantity, reducers incur unnecessarily high overhead in execution time and space consumption. This dissertation describes support for ***reducer arrays***, which offers the same functionality as an array of reducers with significantly less overhead.

Transactional memory is a high-level synchronization mechanism, designed to be easier to use and more composable than fine-grain locking. This dissertation presents ***ownership-aware transactions***, the first transactional memory design that provides provable safety guarantees for "open-nested" transactions.

On architectures that implement memory models weaker than sequential consistency, programs communicating via shared memory must employ ***memory fences*** to ensure correct execution. This dissertation examines the concept of ***location-based memory fences***, which unlike traditional memory fences, incurs latency only when synchronization is necessary.

Thesis Supervisor: Charles E. Leiserson
Title: Professor

Acknowledgments

First and foremost, I would like to thank my wonderful advisor, Charles E. Leiserson. There are many things that I would like to thank Charles for, but as I am sitting here, staring at my X window opened with Vim, trying to write these acknowledgments, somehow words fell short. Nothing I can say will fully express my deep gratitude towards Charles. Charles has been a truly wonderful mentor. For me, graduate school has been a journey of self discovery, and without Charles, this journey would have been much more difficult. Having Charles as my advisor is unequivocally the best decision I made in graduate school; well, perhaps except for marrying my dear husband.

Mentioning who I would like to thank, too. My dear husband Brendan, once a PhD student himself, understands all the anxiety and emotional ups and downs that come with the territory of being a graduate student. The process of writing this thesis and putting everything together has been stressful and grueling. Without his companionship and the sense of solidarity, this process would have been much more isolating and lonely. Moreover, during this stressful period, he took over many of the household-related responsibilities without complaints, minimizing my sources of stress and distraction as best as one could ask for.

I chose an awesome set of thesis committee members. I would like to thank Jan-Willem Maessen, Armando Solar-Lezama, and Guy Steele for serving on my thesis committee. I have been told multiple times by older graduate students that committee members typically read through the intro chapter, and perhaps the next couple ones if you are lucky, but never the whole thesis. Well, my committee members have proven that I was told wrong. In all seriousness, I am grateful for all their insightful comments. One would have found more typos, grammatical errors, and minor technical misstatements without their careful reviews and detailed feedback.

I would like to thank all the Supertechies, past and present. Thanks to Matteo Frigo, for encouraging me to work on the cactus-stack problem (Chapter 3) and for providing critical feedback on the work. I often wondered whether there was anything Matteo cannot answer after my numerous interactions with him. Thanks to Bradley Kuszmaul, my go-to person for any system related questions, who has amazing breath and depth in both systems and theory related knowledge. Thanks to Kunal Agrawal, Jeremy Fineman, and Jim Sukha, my grad school "comrades," as we entered graduate school around the same time and experienced many things graduate school has to offer together, ranging from taking TQE classes, to TAing the undergraduate algorithms class, to attending conferences. It has been fun working and collaborating with Kunal and Jim. Without them the work on the ownership-aware transactions (Chapter 6) would not have been possible. Thanks to Jeremy for throwing all the dinner parties that distracted us at appropriate times. Thanks to Edya Ladan-Mozes, a wonderful collaborator and a supportive friend. Although we argue plenty, it has been a lot of fun working and sharing an office with her. Thanks to TB Schardl, for always providing insightful comments and always being willing to chat about research ideas, even half-baked ones. Without the critical discussions with him, I would have been stuck on the reducer array work (Chapter 5) for much longer. Thanks to Will Hasenplaugh, for giving support and helpful advice during my job search. Thanks to Aamir Shafi, who collaborated with me on the reducer work (Chapter 4), for being a wonderful collaborator and for putting up with my busy schedule. During one of his two terms visiting MIT, I was busy flying around for job interviews. Thanks to all others who have been part of the group throughout the years — Rezaul Chowdhury, Zhiyi Huang, Eka Natarajan, Yuan Tang, and Justin Zhang, for providing a supportive and intellectually stimulating environment.

Besides Supertechies, I have had the fortune to encounter other amazing collaborators outside of the group. A special thanks to Silas Boyd-Wickizer, for doing all the heavy lifting during our initial release of the TLMM Linux-kernel (Section 2.2), which made the work on the cactus-stack problem (Chapter 3) possible. Thanks to Dmitriy V'jukov, who approached Charles and me with

the idea of implementing asymmetric synchronization inside Cilk's work-stealing scheduler, out of which the work on location-based memory fences (Chapter 7) blossomed.

I would like to thank the administrative staff who have been a great help to me throughout my years in the lab — Alissa Cardone, Marcia Davidson and Mary McDavitt. Thanks to Alissa, who endured reading through my master thesis to help with my writing. Thanks to Marcia, who is always organized and efficient in taking care of any administrated-relative things that come up. In particular, I am grateful for her to help me printing out copies of my thesis for the readers the night before my defense, despite the fact that she already had plenty on her plate at that time. Thanks to Mary, who is always willing to hand a helping hand when I needed it.

I would also like to thank the various funding organizations which have funded my research. These funding sources include NSF Grants ACI-0324974, CNS-0540248, CNS-0615215, CNS-1017058, Singapore-MIT Alliance and Sun Microsystems Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the author and do not necessarily reflect the views of any of these organizations.

I would like to give special thanks to a group of close friends who have made my life in Cambridge colorful. I would like to thank Xiaolu Hsi, a friend and a wonderful mentor, who always provides great insights into the dilemmas we sometimes encounter in life. I would like to thank "the brunch girls" — Amy Williams, Shiyun Ruan, Michelle Sander, and Karen Zee — who themselves were once the graduate women at MIT, for listening and griping with me about graduate school life at MIT. I would like to thank my lovely housemates and good friends, Albert Huang and Stacy Wong. Not only have they provided a temporary living arrangement to Brendan and me during the last six months of thesis writing, they have been supportive and encouraging, and provided fun social activities throughout that helped keep me sane.

I would like to thank Tammy and David, my in-laws, for always being supportive and understanding, giving me the space to work whenever I had the chance to visit them.

Lastly, I would like to thank my parents and my dear sister, who have a special place in my heart. They never quite figured out what I was doing in graduate school and what I meant when I said I am doing "research." They nonetheless kept up their curiosity about what I was up to, and for my parents, especially on the topic of the *graduation date*. Without their constant inquiry, I might have stayed in school for even longer.

Contents

Chapter 1

Introduction

Moore's Law [110] states that the number of transistors is expected to double every two years. For over two decades since 1985, the doubling in transistors translated to a doubling in clock frequency, and application developers simply gained performance by riding the wave of clock frequency increase. A few years ago, since the processor power density reached the maximum that the devices could handle, hardware vendors moved to doubling the number of cores every two years in order to continue pursuing performance increase. Nowadays, the vast majority of computer systems—desktops, laptops, game consoles, embedded systems, supercomputers etc.—are built using multicore processor chips. This shift in hardware trends impacts all areas of computer science and changes the way people develop high performance software—one must write parallel programs in order to unlock the computational power provided by modern hardware.

Writing parallel programs is inherently more challenging than writing serial programs, however. Besides coding the desired functionality, the programmer must also worry about parallel task decomposition, scheduling the parallel tasks, and correctly synchronizing concurrent accesses to shared data among the tasks. A decade ago, writing parallel programs was still considered as a domain that requires special expertise. People coded to APIs such as POSIX threads [65], Windows API threads [59], or Java threads [52], structuring their computation into interacting ***persistent threads***, or ***pthreads***.[1] When programming directly on top of these threading APIs, the code tends to be cumbersome and complicated, because the programmer needs to write boiler-plate code to handle the task decomposition and scheduling explicitly. Furthermore, since the logic for task scheduling and communication is set up explicitly, entangled within the rest of the program logic, if the number of available processors changes, the program must be restructured in order to effectively use the resources.

To tackle these challenges and allow parallel programming to be widely adopted, researchers in industry and academia have been actively developing concurrency platforms. A ***concurrency platform*** provides a software abstraction layer running between the operating system and user programs that manages the processors' resources, schedules the computation over the available processors, and provides an interface for the programmer to specify parallel computations.

Contrary to the pthreading programming model, a concurrency platform lifts much of the burden off the programmer by providing a ***processor-oblivious dynamic multithreading*** (***dthreading*** for short) programming model, where the linguistic extensions for parallel control expose the logical parallelism within an application without mentioning the number of processors on which the application will run. With the dthreading programming model, the programmer specifies the logical

[1] No confusion should arise with the use of the term to mean POSIX threads, since POSIX threads are a type of persistent thread.

parallelism of the application, and the underlying runtime system schedules the computation in a way that respects the logical parallelism specified by the programmer. Since the proliferation of multicore architectures, the dthreading programming model has emerged as a dominant paradigm for programming a shared-memory multicore computers, since it provides a layer of ***parallelism abstraction***, which frees the programmer from worrying about load balancing, task scheduling, and restructuring the code when porting the application to a different machine.

The concept of parallelism abstraction is well-understood and widely adopted. Many examples of modern dthreading concurrency platforms exist, such as Cilk++ [94], Cilk Plus [69], Fortress [6], Habenero [9], Hood [21], Java Fork/Join Framework [90], JCilk [30], OpenMP 3.0 [120], Parallel Patterns Library (PPL) [105], Task Parallel Library (TPL) [92], Threading Building Blocks (TBB) [126], and X10 [26]. These dthreading concurrency platforms typically employ a "work-stealing" runtime scheduler, modeled after the scheduler of MIT Cilk [49], which has an efficient implementation and provides provable guarantees on execution time and memory consumption. In a work-stealing runtime scheduler, the processors are virtualized as pthreads, called ***workers***, and the scheduler schedules the computation over these workers in a way that respects the logical parallelism denoted by the programmer.

Whereas parallelism abstraction is a well-understood concept, researchers have only begun to study high-order memory abstractions to support common patterns of parallel programming. A ***memory abstraction*** is an abstraction layer between the program execution and the memory that provides a different "view" of a memory location depending on the execution context in which the memory access is made. For instance, ***transactional memory*** [64] is a type of memory abstraction — memory accesses dynamically enclosed by an `atomic` block appear to occur atomically. While transactional memory has been an active research area for the past few years, its adoption in practice has been slow at best. Similarly, another class of memory abstraction, ***hyperobjects*** [48], which is a linguistic mechanism that allows different branches of a dthreaded program to maintain coordinated local ***views*** of the same nonlocal object, is only supported in Cilk++ [94] and Cilk Plus [69].[2] [3]

Just as a concurrency platform lifts the burden of scheduling and task decomposition off the programmer with an appropriate parallelism abstraction, I contend that a concurrency platform can also mitigate other complexities that arise in parallel programming by providing properly designed memory abstractions. This dissertation discusses the following memory abstractions:

- ***cactus stacks that interoperate with linear stacks***, a new strategy to maintain a cactus stack memory abstraction using ***thread-local memory mapping*** (or ***TLMM***), referred to as ***TLMM-based cactus stacks***. A TLMM-based cactus stack enables a work-stealing runtime system to support *true* interoperability between parallel code and serial code while maintaining provably good resource usage;
- ***reducers with efficient access***, a new way of supporting a reducer mechanism using a memory-mapping approach in a work-stealing runtime system that incurs much less overhead;
- ***reducer arrays***, a new reducer type that supports arrays and allows different branches of a parallel program to maintain coordinated local views of some shared array;
- ***ownership-aware transactions***, the first transactional memory design that provides provable safety guarantees for "open-nested" transactions; and

[2] Technically, Cilk++ is the precursor of Cilk Plus; both are inspired by MIT Cilk, but they extend C++ instead of C.

[3] While the reduction operation that forms the semantic basis of reducer hyperobjects can be found in other modern concurrency platforms (e.g., Fortress [6], PPL [105], TBB [126], and OpenMP 3.0 [120]) and parallel programming languages (e.g., *Lisp [89], High Performance Fortran [79], and NESL [12]), the hyperobject approach to reduction markedly differs from these previous approaches; in particular, hyperobjects operate independently of any parallel control constructs.

- ***location-based memory fences***, a memory fence that forces the executing processor's instruction stream to serialize when another processor attempts to read the guarded memory location, thereby incurring latency only when synchronization is necessary.

In addition, my collaborators and I developed the ***Cilk-M System***, which embodies the first three memory abstractions and serves as a research platform to evaluate the utility of memory abstractions. The rest of this chapter provides a high-level overview of these memory abstractions and summarizes the contributions of the dissertation.

1.1 TLMM-Based Cactus Stacks

In a dthreading language such as Cilk, since multiple children of a function may exist simultaneously, the runtime system employs a cactus stack to support multiple stack views for all the active children simultaneously. In a ***cactus stack***, a function's accesses to stack variables properly respect the function's calling ancestry, even when many of the functions operate in parallel. In all known software implementations of cactus stacks, however, transitioning from serial code (using a linear stack) to parallel code (using a cactus stack) is problematic, because the type of stack impacts the calling conventions used to allocate activation frames and pass arguments. One could recompile the serial code to use a cactus stack, but this strategy is not feasible if the codebase includes legacy or third-party binaries for which the source is not available. We call the property of allowing arbitrary calling between parallel and serial code — including especially legacy (and third-party) serial binaries — ***serial-parallel reciprocity***, or ***SP-reciprocity*** for short.

There seems to be an inherent trade-off between supporting SP-reciprocity and maintaining good time and space bounds, and existing work-stealing concurrency platforms fail to satisfy at least one of these three criteria.[4] We refer to the problem of *simultaneously* achieving all three criteria as the ***cactus-stack problem***.

The incompatibility of cactus stacks and linear stacks impedes the acceptance of dthreading languages for mainstream computing. In particular, SP-reciprocity is especially important if one wishes to incrementally multicore-enable legacy object-oriented software. For example, suppose that a function `A` allocates an object `x` whose type has a member function `foo()`, which we parallelize. Now, suppose that `A` is linked with a legacy binary containing a function `B`, and `A` passes `&x` to `B`, which proceeds to invoke `x->foo(&y)`, where `&y` is a reference to a local variable allocated in `B`'s stack frame. Without SP-reciprocity, this simple callback would not work. Alternatively, one could simply rewrite the entire code base, ensuring that no legacy serial binaries call back to parallel functions; this option, however, is usually not feasible for large code bases or software that uses third party binaries.

If one is not willing to give up on SP-reciprocity, another alternative would be to compromise on the performance bound or space consumption guarantees that the currency platform could otherwise provide; TBB and Cilk Plus make such tradeoffs. Consequently, there exist computations for which TBB exhibits at most constant speedup on P workers, where an ordinary work-stealing scheduler could achieve nearly perfect linear speedup [131]. Similarly, there exist computations for which Cilk Plus fails to achieve good speed-up due to large stack space consumption, but which an ordinary work-stealing scheduler could achieve high speed-up with bounded stack space usage.

In Chapter 3, we will investigate how a good memory abstraction helps solve the cactus-stack problem and enable a concurrency platform to satisfy all three criteria simultaneously. Specifically,

[4] Java-based concurrency platforms do not suffer from the same problem with SP-reciprocity, because they are bytecode interpreted by a virtual-machine environment.

Chapter 3 describes a new strategy to implement cactus stacks in a work-stealing runtime environment by using a novel memory mechanism called thread-local memory mapping. ***Thread-local memory mapping***, or ***TLMM*** designates a region of the process's virtual-address space as "local" to each thread. The TLMM memory mechanism allows a work-stealing runtime scheduler to maintain a cactus-stack memory abstraction, referred to as the ***TLMM-based cactus stack***, in which each worker sees its own view of the linear stack corresponding to its execution context, even though multiple workers may share the same ancestors in their stack view. By maintaining a cactus-stack memory abstraction, a work-stealing scheduler is able to provide strong guarantees on execution time and stack space consumption while obtaining SP-reciprocity.

1.2 Support for Efficient Reducers and Reducer Arrays

Reducer hyperobjects (or ***reducers*** for short) [48] provide a memory abstraction for dthreading that allows different branches of a parallel computation to maintain coordinated local views of the same nonlocal variable. By using a reducer in place of a shared nonlocal variable, one avoids ***determinacy race*** [42] (also called a ***general race*** [116]) on the variable, where logically parallel branches of the computation access some shared memory location.

The concept of a reducer is based on an algebraic ***monoid***: a triple $(T, \otimes, e)$, where T is a set and $\otimes$ is an associative binary operation over T with identity e. During parallel execution, concurrent accesses to a reducer variable cause the runtime to generate and maintain multiple views for a given reducer variable, thereby allowing each worker to operate on its own local view. The runtime system manages these local views and when appropriate, ***reduces*** them together using the associative binary operator in a way that retains the serial semantics and produces deterministic final output, even when the binary operator is not commutative.

During execution, the runtime system employs a hash table, called a ***hypermap***, in each worker, which maps reducer instances to their corresponding views for the given worker. Accessing a reducer thus translates into a ***lookup*** on the hypermap, which is costly — approximately $11.8\times$ overhead compared to a normal memory access. In Chapter 4, we will explore how the TLMM mechanism may support a new way of implementing reducers, referred to as ***memory-mapped reducers***. Memory-mapped reducers allow a more efficient lookup operation compared to the hypermap approach, about $3.3\times$ overhead compared to a memory access. As an extension to the existing implementations of reducer mechanisms, in Chapter 4 we will also discuss runtime support to allow parallel reduce operation, which is not currently supported by other concurrency platforms.

Another natural extension for reducer hyperobjects is to allow array types. Existing implementations of reducers are designed for scalar reducers. If a programmer wishes to parallelize a large application that contains a shared array, she could either write her own reducer library from scratch, or declare an array of reducers. While the second approach seems simple enough, it suffers from three drawbacks which render the mechanism ineffective. First, declaring a reducer variable requires additional space (compared to the original data type) for metadata, so as to allow the runtime system to perform the necessary bookkeeping. The amount of space required for bookkeeping grows linearly with the number of reducer instances times the number of processors used. While the additional space consumption is expected, as a practical matter, it puts a limit how many reducers one can use in an application before its memory consumption becomes a bottleneck. Second, by declaring an array of reducers, access to an individual array element translates into a lookup operation to find the appropriate local view, which incurs considerable overhead. Finally, it turns out that, due to how the reducer mechanism works, a parallel execution using one reducer generates a nondeterministic amount of additional work (compared to its serial counterpart) that grows quadratically with the time it takes to perform a view creation and reduction. If k reducers are used, and the reduce operation for each reducer instance is processed serially, the additional overhead from the reduce operations also grows quadratically on k. While the overhead of managing views cannot be avoided, minimizing the number of reducers used and the time to perform view creation and reduction can effectively decrease the execution time.

In Chapter 5, we will study library support for reducer arrays to address these drawbacks. Specifically, the reducer array library allows the programmer to create a reducer variable corresponding to an array of objects, as long as the object type and operations on each object can be described by a monoid. By associating an array with a reducer, the runtime saves on space consumption due to reducer metadata. More importantly, the compiler is now able to perform optimization on the lookup operations: instead of requiring one lookup per access to the reducer array, only one lookup is required for all accesses within a single ***strand***, a piece of serial code that contains no parallel control. Lastly, the library is designed to optimize on the time it takes to perform view creation and reduction. In particular, the library employs a parallel reduce operation (which requires runtime support described in Chapter 4), further minimizing the time it takes to perform its reduce operation.

Even though the idea of reducer arrays is intriguing, it is nevertheless an open question whether the reducer array constitutes a useful linguistic mechanism in practice. While this library support exhibits significant performance improvement over its counterpart, an array of reducers, it cannot avoid generating additional work associated with view management due to how the reducer mechanism works. This additional work puts a hard limit on how many reducers one can use in a computation before the additional work of managing views dominates the work from the original computation and forms a bottleneck on scalability. In Chapter 5, we will extend the theoretical framework on analyzing programs that use reducers due to Leiserson and Schardl [96], analyze how much "effective parallelism" one can expect when using the reducer array library, and discuss the implications one can derive from the analysis.

1.3 Ownership-Aware Transactions

Transactional memory (TM), another type of memory abstraction, has been proposed as a high-level synchronization mechanism to avoid ***atomicity races*** [42] which cause nonatomic accesses to critical regions (also called ***data races*** [116]). Transactional memory was first proposed by Herlihy and Moss [64] as a hardware mechanism to support atomic updates of multiple independent memory locations. Ever since the advent of multicore architectures, there has been a renewed interest in transactional memory, and numerous designs have been proposed on how to support TM in hardware [7, 35, 56, 111, 124] and software [28, 37, 58, 63, 102, 127, 128], as well as hybrid schemes [29, 81, 97, 98].[5]

In the TM literature, researchers have argued that transactions may be a preferred synchronization mechanism over locking for the masses, for the following reasons. First, TM supports the simplicity of coarse grain locking and at the same time potentially provides performance close to that of fine-grain locking. With TM, the programmer simply encloses critical regions inside an `atomic` block, and the underlying TM system ensures that this section of code executes atomically. A TM system enforces atomicity by tracking memory locations accessed by transactions (using ***read sets*** and ***write sets***), finding transactional conflicts, and aborting transactions that conflict. Assuming conflicts are infrequent, multiple transactions can run concurrently, providing the performance of fine-grain locking.

[5] There have been many research studies of TM; for a survey of TM-related literature, please see [57].

Second, TM is more ***composable*** than locking — one can easily merge two smaller transactions into a larger one while maintaining the atomicity guarantee. For instance, suppose that a library implementing a thread-safe hash table supports `is_full()` and `insert()` function calls by using locks. An application using the hash table may wish to call `is_full()` and subsequently `insert()` only if `is_full()` returns `false`. To achieve the desired the semantics, the application must ensure that `is_full()` and `insert()` are executed atomically (i.e., no other threads call `insert()` during the intermediate state). One possible approach is for the hash table library to support some form of `lock_table()` and `unlock_table()` function calls, which the application can invoke around the `is_full()` and `insert()` to ensure atomicity. This approach references the underlying implementation and breaks the hash table abstraction, however. Another possible approach is for the application to implement its own layer of locking protocol on top of its accesses to the hash table. This approach imposes additional burden on the applications developer; moreover, now both the hash table library and the application must manage its own set of locks for accessing the hash table. The same issue does not arise if the library implements the hash table using transactions. The application can simply enclose the calls to `is_full()` and `insert()` in a transaction, which forms ***nested*** transactions, where an `atomic` block dynamically encloses another `atomic` block, and the underlying TM system guarantees that the calls to these functions appear to execute atomically.

It turns out that previous proposals for handling nested transactions either create large memory footprints and unnecessarily limit concurrency, resulting inefficient execution, or fail to guarantee ***serializability*** [121], a correctness condition often used to reason about TM-based programs, rendering the transactions noncomposable and possibly producing anomalous program behaviors that are tricky to reason about. In Chapter 6, we will examine a TM system design that employs ***ownership-aware transactions*** (OAT) which, compared to previous proposals, admits more concurrency and provides provable safety guarantees, referred to as "abstract serializability."

With OAT, the programmer does not specify transactions explicitly using `atomic` blocks; rather, she programs with transactional modules, and the OAT system guarantees abstract serializability as long as the program conforms to a set of well-defined constraints on how the modules share data. The abstract serializability provides a means for the programmer to reason about the program behavior, and the OAT type system can statically enforce the set of constraints for the most part, and the rest can be checked during execution. With this transactional module interface, the programmer focuses on structuring the code and data into modular components, and the OAT system maintains the memory abstraction that data belonging to a module is updated atomically and thus presents a consistent view to other modules.

1.4 Location-Based Memory Fences

Sequential consistency (SC) [86] provides an intuitive memory model for the programmer, in which all processors observe the same sequence of memory accesses, and within this sequence, the accesses made by each processor appear in its program order. Nonetheless, existing architectures typically implement weaker memory models that relax the memory ordering to achieve higher performance. The reordering affects the correctness of the software execution in the case where it is crucial that the execution follows the program order and the processors must observe the relevant accesses in the same relative order. Therefore, to ensure a correct execution in such cases, architectures that implement weak memory models provide serializing instructions and memory fences to force a specific memory ordering when necessary.

On modern multicore architectures, since threads (surrogates for processors) typically communicate and synchronize via shared memory, the use of memory fences is a *necessary evil* — it is necessary to ensure correct execution for synchronization algorithms that perform simple load-store operations on shared variables to achieve mutual exclusion among threads; it is evil, because it incurs high overhead. I ran a simple microbenchmark on AMD Opteron with 4 quad-core 2 GHz CPUs, and the results show that a thread running alone and executing the Dekker protocol [39] with a memory fence, accessing only a few memory locations in the critical section, runs $4-7$ times slower than when it is executing the same code without a memory fence.

This high overhead may be unnecessary. Traditional memory fences are program-based; meaning, a memory fence enforces a serialization point in the program instruction stream — it ensures that all memory references before the fence in the program order have taken effect *globally* (i.e., visible to all processors) before the execution continues onto instructions after the fence. Such program-based memory fences always cause the processor to stall, even when the synchronization is unnecessary during a particular execution.

In Chapter 7, we will turn our attention to the notion of a ***location-based memory fence*** that has the same semantic guarantees as an ordinary memory fence,[6] but which incurs latency only when synchronization is needed. Unlike a program-based memory fence, a location-based memory fence serializes the instruction stream of the executing thread T_1 only when a different thread T_2 attempts to read the memory location which is guarded by the location-based memory fence. This notion of location-based memory fences is a memory abstraction, because the write associated with the fence behaves differently depending on the execution context — it behaves as a memory fence when synchronization is necessary but otherwise behaves as an ordinary write.

As we will see in Chapter 7, location-based memory fences can be supported by a lightweight hardware mechanism, which requires only a small modification to existing architectures. Furthermore, we will evaluate the feasibility of location-based memory fences with a software prototype to simulate the effect of location-based memory fences. Even though the software prototype incurs higher overhead compared to what the hardware mechanism would when synchronization is needed, the experiments show that applications still perform better using location-based memory fences than using program-based memory fences.

1.5 Contributions

This dissertation consists of two parts. The first part describes the Cilk-M system and memory abstractions that the Cilk-M system embodies. Chapter 2 offers a brief overview of the Cilk technology and the implementation of TLMM to provide background for the next three chapters. Chapters 3–5 discuss the three memory abstractions under Cilk-M in details, including their evaluations. The second part includes Chapters 6 and 7, which describe the other two memory abstractions that are independent from each other. Chapter 8 offers some concluding remarks. More specifically, my dissertation describes the following contributions:

- ***The design and implementation of TLMM-Based cactus stacks in Cilk-M***
  Chapter 3 presents TLMM-based cactus stacks, a strategy to maintain a cactus-stack memory abstraction in a work-stealing runtime system which is critical in solving the cactus-stack problem. To evaluate the TLMM-based cactus stacks, Chapter 3 analyzes the performance and space usage of the Cilk-M system both theoretically and empirically. The Cilk-M system provides strong guarantees on scheduler performance and stack space. Benchmark results indicate that the performance of the Cilk-M system is comparable to the Cilk 5.4.6 system

[6] To be more precise, the proposed implementation for a location-based memory fence provides the same semantic guarantees as an ordinary memory fence if the program satisfies certain conditions, which we elaborate in Chapter 7.

and Cilk Plus, and the consumption of stack space is modest. This work was done jointly with Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson and appears in [91].

- ***The design and implementation of memory-mapped reducers in Cilk-M***
  Chapter 4 investigates how a reducer mechanism can be supported using TLMM, which permits a much more efficient lookup operations on reducers, approximately 4× faster than the hypermap approach. Chapter 4 also describes how the Cilk-M system supports parallel reduce operations, which are currently not supported in other concurrency platforms.

- ***The design and implementation of reducer arrays in Cilk-M***
  Chapter 5 investigates library support for reducer arrays, which offer significant performance improvement over arrays of reducers that provide the same functionality. In addition, Chapter 5 extends the theoretical analysis for analyzing programs that use reducers due to Leiserson and Schardl [96] to incorporate the use of reducers that employ parallel reduce operations, and offers some insight as to when the additional work generated by reducers becomes a bottleneck in scalability. This work was done jointly with Aamir Shafi, Tao B. Schardl, and Charles E. Leiserson.

- ***The design of ownership-aware transactional memory***
  Chapter 6 explores a TM system design that supports ownership-aware transactions (OAT), which is the first transactional memory design that supports "open-nested" transactions that are composable. The framework of OAT incorporates the notion of modules into the TM system and uses a commit mechanism that handles a piece of data differently depending on which modules owns the data. Chapter 6 also provides a set of precise constraints on interactions and sharing of data among modules based on notions of abstraction. The OAT commit mechanism and these restrictions on modules allow us to prove that ownership-aware TM has clean memory-level semantics. Compared to previous proposals for supporting nested transactions, the OAT system admits more concurrency and provides provable safety guarantees. This work was done jointly with Kunal Agrawal and and Jim Sukha and appears in [4].

- ***The design of location-based memory fences***
  Chapter 7 introduces the concept of location-based memory fences, which unlike the conventional program-based memory fences, incur latency only when synchronization is necessary. Chapter 7 also describes a lightweight hardware mechanism for implementing the location-based memory fences, which requires only a small modification to existing architectures. This work was done jointly with Edya Ladan-Mozes and Dmitry Vyukov and appears in [84].

The Cilk-M system came out as the resulting artifact of the evaluation process, which was a joint effort with Silas Boyd-Wickizer, Zhiyi Huang, Charles E. Leiserson, and Aamir Shafi. We modified the Linux operating system kernel to provide support for TLMM, reimplemented the cactus stack in the open-source Cilk-5 runtime system, and added support for reducer hyperobjects. We also ported the Cilk-M system to be compatible with the Cilk Plus compiler, so that the runtime can be linked with code compiled using the Cilk Plus compiler. The Cilk-M system is unique in that it employs TLMM to implement these memory abstractions. Moreover, Cilk-M is the first C/C++-based dthreading concurrency platform that *simultaneously* supports SP-reciprocity, scalable performance, and bounded memory consumption.

Part I:
Memory Abstractions in Cilk-M

Chapter 2

Introduction to Cilk-M

Cilk-M is a dynamically multithreaded concurrency platform that employs an algorithmically sound work-stealing scheduler [20] modeled after the scheduler of MIT Cilk-5 [49]. It embodies a TLMM-based cactus stack and memory-mapped reducer hyperobjects and serves as a research platform to evaluate the utility of memory abstractions. Cilk-M inherited its performance model and the work-stealing algorithm from its predecessor Cilk-5. Like Cilk-5, Cilk-M supports scalable performance and bounded memory consumption. On the other hand, Cilk-M differs from Cilk-5 in that it supports seamless transitioning between parallel code and serial code, attributed to its use of a TLMM-based cactus stack. In fact, Cilk-M is the first C/C++-based concurrency platform that supports all three criteria simultaneously.

Implementation wise, what distinguishes Cilk-M from other concurrency platforms is its utilization of the ***thread-local memory mapping (TLMM)*** mechanism. Whereas thread-local storage [129] gives each thread its own local memory at different virtual addresses within shared memory, TLMM allows a portion of the virtual-memory address space to be mapped independently by the various threads. The TLMM mechanism requires operating system support, which my collaborators and I implemented by modifying the open-source Linux operating system kernel. TLMM provides a novel mechanism for implementing memory abstractions, for which Cilk-M's implementation of cactus stacks and reducer hyperobjects attest.

This chapter serves to introduce Cilk-M, which embodies the memory abstractions described in Chapters 3, 4, and 5. Section 2.1 gives an overview of the Cilk-M system implementation, its linguistic and performance models, and the work-stealing scheduler. TLMM is a mechanism shared by all memory abstractions under Cilk-M. Section 2.2 describes how we modified the Linux kernel to provide support for TLMM.[1] Since TLMM requires modification to the operating system, Section 2.3 considers another possible memory-mapping solution to simulate the TLMM effect without requiring operating-system support.

2.1 Cilk Technology and the Development of Cilk-M

A brief history of Cilk technology

Cilk-M is an implementation of Cilk. Before we overview the development and implementation of Cilk-M, we shall first overview a brief history of Cilk technology to account for where the major concepts inherited by Cilk-M originate. Cilk technology has developed and evolved over more than 15 years since its origin at MIT. Portions of the history I document here were before my time at

[1] Silas Boyd-Wickizer is the main contributor of our first TLMM modification to the Linux kernel.

MIT. The text under this subheading is partially abstracted from the "Cilk" entry in *Encyclopedia of Distributed Computing* [93] with the author's consent. I invite interested readers to go through the original entry for a more complete review of the history.

Cilk (pronounced "silk") is a linguistic and runtime technology for algorithmic multithreaded programming originally developed at MIT. The philosophy behind Cilk is that a programmer should concentrate on structuring her or his program to expose parallelism and exploit locality, leaving Cilk's runtime system with the responsibility of scheduling the computation to run efficiently on a given platform. The Cilk runtime system takes care of details like load balancing, synchronization, and communication protocols. Cilk is algorithmic in that the runtime system guarantees efficient and predictable performance. Important milestones in Cilk technology include the original Cilk-1 [15, 18, 74],[2] Cilk-5 [46, 49, 125, 132], and the commercial Cilk++ [27, 66, 94].

The first implementation of Cilk, Cilk-1, arose from three separate projects at MIT in 1993. The first project was theoretical work [19,20] on scheduling multithreaded applications. The second was StarTech [73, 82, 83], a parallel chess program built to run on the Thinking Machines Corporation's Connection Machine Model CM-5 Supercomputer [95]. The third project was PCM/Threaded-C [54], a C-based package for scheduling continuation-passing-style threads on the CM-5. In April 1994 the three projects were combined and christened Cilk. Cilk-1 is a general-purpose runtime system that incorporated a provably efficient work-stealing scheduler. While it provided a provably efficient runtime support, it offered little linguistic support.

Cilk-5 introduced Cilk's linguistic model, which provided simple linguistic extensions such as **spawn** and **sync** for multithreading to ANSI C. The extension is ***faithful***, which means that parallel code retains its serial semantics when run on one processor. Furthermore, the program would be an ordinary C program if the keywords for parallel controls were elided, referred to as the ***serial elision***. Cilk-5 was first released in March 1997 [49], which included a provably efficient runtime scheduler like its predecessor, and a source-to-source compiler, compiling Cilk code to processed C code with calls to the runtime library.

In September 2006, responding to the multicore trend, MIT spun out the Cilk technology to Cilk Arts, Inc., a venture-funded start-up founded by technical leaders Charles E. Leiserson and Matteo Frigo, together with Stephen Lewin-Berlin and Duncan C. McCallum. Although Cilk Arts licensed the historical Cilk codebase from MIT, it developed an entirely new codebase for a C++ product aptly named Cilk++ [27,94], which was released in December 2008 for the Windows Visual Studio and Linux/gcc compilers.

Cilk++ improved upon the MIT Cilk-5 in several ways. The linguistic distinction between Cilk functions and C/C++ functions was lessened, allowing C++ "call-backs" to Cilk code, as long as the C++ code was compiled with the Cilk++ compiler.[3] The **spawn** and **sync** keywords were renamed **cilk_spawn** and **cilk_sync** to avoid naming conflicts. Loops were parallelized by simply replacing the **for** keyword with the **cilk_for** keyword, which allows all iterations of the loop to operate in parallel. Cilk++ provided full support for C++ exceptions. It also introduced reducer hyperobjects. A Cilk++ program, like a Cilk program, retains its serial semantics when run on one processor. Moreover, one may obtain the ***serialization*** of a Cilk++ program, which is the same concept as serial elision, by eliding **cilk_spawn** and **cilk_sync** and replacing **cilk_for** with **for** .

Cilk Arts was sold to Intel Corporation in July 2009, which continued developing the technology. In September 2010, Intel released its ICC compiler with Intel Cilk Plus [67, 69]. The product included Cilk support for C and C++, and the runtime system provided transparent integration with

[2] Called "Cilk" in [15, 18, 74], but renamed "Cilk-1" in [49] and other MIT documentation.

[3] This distinction was later removed altogether by Intel Cilk Plus, though at the expense of sacrificing the performance and space guarantees provided by a working-stealing scheduler. We will explore this issue in more depth in Chapter 3

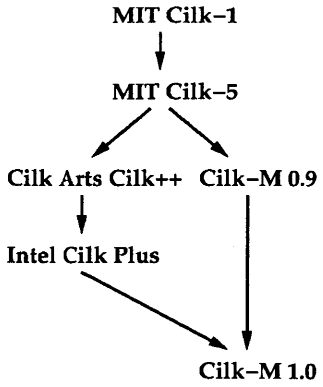


Figure 2-1: The lineage of Cilk-M 0.9 and Cilk-M 1.0.

legacy binary executables.

The development of Cilk-M

Cilk-M's runtime system is based on the open-source Cilk-5 runtime system,[4] modified to incorporate the use of a TLMM-based cactus stack. Due to its use of TLMM, the Cilk-M system currently only runs on x86 64-bit architectures.

The Cilk-M system started out being only a runtime scheduler (referred to as Cilk-M 0.9) and had no compiler support. Cilk-5's source-to-source compiler, which supports the basic primitives for parallel control, does not work with the Cilk-M runtime system due to the differences in how the two systems maintain cactus stacks. To evaluate the Cilk-M 0.9 runtime system, my collaborators and I manually hand-compiled benchmarks using gcc's inline assembly feature to force the compiler to generate the desired assembly code. Manually compiling all benchmarks soon became impractical, given that we wanted to experiment with larger applications that use reducers.

It turns out that Cilk-M's special calling convention closely resembles the calling convention for parallel functions in Cilk Plus [69]. We ported the Cilk-M runtime to adopt Cilk Plus' Application Binary Interface (ABI) [68] so as to interface with the code compiled by the Cilk Plus compiler (referred to as Cilk-M 1.0). Interfacing with the Cilk Plus compiler enabled us to obtain compiler support for compiling large C and C++ applications with Cilk Plus keywords for parallel control with much less engineering effort than what building a full compiler would have required.

Figure 2-1 shows the lineage of Cilk-M 0.9 and Cilk-M 1.0 and summarizes the relation between different versions of Cilk that I mentioned. Cilk-M inherited Cilk-5's simple linguistics, although it supports the C++ syntax like Cilk++ and Cilk Plus (including **cilk_for**) due to its use of the Cilk Plus compiler. Cilk-M's performance model and its work-stealing scheduler can be traced back to Cilk-1, although the "work-first principle" [49] mentioned later in this section was derived and exploited since the implementation of Cilk-5. Henceforth, when I describe the Cilk-M system, I mean the Cilk-M 1.0 implementation, unless I state Cilk-M 0.9 specifically.

Cilk-M's linguistic model

Cilk-M supports three main keywords for parallel control: **cilk_spawn**, **cilk_sync**, and **cilk_for**. Parallelism is created using the keyword **cilk_spawn**. When a function invocation is preceded by

[4] The open-source Cilk-5 system is available at `http://supertech.csail.mit.edu/cilk/cilk-5.4.6.tar.gz`.

the keyword **cilk_spawn**, the function is ***spawned*** and the scheduler may continue to execute the continuation of the caller in parallel with the spawned subroutine without waiting for it to return. The complement of **cilk_spawn** is the keyword **cilk_sync**, which acts as a local barrier and joins together the parallelism forked by **cilk_spawn**. The Cilk-M runtime system ensures that statements after a **cilk_sync** are not executed until all functions spawned before the **cilk_sync** statement have completed and returned.

The keyword **cilk_for** is the parallel counterpart of the looping construct **for** in C and C++ that permits loop iterations to run in parallel. The Cilk Plus compiler converts the **cilk_for** into an efficient divide-and-conquer recursive traversal over the iteration space. From the runtime system's perspective, the **cilk_for** construct can be desugared into code containing **cilk_spawn** and **cilk_sync**. Certain restrictions apply to the loop initializer, condition, and increment, for which I omit the details here and refer interested readers to [69].

In Cilk-5, there is a clear distinction between function types — a function that contains keywords for parallel control must be declared to be a ***Cilk function***, and a Cilk function must be spawned but not called. Similarly, only Cilk functions but not C functions can be spawned. Since the Cilk-M system supports SP-reciprocity, or, seamless interoperability between serial and parallel code, this delineation between serial and parallel code is lifted. The compiler no longer needs to keep track of function types, and whether there is parallelism or not depends on whether a function is called or spawned — any function may be called as well as spawned; if a function is spawned, it may execute in parallel with the continuation of its parent; if it is called, while it may execute in parallel with its children, the continuation of its parent cannot be resumed until it returns. Nevertheless, we shall keep the same terminology and refer to functions that contain keywords for parallel controls as Cilk functions.

Although Cilk-M supports large C++ applications compiled using the Cilk Plus compiler, the current implementation does not handle exceptions that occur during parallel execution. In principle, Cilk-M could support exceptions, and the implementation might be simpler than that in Cilk Plus, since on Windows, the structured exception handling mechanism provided by the operating system expects the frame allocation to follow a linear stack layout (i.e., a child frame should be allocated at a relatively lower address compared to that of its parent, assuming the stack grows from high to low addresses). As we shall see in Chapter 3, the way Cilk Plus runtime maintains a cactus stack does not necessarily satisfy this condition, whereas Cilk-M does due to its use of a TLMM-based cactus stack.

Cilk-M's performance model

Two important parameters dictate the performance of a Cilk computation: its ***work***, which is the execution time of the computation on one processor, and its ***span***[5], which is the execution time of the computation on an infinite number of processors.

With these two parameters, one can give two fundamental lower bounds on how fast a Cilk program can run. Let us denote the execution of a given computation on P processors as T_P. Then, the work of the computation is T_1, and the span is T_∞. The first lower bound, referred to as the ***Work Law***, is $T_P \geq T_1/P$, because at each time step, at most P units of work can be executed, and the total work is T_1. The second lower bound, referred to as the ***Span Law***, is $T_P \geq T_\infty$, because a finite number of processors cannot execute faster than an infinite number of processors. Assuming an ideal parallel computer, a work-stealing scheduler executes in time

$$T_P \leq T_1/P + c_\infty T_\infty. \qquad (2.1)$$

[5] "Span" is sometimes called "critical-path length" [18] and "computation depth" [13] in the literature.

The first term on the right hand side of Equation 2.1 is referred to as the ***work term***, and the second term as the ***span term***. One can also define the ***average parallelism*** as $\overline{P} = T_1/T_\infty$, which corresponds to the maximum possible speedup that the application can obtain, and the ***parallel slackness*** to be the ratio $\overline{P}/P$. Assuming sufficient parallel slackness, meaning $\overline{P}/P \gg c_\infty$, then it follows that $T_1/P \gg c_\infty T_\infty$. Hence, from Inequality 2.1, we obtain that $T_P \approx T_1/P$, which means that we achieve linear speedup when the number of processors P is much smaller than the average parallelism $\overline{P}$. Thus, when sufficient parallel slackness exists, the span overhead c_∞ has little effect on performance.

This performance model gives rise to the ***work-first principle*** [49], which states:

> "Minimize the scheduling overhead borne by the work of a computation. Specifically, move overheads out of the work and onto the [span]."

As we shall see in the later chapters, the work-first principle pervades the implementation of Cilk-M. In particular, the use of a TLMM-based cactus stack in Cilk-M helps minimize the work compared to a heap-based cactus stack, but at the additional cost of a larger c_∞ term. Nevertheless, when an application exhibits ample parallelism, the larger c_∞ term has little effect on performance.

Cilk-M's work-stealing runtime scheduler

Cilk-M's work-stealing scheduler load-balances parallel execution across the available worker threads. Like Cilk-5, Cilk-M follows the "lazy task creation" strategy of Kranz, Halstead, and Mohr [80], where the worker suspends the parent when a child is spawned and begins work on the child.[6] Operationally, when the user code running on a worker encounters a **cilk_spawn**, it invokes the child function and suspends the parent, just as with an ordinary subroutine call, but it also places the parent frame on the bottom of a ***deque*** (double-ended queue). When the child returns, it pops the bottom of the deque and resumes the parent frame. Pushing and popping frames from the bottom of the deque is the common case, and it mirrors precisely the behavior of C or other Algol-like languages in their use of a stack.

The worker's behavior departs from ordinary serial stack execution if it runs out of work. This situation can arise if the code executed by the worker encounters a **cilk_sync**. In this case the worker becomes a ***thief***, and it attempts to steal the topmost (oldest) frame from a ***victim*** worker. Cilk-M's strategy is to choose the victim randomly, which can be shown [20, 49] to yield provably good performance. If the steal is ***successful***, the worker resumes the stolen frame.

Another situation where a worker runs out of work occurs if it returns from a spawned child to discover that its deque is empty. In this case, it first checks whether the parent is stalled at a **cilk_sync** and if this child is the last child to return. If so, it performs a ***joining steal*** and resumes the parent function, passing the **cilk_sync** at which the parent was stalled. Otherwise, the worker engages in random work-stealing as in the case when a **cilk_sync** was encountered.

What I have described thus far is a general overview of how a work-stealing scheduler operates, which applies to the Cilk-5 scheduler as well. Since the Cilk-M system supports SP-reciprocity, the Cilk-M runtime differs from the Cilk-5 runtime in that it must keep track of how a function is invoked to maintain the call versus spawn semantics accordingly. Maintaining the correct semantics during execution is mainly a matter of handling the runtime data structure differently. In this regard, many of the implementation details of the Cilk-M runtime resemble those of the Cilk++ runtime system, and I refer interested readers to [48]. In particular, an entry in a ready deque may be either

[6] An alternative strategy is for the worker to continue working on the parent, and have thieves steal spawned children. Cilk-1 [18], TBB [126], and TPL [92] employ this strategy, but it can require unbounded bookkeeping space even on a single processor.

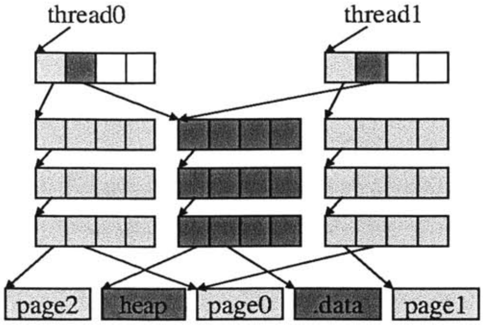


Figure 2-2: Example of a x86 64-bit page-table configuration for two threads on TLMM-Linux. The portion of the data structure dealing with the TLMM region is shaded light grey, and the remainder corresponding to the shared region is shaded dark grey. In the TLMM region, thread0 maps page2 first and then page0, whereas thread1 maps page1 first and then page0. The pages associated with the heap and the data segments are shared between the two threads.

a single frame, or a sequence of frames, representing a sequence of called Cilk functions. When a steal occurs, the entire sequence in an entry is stolen instead of just a single frame. Doing so ensures that a caller of a Cilk function cannot be stolen and resumed before the Cilk function returns.

2.2 Support for TLMM

A traditional operating system provides each process with its own virtual-address space. No two processes share the same virtual-address space, and all threads within a given process share the process's entire virtual-address space. TLMM, however, designates a region of the process's virtual-address space as "local" to each thread. This special ***TLMM region*** occupies the same virtual-address range for each thread, but each thread may map different physical pages to the TLMM region. The rest of the virtual-address space outside of the TLMM region remains shared among all threads within the process.

My collaborators and I modified the Linux kernel to implement TLMM, referred to as the TLMM-Linux, which provides a low-level virtual-memory interface organized around allocating and mapping physical pages. The design attempts to impose as low overhead as possible while allowing the Cilk-M runtime system to implement its work-stealing protocol efficiently. In addition, the design tries to be as general as possible so that the API can be used by other user-level utilities, applications, and runtime systems besides Cilk-M. This section describes the implementation of TLMM-Linux and the TLMM interface.

TLMM implementation

We implemented TLMM for Linux 2.6.32 running on x86 64-bit CPU's, such as AMD Opterons and Intel Xeons. We added about 600 lines of C code to manage TLMM virtual-memory mappings and modified several lines of the context-switch and memory-management code to be compatible with TLMM.

Figure 2-2 illustrates the design. TLMM-Linux assigns a unique root page directory to each thread in a process. The x86 64-bit page tables have four levels, and the page directories at each level contain 512 entries. One entry of the root-page directory is reserved for the TLMM region, which corresponds to 512-GByte of virtual address space, and the rest of the entries correspond to

addr_t sys_reserve(size_t n):
Reserve n bytes for the TLMM region, and return the start address.

pd_t sys_palloc(void):
Allocate a physical page, and return its descriptor.

sys_pfree(pd_t p):
Free the page descriptor p.

sys_pmap(unsigned int n, pd_t p[], addr_t a):
Map the n pages represented by the descriptors in p starting at virtual address a.

Figure 2-3: System-call API for TLMM.

the shared region. Threads in TLMM-Linux share page directories that correspond to the shared region. Therefore, the TLMM-Linux virtual-memory manager needs to synchronize the entries in each thread's root page directory and populate the shared lower-level page directories only once.

TLMM interface

Figure 2-3 summarizes the TLMM system call interface. sys_reserve marks n bytes of the calling thread's process address space as the TLMM region and returns the starting address of the region. sys_palloc allocates a physical page and returns its page descriptor. A page descriptor is analogous to a file descriptor and can be accessed by any thread in the process. sys_pfree frees a page descriptor and its associated physical page.

To control the physical-page mappings in a thread's TLMM region, the thread calls sys_pmap, specifying an array of page descriptors to map, as well as a base address in the TLMM region at which to begin mapping the descriptors. sys_pmap steps through the array of page descriptors, mapping physical pages for each descriptor to subsequent page-aligned virtual addresses, to produce a continuous virtual-address mapping that starts at the base address. A special page-descriptor value PD_NULL indicates that a virtual-address mapping should be removed. Thus, a thief in Cilk-M that finishes executing a series of functions that used a deep stack can map a shorter stolen stack prefix with a single system call.

This low-level design for the TLMM-Linux interface affords a scalable kernel implementation. One downside, however, is that the kernel and the runtime system must both manage page descriptors. The kernel tracks at which virtual addresses the page descriptors are mapped. The runtime tracks the mapping between page descriptors and pages mapped in the TLMM region so as to allow sharing among workers — two workers share pages by mapping the same physical pages in their respective TLMM regions. As we shall see in Chapter 3, this scenario indeed comes up in the maintenance of TLMM-based cactus stacks. We have considered an alternative interface design where the TLMM-Linux provides another level of abstraction so that the runtime does not need to keep track of the page mappings, but this interface would force the runtime system to bear additional overhead between steals, so we opted for this low-level interface instead. I will revisit this point in more detail later in Section 3.2.

The most unfortunate aspect of the TLMM scheme for solving the cactus-stack problem is that it requires a change to the operating system. Section 2.3 sketches an alternative "workers-as-processes" scheme, which, although it does not require operating-system support, has other deficiencies.

2.3 An Alternative to TLMM

Some may view TLMM as too radical an approach to implement memory abstractions, because it involves modifying the operating system. This section considers another possible memory-mapping solution that simulates the effect of TLMM which does not require operating-system support. The idea of the ***workers-as-processes*** scheme is to implement workers as processes, rather than threads, thereby allowing each worker to map its address range independently and use memory mapping to support the part of the address range that is meant to be shared. This section sketches a design for this alternative scheme and discusses its ramifications.

During the start-up of the workers-as-processes scheme, each worker uses memory-mapping to share the heap and data segments across the workers' address spaces by invoking `mmap` with a designated file descriptor on the virtual-address range of where the heap and data segments reside. Since processes by default do not share memory, this strategy provides the illusion of a fully shared address space for these segments. Since workers may need to share part of their stacks to maintain a cactus stack memory abstraction, the runtime system must also memory-map all the workers' stacks to the file, recording the file offsets for all pages mapped in the stacks so that they can be manipulated. In addition, other resources — such as the file system, file descriptors, signal-handler tables, and so on — must be shared, although at least in Linux, this sharing can be accomplished straightforwardly using the `clone` system call.

Although this workers-as-processes approach appears well worth investigating, there are a few complications that one needs to deal with if this approach is taken. Here is a summary of challenges.

First, the runtime system would incur some start-up overhead to set up the shared memory among workers. A particular complication would occur if the runtime system is initialized in the middle of a callback from C to Cilk for the first time. In this case, the runtime system must first unmap the existing heap segment used by the C computation, remap the heap segment with new pages so that the mapping is backed by a file (so as to allow sharing), and copy over the existing data from the old mapping to the new mapping.

Second, the overhead for stealing would increase. In order to maintain a cactus-stack memory abstraction, a thief must remap its stack after a successful steal, so as to reflect the stolen frame (and its ancestors) that it shares with the victim. If m is the number of pages mapped in the victim's stack that the thief must map to share, the thief might need to invoke `mmap` m times, once for each address range, rather than making a single call as with our TLMM implementation, because it is unlikely that these consecutive pages in the victim's stack reside contiguously in the designated file. These m calls would result in $2m$ kernel crossings, and thus increase the steal overhead. One might imagine an `mmap` interface that would support mapping of multiple physical pages residing in a noncontiguous range in the designated file, but such an enhancement would involve a change to the operating system, exactly what the workers-as-processes scheme tries to avoid.

Finally, and perhaps most importantly, workers-as-processes makes it complicated to support system calls that change the address space, such as `mmap` and `brk`. When one worker invokes `mmap` to map a file into shared memory, for example, the other workers must do the same. Thus, one must implement a protocol to synchronize all the workers to perform the mapping before allowing the worker that performed the `mmap` to resume. Otherwise, a race might occur, especially if the application code communicates between workers through memory. This protocol would likely be slow because of the communication it entails. Furthermore, in some existing implementation of system call libraries such as `glibc`, calling `malloc` with size larger than 128 KBytes results in invoking `mmap` to allocate a big chunk of memory. Therefore, with this scheme, one would need to rewrite the `glibc` library to intercept the `mmap` call and perform the synchronization protocol among workers for the newly allocated memory as well.

Despite these challenges, the workers-as-processes "solution" appears to be an interesting research direction. It may be that hybrid schemes exist which modify the operating system in a less intrusive manner than what TLMM does, for example, by allowing noncontiguous address ranges in mmap, by supporting mmap calls across processes, etc. We adopted TLMM's strategy of sharing portions of the page table, because we could explore a memory-mapping solution for implementing memory abstractions with relatively little engineering effort. Our work focuses more on such solution's implication on the runtime system, however, and not as much on how the memory-mapping should be supported. Most of the work described in the first part of this dissertation, including the design of the runtime system and the theoretical bounds, applies to the workers-as-processes approach as well. The Cilk-M system seems to perform well, which may motivate the exploration of other, possibly more complex strategies that have different systems ramifications.

Chapter 3

TLMM-Based Cactus Stacks

Work stealing [8, 18, 20, 21, 24, 41, 43, 45, 49, 55, 75, 80, 82, 118, 133] is fast becoming a standard way to load-balance dynamic multithreaded computations on multicore hardware. Concurrency platforms that support work stealing include Cilk-1 [18], Cilk-5 [49], Cilk++ [94], Cilk Plus [69], Fortress [6], Hood [21], Java Fork/Join Framework [90], Task Parallel Library (TPL) [92], Threading Building Blocks (TBB) [126], and X10 [26]. Work stealing admits an efficient implementation that guarantees bounds on both time and stack space [20, 49], but existing implementations that meet these bounds — including Cilk-1, Cilk-5, and Cilk++ — suffer from interoperability problems with legacy (and third-party) serial binary executables that have been compiled to use a linear stack.[1] This chapter illustrates a strategy for maintaining a cactus-stack memory abstraction, called a ***TLMM-based cactus stack***, with which one can build algorithmically sound work-stealing concurrency platforms that interoperate seamlessly with legacy serial binaries.

An execution of a serial Algol-like language, such as C [77] or C++ [130], can be viewed as a "walk" of an ***invocation tree***, which dynamically unfolds during execution and relates function instances by the "calls" relation: if a function instance A calls a function instance B, then A is a ***parent*** of the ***child*** B in the invocation tree. Such serial languages admit a simple array-based stack for allocating function activation frames. When a function is called, the stack pointer is advanced, and when the function returns, the original stack pointer is restored. This style of execution is space efficient, because all the children of a given function can use and reuse the same region of the stack. The compact linear-stack representation is possible only because in a serial language, a function has at most one extant child function at any time.

In a dynamically multithreaded language, such as Cilk-5 [49] or Cilk Plus [69], a parent function can also ***spawn*** a child — invoke the child without suspending the parent — thereby creating parallelism. The notion of an invocation tree can be extended to include spawns, as well as calls, but unlike the serial walk of an invocation tree, a parallel execution unfolds the invocation tree more haphazardly and in parallel. Since multiple children of a function may exist simultaneously, a linear-stack data structure no longer suffices for storing activation frames. Instead, the tree of extant activation frames forms a ***cactus stack*** [60], as shown in Figure 3-1. The implementation of cactus stacks is a well-understood problem for which low-overhead implementations exist [49, 51].

In all known software implementations, however, transitioning from serial code (using a linear stack) to parallel code (using a cactus stack) is problematic, because the type of stack impacts the calling conventions used to allocate activation frames and pass arguments. The property of allowing

[1] The interoperability problem is not inherent to languages that are Java-based and byte-code interpreted by a virtual-machine environment such as Fortress, Java Fork/Join Framework, TPL, and X10, because in such languages, no address to a stack frame can be captured. Some of these languages, in their current forms, do suffer from a similar problem due to implementation choices, however.

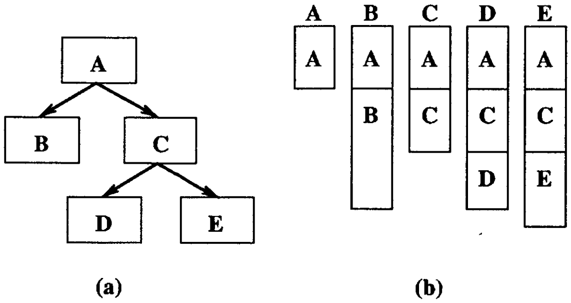


Figure 3-1: A cactus stack. **(a)** The invocation tree, where function A invokes B and C, and C invokes D and E. **(b)** The view of the stack by each of the five functions. In a serial execution, only one view is active at any given time, because only one function executes at a time. In a parallel execution, however, if some of the invocations are spawns, then multiple views may be active simultaneously.

arbitrary calls between parallel and serial code — including especially legacy (and third-party) serial binaries — is referred to as ***serial-parallel reciprocity***, or ***SP-reciprocity*** for short.

SP-reciprocity is especially important if one wishes to multicore-enable legacy object-oriented environments by parallelizing an object's member functions. For example, suppose that a function A allocates a new object x whose type has a member function foo(), which we parallelize. Now, suppose that A is linked with a legacy binary containing a function B, and A passes &x to B, which proceeds to invoke x->foo(&y), where &y is a reference to a local variable allocated in B's stack frame. Without SP-reciprocity, this simple callback does not work.

Existing work-stealing concurrency platforms that support SP-reciprocity fail to provide provable bounds on either scheduling time or consumption of stack space. These bounds typically follow those of Blumofe and Leiserson [20]. Let T_1 be the ***work*** of a deterministic computation — its serial running time — and let T_∞ be the ***span*** of the computation — its theoretical running time on an infinite number of processors. Then, a work-stealing scheduler can execute the computation on P processors in time

$$T_P \leq T_1/P + c_\infty T_\infty \, , \tag{3.1}$$

where $c_\infty > 0$ is a constant representing the ***span overhead***. As we have discussed in Section 2.1 (Cilk-M's performance model), this formula guarantees linear speedup when $P \ll T_1/T_\infty$, that is, the number P of processors is much less than the computation's ***parallelism*** T_1/T_∞. Moreover, if S_1 is the stack space of a serial execution, then the (cactus) stack space S_P consumed during a P-processor execution satisfies

$$S_P \leq PS_1 \, . \tag{3.2}$$

Generally, we shall measure stack space in hardware pages, where we leave the page size unspecified. Many systems set an upper bound on S_1 of 256 4-KByte pages.

We shall refer to the problem of simultaneously achieving the three criteria of SP-reciprocity, a good time bound, and a good space bound, as the ***cactus-stack problem***. This chapter shows how the Cilk-M system utilizes operating-system support for thread-local memory mapping (TLMM) to support full SP-reciprocity, so that a cactus stack interoperates seamlessly with the linear stack of legacy binaries, while simultaneously providing bounds on scheduling time and stack space.

The Cilk-M worker threads, which comprise the distributed scheduler, allow the user code to operate using traditional linear stacks, while the runtime system implements a cactus stack behind the scenes using TLMM support. Since TLMM allows the various worker stacks to be aligned, pointers to ancestor locations in the cactus stack are dereferenced correctly no matter which worker executes the user code.

Our prototype TLMM-Linux operating system and the Cilk-M runtime system solve the cactus-stack problem. In Cilk-M, we shall define a ***Cilk function*** to be a function that spawns, and the ***Cilk depth*** of an application to be the maximum number of Cilk functions nested on the stack during a serial execution. Suppose that an application has work T_1, span T_∞, consumes stack space S_1 on one processor, and has a Cilk depth D. Then, analogously to Inequalities (3.1) and (3.2), the Cilk-M scheduler executes the computation on P processors in time

$$T_P \leq T_1/P + c_\infty T_\infty \ , \tag{3.3}$$

where $c_\infty = O(S_1 + D)$, and it consumes stack space

$$S_P \leq P(S_1 + D) \ . \tag{3.4}$$

Inequality (3.3) guarantees linear speedup when $P \ll T_1/(S_1 + D)T_\infty$.

This chapter includes performance evaluation of Cilk-M on a variety of benchmarks, comparing it to two other concurrency platforms: the original Cilk 5.4.6, whose code base the Cilk-M runtime system is based on, and Cilk Plus, a commercial-grade implementation. These studies indicate that the time overhead for managing the cactus stack with TLMM is generally as good or better than Cilk-5 and comparable to Cilk Plus. In terms of space consumption, experimental results indicate that the per-worker consumption of stack space in Cilk-M is no more than 2.75 times the serial space requirement across benchmarks. The evaluation also includes a study on overall space cconsumption (both stack and heap) comparison between Cilk-M[2] and Cilk 5.4.6 to better understand the trade-offs made between the Cilk-M runtime implementing a TLMM-based cactus stack and the Cilk-5 runtime employing a heap-based cactus stack. Experiemental results show that the overall space consumption of Cilk-M is comparable to or better than that of Cilk-5.

The remainder of this chapter is organized as follows. Section 3.1 provides background on time and space bounds guaranteed by a work-stealing schedulers using Cilk-5 as a model and describes a range of conventional approaches that fail to solve the cactus-stack problem. Section 3.2 describes how Cilk-M leverages TLMM support to solve the cactus-stack problem. Section 3.3 analyzes the performance and space usage of the Cilk-M system both theoretically and empirically. Section 3.4 provides some concluding remarks.

3.1 The Cactus-Stack Problem Seems Hard

This section overviews challenges in supporting SP-reciprocity while maintaining bounds on space and time, illustrating the difficulties that various traditional strategies encounter. Before we dive into how various strategies fail to solve the cactus-stack problem, we shall first briefly review the theoretical bounds on space and time guaranteed by a work-stealing scheduler, using Cilk-5 [49] as an example.

Recall how a work-stealing scheduler operates from Section 2.1. For the most part, a worker pushes and pops frames from the bottom of its own deque, which mirrors precisely the behavior of C or other Algol-like languages in their use of a stack. Only when a worker runs out of work, its behavior diverges; the worker turns into a thief, randomly chooses a victim, and attempts to steal the topmost (oldest) frame from the victim worker.

[2] Here, I am referring to Cilk-M 0.9 specifically, because the way a spawn statement is compiled in Cilk-M 1.0 using the Cilk Plus compiler diverges greatly from that in Cilk-5.

Strategy	*SP-Reciprocity*	*Time Bound*	*Space Bound*
1. Recompile everything	no	very strong	very strong
2. One stack per worker	yes	very strong	no
3. Depth-restricted stealing	yes	no	very strong
4. Limited-depth stacks	yes	no	very strong
5. New stack when needed	yes	very strong	weak
6. Recycle ancestor stacks	yes	strong	weak
7. TLMM cactus stacks	yes	strong	strong

Figure 3-2: Attributes of different strategies for implementing cactus stacks.

The analysis of the Cilk-5 scheduler's performance is complicated (see [20]), but at a basic level, the reason it achieves the bound in Inequality (3.1) is that every worker is either working, in which case it is chipping away at the T_1/P term in the bound, or work-stealing, in which case it has a good probability of making progress on the T_∞ term. If the scheduler were to wait, engage in bookkeeping, or perform any action that cannot be amortized against one of these two terms, the performance bound would cease to hold, and in the worst case, result in much less than linear speedup on a program that has ample parallelism.

The analysis of the Cilk-5 scheduler's space usage is more straightforward. The scheduler maintains the so-called ***busy-leaves property*** [20], which says that at every moment during the execution, every ***extant*** — allocated but not yet deallocated — leaf of the spawn tree has a worker executing it. The bound on stack space given in Inequality (3.2) follows directly from this property. Observe that any path in the spawn tree from a leaf to the root corresponds to a path in the cactus stack, and the path in the cactus stack contains no more than S_1 space. Since there are P workers, PS_1 is an upper bound on stack space (although it may overcount). Tighter bounds on stack space have been derived for specific applications [16] using the Cilk-5 scheduler and for other schedulers [14].

Most strategies for implementing a cactus stack fail to satisfy all three criteria of the cactus-stack problem. Figure 3-2 categorizes attributes of the strategies of which I am aware. This list of strategies is not exhaustive but is meant to illustrate the challenges in supporting SP-reciprocity while maintaining bounds on space and time, and to motivate why naive solutions to the cactus-stack problem do not work. We will now overview these strategies.

The main constraint on any strategy is that once a frame has been allocated, its location in virtual memory cannot be changed, because generally, there may be a pointer to a variable in the frame elsewhere in the system. Moreover, the strategies must respect the fact that a legacy binary can act as an adversary, allocating storage on the stack at whatever position the stack pointer happens to lie. Thus, when a legacy function is invoked, the runtime system has only one "knob" to dial — namely, choosing the location in virtual memory where the stack pointer points — and there had better be enough empty storage beneath that location for all the stack allocations that the binary may choose to do. (Many systems assume that a stack can be as large as 1 MByte.) A strategy does have the flexibility to choose how it allocates memory in parallel code, that is, code that spawns, since that is not legacy code, and it can change the stack pointer. It must ensure, however, that when it invokes legacy serial code, there is sufficient unallocated storage on the stack for whatever the legacy serial code's needs might be.

Strategy 1: Recompile everything

This approach allocates frames off the heap and "eats the whole elephant" by recompiling all legacy serial functions to use a calling convention that directly supports a cactus stack. Very strong time and space bounds can be obtained by Strategy 1, and it allows serial code to call back to parallel code, as long as the serial code is recompiled to use the same calling convention that supports a cactus stack. This strategy does not provide true SP-reciprocity, however, since serial functions in legacy (and third-party) binary executables, which were compiled assuming a linear stack, cannot call back to parallel code. Cilk++ [66] employs this strategy.

An interesting alternative is to use binary-rewriting technology [88, 109, 115] to rewrite the legacy binaries so that they use a heap-allocated cactus stack. This approach may not be feasible due to the difficulty of extracting stack references in optimized code. Moreover, it may have trouble obtaining good performance because transformations must err on the side of safety, and dynamically linked libraries might need to be rewritten on the fly, which would preclude extensive analysis.

Strategy 2: One stack per worker

This strategy gives each worker an ordinary linear stack. Whenever a worker steals work, it uses its stack to execute the work. For example, imagine that a worker W_1 runs parallel function `foo`, which spawns `A`. While W_1 executes `A`, another worker W_2 steals `foo` and resumes the continuation of `foo` by setting its base pointer to the top of `foo`, which resides on W_1's stack, and setting its stack pointer to the next available space in its own stack, so that the frames of any function called or spawned by `foo` next is allocated on W_2's stack.

With Strategy 2, the busy-leaves property no longer holds, and the stacks can grow much larger than S_1. In particular, W_1 must steal work if `foo` is not yet ready to sync when W_1 returns from `A`. Since `foo` is not ready to be resumed and cannot be popped off the stack, W_1 can only push the next stolen frame below `foo`. If `foo` is already deep in the stack and W_1 happens to steal a frame shallow in the stack, then W_1's stack could grow almost as large as $2S_1$. That is not so bad if it only happens once, but unfortunately, this scenario could occur recursively, yielding impractically large stack space consumption.

Strategy 3: Depth-restricted stealing

This approach is another modification of Strategy 2, where a worker is restricted from stealing any frame shallower than the bottommost frame on its stack. Thus, stacks cannot grow deeper than S_1. The problem with Strategy 3 is that a worker may be unable to steal even though there is work to be done, sacrificing the time bound. Indeed, Sukha [131] has shown that there exist computations for which depth-restricted work-stealing exhibits at most constant speedup on P workers, where ordinary work-stealing achieves nearly perfect linear speedup. TBB [126] employs a heuristic similar to depth-restricted work-stealing to limit stack space.

Strategy 4: Limited-depth stacks

This approach is similar to Strategy 2, except that a limit is put on the depth a stack can grow. If a worker reaches its maximum depth, it waits until frames are freed before stealing. The problem with Strategy 4 is that the cycles spent waiting cannot be amortized against either work or span, and thus the time bound is sacrificed, precluding linear speedup on codes with ample parallelism.

Strategy 5: New stack when needed

This strategy, which is similar to Strategy 2, allocates a new stack on every steal. In the scenario described in Strategy 2, when W_1 goes off to steal work, Strategy 5 switches to a new stack to execute the stolen work. Thus, nothing is allocated below `foo`, which avoids the unbounded space blowup incurred by Strategy 2.

Since Strategy 5 maintains the busy-leaves property, the total physical memory used for extant frames at any given moment is bounded by PS_1. The extant frames are distributed across stacks, however, where each stack may contain as little as a single extant frame. Since each stack may individually grow as large as S_1 over time and the stacks cannot be recycled until they contain no extant frames, the virtual-address space consumed by stacks may grow up to DPS_1, where D is the Cilk depth (defined at the beginning of the chapter), a weak bound. Moreover, Strategy 5 may incur correspondingly high swap-space usage. Swap space could be reduced by directing the operating system to unmap unused stack frames when they are popped so that they are no longer backed up in the swap space on disk, but this scheme seems to laden with overhead. It may be possible to implement the reclamation of stack space lazily, however.

Cilk Plus [69] employs a heuristic that is a combination of Strategy 4 and Strategy 5 — the runtime system manages a large pool of linear stacks and uses Strategy 5 when there are still stacks available in the pool; only when the pool exhausts, the runtime system switches to a heuristic similar to Strategy 4.

Strategy 6: Reuse ancestor stacks

This scheme is like Strategy 5, but before allocating a new stack after stealing a frame, it checks whether an ancestor of the frame is suspended at a **cilk_sync** and that the ancestor is the bottom-most frame on the stack. If so, it uses the ancestor's stack rather than a new one. Strategy 6 is safe, because the ancestor cannot use the stack until all its descendants have completed, which includes the stolen frame. Although Strategy 6 may cut down dramatically on space compared with Strategy 5, it has been shown [47] to still require at least $\Omega(P^2 S_1)$ stack space for some computations. As with Strategy 7, the time bound obtained with this strategy exhibits some additional steal overhead compared to Inequality (3.2), which results from the traversal of ancestors' frames when searching for a reusable stack.

Strategy 7: TLMM-based cactus stacks

The strategy employed by Cilk-M and explored in this chapter. In particular, this strategy obtains the strong bounds given by Inequalities (3.3) and (3.4).

3.2 TLMM-Based Cactus Stacks in Cilk-M

Cilk-M leverages TLMM to solve the cactus-stack problem by modifying the Cilk-5 runtime system in two key ways. First, whereas Cilk-5 uses a heap-allocated cactus stack, Cilk-M uses a linear stack in each worker, fusing them into a cactus stack using TLMM. Second, whereas Cilk-5 uses a special calling convention for parallel functions and forbids transitions from serial code to parallel code, Cilk-M uses the standard C subroutine linkage for serial code and a compatible linkage for parallel code. This section describes how the Cilk-M runtime system implements these two modifications.

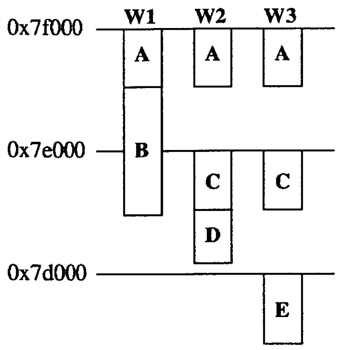


Figure 3-3: The view of stacks mapped in the TLMM region of each worker. The stack layout corresponds to the execution of the invocation tree shown in Figure 3-1. The horizontal lines indicate page boundaries, and the hexadecimal values on the left correspond to the virtual-memory addresses.

The Cilk-M cactus stack

Recall that any strategy for solving the cactus-stack problem must obey the constraint that once allocated, a stack frame's location in virtual memory cannot be moved. Cilk-M respects this constraint by causing each worker thread to execute user code on a stack that resides in its own TLMM region. Whenever a successful steal occurs, the thief memory-maps the stolen frame and the ancestor frames in the invocation tree — the ***stolen stack prefix*** — so that these frames are shared between the thief and victim. The sharing is achieved by mapping the physical pages corresponding to the stolen stack prefix into the thief's stack, with the frames occupying the same virtual addresses at which they were initially allocated. Since the physical pages corresponding to the stack prefix are mapped to the same virtual addresses, a pointer to a local variable in a stack frame references the same physical location no matter whether the thief or the victim dereferences the pointer.

Consider the invocation tree shown in Figure 3-1(a) as an example. Imagine three workers working on the three extant leaves B, D, and E. Figure 3-3 illustrates the corresponding TLMM region for each worker. Upon a successful steal, Cilk-M must prevent multiple extant child frames from colliding with each other. For instance, worker W_1 starts executing A, which spawns B and worker W_2 steals A from W_1, maps the stack prefix (i.e., the page where A resides) into its stack, resumes A, and subsequently spawns C. In this case, W_2 cannot use the portion of the page below frame A, because W_1 is using it for B. Thus, the thief, W_2 in this example, advances its stack pointer to the next page boundary upon a successful steal.

Continuing with the example, W_2 executes C, which spawns D. Worker W_3 may steal A from W_2 but, failing to make much progress on A due to a **cilk_sync**, be forced to steal again. In this case, W_3 happens to steal from W_2 again, this time stealing C. Thus, W_3 maps into its stack the pages where A and C reside, aligns its stack pointer to the next page boundary to avoid conflicting with D, resumes C, and spawns E.[3] In this example, W_1 and W_2 each map 2 pages in their respective TLMM regions, and W_3 maps 3. The workers use a total of 4 physical pages: 1 page for each of A, C, and E, and an additional page for B. Function D is able to share a page with C.

Upon a successful steal, the thief always advances its stack pointer to the next page boundary before resuming the stolen parent frame to avoid conflicting with the parallel child executing on the victim. Advancing the stack pointer causes the thief's stack to be fragmented.[4] Cilk-M mitigates

[3] Actually, this depiction omits some details, which will be elaborated more fully later in this section.

[4] An alternative strategy to prevent collision is to have workers to always spawn at a page boundary. This strategy, however, would cause more fragmentation of the stack space and potentially use more physical memory.

fragmentation by employing a ***space-reclaiming policy*** in which the stack pointer is reset to the bottom of the frame upon a joining steal or a successful sync. This space-reclaiming policy is safe, because all other parallel subcomputations previously spawned by the frame have returned, and so the executing worker is not sharing this portion of the stack with any other worker.

Since a worker's TLMM region is not addressable by other workers, one deficiency of the TLMM strategy for implementing cactus stacks is that it does not support legacy serial binaries where the stack must be visible externally to other threads. For instance, an application that uses MCS locks [108] might allocate nodes for the lock queues on the local stack, rather than out of the heap. This code would generally not work properly under Cilk-M, because the needed nodes might not be visible to other threads. This issue seems to be more theoretical than practical, however, because I am unaware of any legacy applications that use MCS locks in this fashion or otherwise need to see another worker's stack. Nevertheless, the limitation is worth noting.

Alternative TLMM interface

Section 2.2 mentioned that we have considered an alternative interface design for TLMM so that the runtime system does not need to keep track of the page mapping. In this alternative design, the TLMM interface directly provides a cactus-stack abstraction, so a thief can switch to a victim's stack with a system call that takes a stack identifier and a TLMM address as arguments. The kernel maps the pages of the victim's stack into the calling thread's TLMM region. This alternative design frees the Cilk-M runtime from tracking individual page descriptors.

There are a couple downsides to this design. First, the interface is designed specifically for building a TLMM cactus stack. Since TLMM is useful for other purposes, we preferred a more general interface over this one. One could design a more general interface, such as changing the system call to take a thread identifier and a TLMM address range instead, but the second issue is more difficult to circumvent. That is, if the runtime does not explicitly track page descriptors, both the kernel and the Cilk-M runtime need to hold locks during the map system call. For Cilk-M, this synchronization is necessary to prevent a race where, after a thief steals a frame, the victim steals a different frame and remaps its own stack before the thief can map the original stack of the victim. It is likely that the kernel would also use a lock to ensure consistency while copying page mappings from the victim's stack to the thief's stack.

The low-level interface in TLMM-Linux avoids this problem, because a thief can copy the page descriptors of its victim's stack pages at user-level before it invokes `sys_pmap`. During `sys_pmap`, since the kernel reads from and writes to only the page mappings in the calling thread's TLMM region, it does not need to acquire any locks when mapping and unmapping pages. In contrast, in the alternative design, the thief must hold the lock on the victim's deque not only to identify the pages to steal, but also while the pages are being mapped by the operating system. Thus, the alternative scheme locks out other thieves from stealing from the victim for a longer time.

Cilk-M's calling convention

TLMM allows Cilk-M to support a cactus stack in which a frame can pass pointers to local variables to its descendants, but additional care must be taken to ensure that transitions between serial and parallel code are seamless. Specifically, the parallel code must use calling conventions compatible with those used by serial code.

Before we discuss Cilk-M's calling convention, we shall digress for a moment to outline the calling convention used by ordinary C functions. The calling convention described here is based on the x86 64-bit architecture [103], the platform on which the Cilk-M system is implemented.

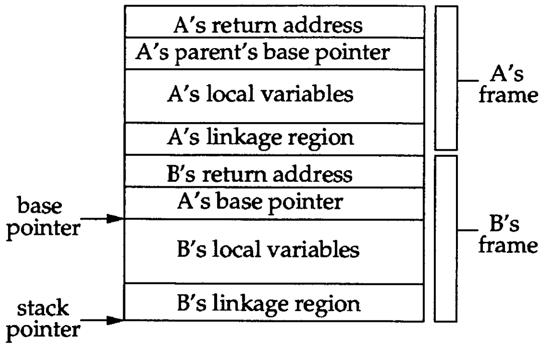


Figure 3-4: The layout of a linear stack with two frames. The figure shows a snapshot of a linear stack during execution, where A has called the currently executing function B. The figure labels the stack frame for each function on the right and marks the current base and stack pointers on the left.

Figure 3-4 illustrates the stack-frame layout for a linear stack, assuming that the stack grows downward, where a function A calls a function B. The execution begins with A's frame on the stack, where the frame contains (from top to bottom) A's return address, A's caller's base pointer, and some space for storing A's local variables and passing arguments. Typically, arguments are passed via registers. If the argument size is too large, or when there are more arguments than the available registers, some arguments are passed via memory. We shall refer to these arguments as ***memory arguments*** and the region of frame where memory arguments are passed as the ***linkage region***.

Modern compilers generate code in the function prologue to reserve enough space in a frame for its function's local variables, as well as a linkage region large enough to pass memory arguments to any potential child function that the function may invoke, which takes an extra compiler pass to compute. Thus, in this example, when A calls B, the execution simply moves values to A's linkage region. Even though this linkage region is reserved by A's prologue and is considered to be part of A's frame, it is accessed and shared by both A and A's callee (e.g., B). Function A may access the area via either a positive offset from A's stack pointer or, if the exact frame size is known at compile time, a negative offset from its base pointer. On the other hand, A's callee typically accesses this area to retrieve values for its parameters via a positive offset from its base pointer, but it could also access this area via its stack pointer if the frame size is known at compile time.

This calling convention assumes a linear stack where a parent's frame lies directly above its child's frame and the shared linkage region is sandwiched between the two frames. All children of a given function access the same linkage region to retrieve memory arguments, since the calling convention assumes that during an ordinary serial execution, at most one child function exists at a time. While these assumptions are convenient for serial code, it is problematic for parallel code employing work stealing, because multiple extant children cannot share the same linkage region. Furthermore, a gap may exist between the parent frame and the child frame in the TLMM-based cactus stack if the child frame is allocated immediately after a successful steal.

To circumvent these issues while still obeying the calling convention, a worker in Cilk-M, upon a successful steal, allocates a fresh linkage region by advancing its stack pointer a little further beyond the next page boundary.[5] This strategy allocates the linkage region immediately above the child frame and allows additional linkage region to be created only when parallel execution occurs. Since multiple linkage regions may exist for multiple extant children, some care must be taken so

[5] For simplicity, Cilk-M 0.9 reserves a fixed amount, 128 bytes, for each linkage region. Had we built a Cilk-M compiler, it would calculate the space required for each linkage region and pass that information to the runtime.

that the parent passes the memory arguments via the appropriate linkage region, which we will examine next.

Compiler invariants for Cilk functions in Cilk-M

To ensure execution correctness and to obey the Cilk-M calling convention, all the compiled Cilk functions must maintain the following invariants:

1. All memory arguments are passed via the stack pointer with positive offsets.
2. All local variables are referenced via the base pointer with negative offsets.
3. Before each **cilk_spawn** statement, all live registers are flushed onto the stack.
4. If a **cilk_sync** fails, all live registers are flushed onto the stack.
5. When resuming a stolen function after a **cilk_spawn** or **cilk_sync** statement, restore live register values from the stack.
6. When a call or spawn returns, flush the return value from the register onto the stack.
7. The frame size is fixed before any **cilk_spawn** statement.

Invariants 1 and 2 ensure correct execution in the event where a gap exists between the frames of the caller and the callee. Using the stack pointer to pass arguments to the child frame ensures that the arguments are stored right above the child frame. Similarly, the locals need to be referenced by the base pointer with negative offsets, since the stack pointer may have changed.

Invariants 3–6 ensure that a thief resuming the stolen function accesses the most up-to-date values for local variables, including return values from subroutines. This method is analogous to Cilk-5's strategy of saving execution states in heap-allocated frames [49]. Cilk-M adapts the strategy to store live values directly on the stack, which is more efficient.

Finally, although Invariant 7 is not strictly necessary, it is a convenient simplification, because it ensures that a frame is allocated in a contiguous virtual-address space. Since a frame may be stolen many times throughout the computation, if a thief were allowed to allocate more stack space upon a successful steal, the frame allocation would end up fragmented and allocated in noncontiguous virtual-address spaces.

3.3 An Evaluation of TLMM-Based Cactus Stacks

This section evaluates the TLMM-based cactus stacks in Cilk-M. First, we will study theoretical bounds on stack space and running time, which although not as strong as those of Cilk-5, nevertheless provide reasonable guarantees. Next, we will compare Cilk-M's empirical performance to that of the original Cilk-5 system and the Cilk Plus system. The results indicate that Cilk-M performs similarly to both and that the overhead for remapping stacks is modest. Cilk-M's consumption of stack space appears to be well within the range of practicality, and its overall space consumption (including stack and heap space) is comparable to that of Cilk-5.

Theoretical bounds

We shall first analyze the consumption of stack space for an application run under Cilk-M. Let S_1 be the number of pages in a serial execution of the program, let S_P be the number of pages that Cilk-M consumes when run on P workers, and let D be the Cilk depth of the application. The bound $S_P \leq P(S_1 + D)$ given in Inequality (3.4) holds, because the worst-case stack depth of a worker is $S_1 + D$ pages. This worst case occurs when every Cilk function on a stack that realizes the Cilk depth D is stolen. The stack pointer is advanced to a page boundary for each of these D stolen

Application	*Input*	*Description*
`cholesky`	4000/40000	Cholesky factorization
`cilksort`	10^8	Parallel merge sort
`fft`	2^{26}	Fast Fourier transform
`fib`	42	Recursive Fibonacci
`fibx`	280	Synthetic benchmark with deep stack
`heat`	2048×500	Jacobi heat diffusion
`knapsack`	29	Recursive knapsack
`lu`	4096	LU-decomposition
`matmul`	2048	Matrix multiply
`nqueens`	14	Count ways to place N queens
`qsort`	10^8	Parallel quick sort
`rectmul`	4096	Rectangular matrix multiply
`strassen`	4096	Strassen matrix multiply

Figure 3-5: The 13 benchmark applications.

frames, contributing an extra D to the normal number S_1 of pages in the stack. Since there are P workers, the bound follows.

As we shall see from the benchmark studies, this upper bound is loose in terms of actual number of pages. First, since different stack prefixes are shared among workers, the shared pages are double-counted. Second, we should not expect, which the benchmark studies bear out, that every frame on a stack is stolen. Moreover, the space-reclaiming policy also saves space in practice. Nevertheless, the theoretical bound provides confidence that space utilization cannot go drastically awry.

Cilk-M achieves the time bound $T_P \leq T_1/P + c_\infty T_\infty$ given in Inequality (3.3), where T_1 is the work of the program, T_∞ is its span, and $c_\infty = O(S_1 + D)$. In essense, the bound reflects the increased cost of a steal compared to the constant-time cost in Cilk-5. In the worst case, every steal might need to map a nearly worst-case stack of depth $S_1 + D$, which costs $O(S_1 + D)$ time. This thesis does not cover the full theoreticl arguments required to prove this bound; it can be proved using the techniques of [8] and [20], adapted to consider the extra cost of stealing in Cilk-M.

As with the space bound, the time bound is loose, because the worst-case behavior used in the proof is unlikely. One would not normally expect to map an entire nearly worst-case stack on every steal. Nevertheless, the bound provides confidence, because applications with sufficient parallelism are guaranteed to achieve near-perfect linear speedup on an ideal parallel computer, as is assumed by prior theoretical studies.

Empirical studies

Theoretical bounds alone, especially those based on asymptotic analysis, do not suffice to predict whether a technology works in practice, where the actual values of constants matter. In particular, my collaborators and I had two main concerns when we started this work. The first concern was whether the cost of entering and exiting the kernel would be too onerous to allow a memory-mapping solution to the cactus-stack problem. The second concern was whether the fragmentation of the stack would consume too much space, rendering the solution impractical.

To address the first concern, we compared the performance of Cilk-M with Cilk-5 and Cilk Plus empirically on 13 applications. The benchmark results indicate that Cilk-M performs similarly with the two systems, with Cilk-M sometimes outperforming Cilk-5 despite the additional overhead for remapping the stacks.

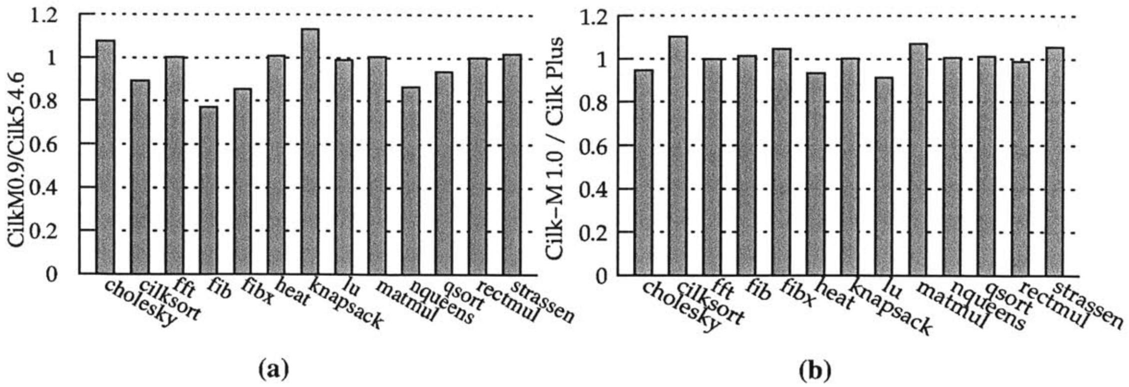
To address the second concern, we profiled the stack space of the applications running on Cilk-M with 16 cores. The data from this experiment indicate that the per-worker consumption of stack space on these benchmarks was at most a factor of 2.75 more than the serial space requirement, which is modest. Due to the fragmentation of the stack, Cilk-M indeed has higher stack space overhead than Cilk-5; as a trade-off, however, Cilk-5 tends to consume more heap space than Cilk-M due to the use of a heap-allocated cactus stack. To better understand the trade-offs made between the two runtime systems, we profiled the stack and heap space consumption of each system running the applications with 16 cores. The benchmark results indicate that the additional stack space overhead in Cilk-M is inexpensive when one considers the overall space consumption. We did not compare the overall space consumption between Cilk-M and Cilk Plus, because Cilk Plus does not provide guarantees on space consumption. Moreover, at the time when we performed the evaluation, the source for the Cilk Plus runtime system was not available, making it difficult to perform such an evaluation.

General setup. We compared Cilk-5 with Cilk-M 0.9 and compared Cilk Plus with Cilk-M 1.0 (the differences between the versions are described in Section 2.1). We compared Cilk-5 with Cilk-M 0.9 instead of with Cilk-M 1.0, because the way that a spawn statement is compiled in Cilk-M 1.0 markedly differs from that in Cilk-5 — besides the fact that Cilk-5 uses a heap-based cactus stack, the Cilk Plus compiler generates an additional function wrapper around each spawn statement [68]. Whereas applications for Cilk-M 0.9 were hand-compiled to force the compiler to generate the desired assembly code, following the invariants described in Section 3.2, applications for Cilk-5 were compiled with the source-to-source translator included in the Cilk-5 distribution to produce C postsource. The postsources for both systems were compiled with gcc 4.3.2 using `-O2` optimization. On the other hand, applications for Cilk-M 1.0 and Cilk Plus were compiled using the Cilk Plus compiler version 12.0.0 using `-O2` optimization; the runtime system constitutes the only difference.

The system was evaluated with 13 benchmark applications, all of which are included in the Cilk-5 distribution except `fibx`, which is a synthetic benchmark devised to generate large stacks. Figure 3-5 provides a brief description of each application. In addition, we modified the `knapsack` benchmark to allow for more deterministic timing. The `knapsack` from the distribution uses pruning, which causes high variance among parallel execution times, because whether a branch gets pruned or not depends on what is the best value found so far, which can differ from run to run due to scheduling. Thus, we removed the pruning in the `knapsack` benchmark for the evaluation.

All experiments were conducted on an AMD Opteron system with 4 quad-core 2 GHz CPU's having a total of 8 GBytes of memory. Each core on a chip has a 64-KByte private L1-data-cache and a 512-KByte private L2-cache, but all cores on a chip share a 2-MByte L3-cache.

Relative performance. Figure 3-6 (a) compares the performance of the applications run on Cilk-M 0.9 and Cilk-5. For each application we measured the mean of 10 runs on each of Cilk-M 0.9 and Cilk-5, and the mean on each has standard deviation less than 3%. The mean for Cilk-M 0.9 is normalized by the mean for Cilk-5. Cilk-M 0.9 performs similarly to Cilk-5 for most of the applications and is sometimes faster. The performance difference can be accounted partially by the differences in the compiled code, which accounts for the fact that Cilk-5 uses a heap-based cactus stack, and Cilk-M simply flushes variables to linear stacks. For instance, when executed on a single processor, `fib`, `fibx`, and `nqueen` execute faster on Cilk-M 0.9 than on Cilk-5, while `cholesky` and `knapsack` execute slower on Cilk-M 0.9. Figure 3-6 (b) compares the performance of the same set of applications run on Cilk-M 1.0 and Cilk Plus. Again, Cilk-M 1.0 performs similarly to



(a) **(b)**

Figure 3-6: **(a)** The relative execution time of Cilk-M 0.9 compared to Cilk-5 for 13 Cilk applications on 16 cores. **(b)** The relative execution time of Cilk-M 1.0 compared to Cilk Plus for 13 Cilk applications on 16 cores. Each value is calculated by normalizing the execution time of the application on Cilk-M with the execution time of the application on Cilk-5 and Cilk Plus respectively.

Cilk Plus. These results indicate that the additional overhead in Cilk-M for remapping the stacks is modest and does not impact application performance in general. Moreover, the good performance on `fib`, which involves mostly spawning and function calls and little computation *per se*, indicates that the Cilk-M linear-stack-based calling convention is generally superior to the Cilk-5 heap-based one.

Space utilization. Figure 3-7 shows the stack space consumption of the benchmark applications running on Cilk-M 0.9 and Cilk-M 1.0 with 16 cores. Since the consumption of stack pages depends on scheduling, it varies from run to run. Each application was run 10 times and the data shows the maximum number of pages used. Overall, the applications used less space than predicted by the theoretical bound, and sometimes much less, confirming the observation that the upper bound given in Inequality (3.4) is loose. Indeed, none of the applications used more than 2.75 times the stack space per worker of the serial stack space.

Figure 3-8 shows the stack and heap space consumptions of the benchmark applications running on Cilk-M 0.9 and on Cilk-5 with 16 workers. Both runtime systems employ an internal memory allocator that maintains local memory pools for workers to minimize contention and a single global pool to rebalance the memory distribution between local pools. The heap consumption is measured by the total number of physical pages requested by the memory allocator from the operating system at the end of the execution.[6] Again, we ran each application 10 times and recorded the maximum number of pages used.

Across all applications, Cilk-M 0.9 uses about 2–4 times, and in one case (i.e., `qsort`) 5 times more pages on the stack, than that of Cilk-5 due to fragmentation resulting from successful steals. The additional space overhead caused by fragmentation is never referenced by the runtime or the user code, however, and thus the additional stack space usage does not cause memory latency. On the other hand, Cilk-5 tends to use comparable or slightly more heap space than used by Cilk-M 0.9 (less than 3 times more), except for one application, `fft`. Since `fft` contains some machine generated code for the base cases, the Cilk functions in `fft` contain large number of temporary local variables that are used within the functions but not across **cilk_spawn** statements. The `cilk2c`

[6] This measurement does not include space for the runtime data structures allocated at the system startup, which is relatively small, comparable between the two systems, and stays constant with respect to the number of workers.

		Space Usage for Cilk-M 0.9				*Space Usage for Cilk-M 1.0*			
Application	D	S_1	$S_{16}/16$	*ratio*	S_1+D	S_1	$S_{16}/16$	*ratio*	S_1+D
`cholesky`	12	2	3.25	1.63	14	3	3.56	1.19	15
`cilksort`	18	2	3.06	1.51	20	3	3.63	1.21	21
`fft`	22	4	3.81	0.95	26	6	4.81	0.80	28
`fib`	43	2	3.44	1.72	45	4	4.50	1.13	47
`fibx`	281	8	8.44	1.05	289	22	18.81	0.86	303
`heat`	10	2	2.44	1.22	12	2	2.88	1.44	12
`knapsack`	30	2	2.88	1.44	32	4	4.13	1.03	34
`lu`	10	2	3.06	1.53	12	2	3.31	1.66	12
`matmul`	22	2	3.38	1.69	24	3	3.88	1.29	25
`nqueens`	16	2	3.31	1.66	18	3	3.50	1.17	19
`qsort`	58	2	5.50	2.75	60	6	5.93	0.99	64
`rectmul`	27	2	4.00	2.00	29	4	4.69	1.17	31
`strassen`	8	2	3.00	1.50	10	2	3.56	1.78	10

Figure 3-7: Consumption of stack space per worker for 13 Cilk applications running on Cilk-M 0.9 and Cilk-M 1.0, as measured in 4-KByte pages. The value D is the Cilk depth of the application. The serial space S_1 was obtained by running the application on one processor. The value S_{16} was measured by taking the maximum of 10 runs on 16 cores. Shown is the average space per worker $S_{16}/16$. The value *ratio* is the ratio between average space per worker when running on 16 processors and the serial space usage when running on one processor, i.e., $(S_{16}/16)/S_1$. Finally, the S_1+D column shows the theoretical upper bound for consumption of stack space per worker from Inequality (3.4).

compiler used by Cilk-5 faithfully generates space for these variables on the heap-allocated cactus stack, resulting in large heap space usage. With the same program, Cilk-M 0.9 uses the same amount of stack space for these temporary local variables as however much space a C compiler would allocate for them. Finally, when comparing the overall space consumption, Cilk-M 0.9 tends to use less space than Cilk-5, except for `fib`, `fibx`, and `qsort`. The Cilk functions in these applications have very few local variables, and therefore their heap-allocated cactus stack in Cilk-5 consumes relatively little space. Furthermore, `fibx` is a synthetic benchmark that we devised to generate large stacks (i.e., with large Cilk depth), so Cilk-M 0.9 ends up having a deep stack for `fibx`.

3.4 Conclusion

From an engineering perspective, this chapter laid out some choices for implementers of work-stealing environments. There seem to be four options for solving the cactus-stack problem: sacrificing interoperability with binaries that assume a linear-stack calling convention, sacrificing a time bound, sacrificing a space bound, and coping with a memory-mapping solution similar to those laid out in this paper.

Sacrificing interoperability limits the ability of the work-stealing environment to leverage past investments in software. An engineering team may be willing to sacrifice interoperability if it is developing a brand-new product, but it may be more cautious if it is trying to upgrade a large codebase to use multicore technology.

Sacrificing the time or space bound may be fine for a product where good performance and resource utilization are merely desirable. It may be unreasonable, however, for a product hoping to meet a hard or soft real-time constraint. Moreover, even for everyday software where fast performance is essential for good response times, time and space bounds provide a measure of predictability.

	Cilk-M	*Cilk-5*	*Cilk-M*	*Cilk-5*	*Cilk-M*	*Cilk-5*
Application	S_{16}	S_{16}	H_{16}	H_{16}	*Sum*	*Sum*
`cholesky`	52	16	193	345	245	361
`cilksort`	49	16	201	265	250	281
`fft`	61	48	169	1017	230	1065
`fib`	55	16	169	185	224	201
`fibx`	135	64	217	217	352	281
`heat`	39	16	185	273	224	289
`knapsack`	46	16	161	368	207	384
`lu`	49	16	177	265	226	281
`matmul`	54	16	169	265	223	281
`nqueens`	53	16	161	249	214	265
`qsort`	88	16	193	192	281	208
`rectmul`	76	32	169	240	245	272
`strassen`	48	16	161	417	209	433

Figure 3-8: Comparison of the overall stack and heap consumptions between Cilk-M 0.9 and Cilk-5 for 13 Cilk applications running with 16 workers. The values were measured by taking the maximum of 10 runs on 16 cores, and measured in 4-KByte pages. The last two columns show the sum of the stack and heap space consumptions for the two systems.

Coping with memory mapping by modifying the operating system may not be possible for those working on closed operating systems which they cannot change, but it may be fine for applications running on an open-source platform. Moreover, as multicore platforms grow in importance, future operating systems may indeed provide TLMM-like facilities to meet the challenges. In the shorter term, if it is not possible to modify the operating system, it may still be possible to implement a workers-as-processes scheme as described in Section 2.3 in order.

The particular engineering context will shape which option is the most reasonable, and in developing the case for a memory-mapped solution to the cactus-stack problem, we have placed an important new option on the table.

Chapter 4

Memory-Mapped Reducer Hyperobjects

Reducer hyperobjects (or ***reducers*** for short) [48] have been shown to be a useful linguistic mechanism to avoid "determinacy races" [42] (also referred as "general races" [116]) in dynamic multithreaded programs. Reducers allow different logical branches of a parallel computation to maintain coordinated local views of the same nonlocal variable. Whenever a reducer is updated — typically using an associative operator — the worker thread on which the update occurs maps the reducer access to its local view and performs the update on that local view. As the computation proceeds, the various views are judiciously ***reduced*** (combined) by the runtime system using an associative ***reduce*** operator to produce a final value.

Although existing reducer mechanisms are generally faster than other solutions for updating nonlocal variables, such as locking and atomic-update, they are still relatively slow. Concurrency platforms that support reducers, specifically Intel's Cilk Plus [69] and Cilk++ [94], implement the reducer mechanism using a ***hypermap approach*** in which each worker employs a thread-local hash table to map reducers to their local views. Since every access to a reducer requires a hash-table lookup, operations on reducers are relatively costly — about 12× overhead compared to an ordinary L1-cache memory access. Perhaps not surprisingly, besides the TLMM-based cactus stacks, the TLMM mechanism can be used to build other types of memory abstractions, such as reducer hyperobjects. This chapter investigates a ***memory-mapping approach*** for supporting reducers, which employs the thread-local memory mapping (TLMM) mechanism as described in Section 2.2 to improve the performance of reducers. The memory-mapping reducer mechanism leverages the efficient virtual-address translation, mapping reducers to local views.

A memory-mapping reducer mechanism must address four key questions:

1. What operating-system support is required to allow the virtual-memory hardware to map reducers to their local views?
2. How can a variety of reducers with different types, sizes, and life spans be handled?
3. How should a worker's local views be organized in a compact fashion to allow both constant-time lookups and efficient sequencing during reductions?
4. Can a worker efficiently gain access to another worker's local views without extra memory mapping?

The memory-mapping approach answers each of these questions using simple and efficient strategies.

1. The operating-system support employs TLMM, which enables the virtual-memory hardware to map the same virtual address to different views in the different worker threads, allowing reducer lookups to occur without the overhead of hashing.

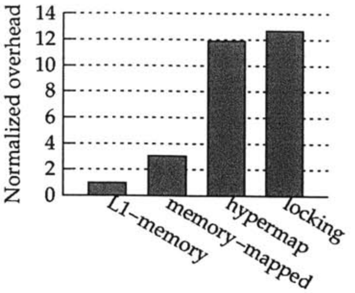


Figure 4-1: The relative overhead for ordinary L1-cache memory accesses, memory-mapped reducers, hypermap reducers, and locking. Each value is calculated by the normalizing the average execution time of the microbenchmark for the given category by the average execution time of the microbenchmark that performs L1-cache memory accesses.

2. The thread-local region of the virtual-memory address space only holds pointers to local views and not the local views themselves. This ***thread-local indirection*** strategy allows a variety of reducers with different types, sizes, and life spans to be handled.
3. A ***sparse accumulator (SPA)*** data structure [50] is used to organize the worker-local views. The SPA data structure has a compact representation that allows both constant-time random access to elements and sequencing through elements stored in the data structure efficiently.
4. By combining the thread-local indirection and the use of the SPA data structure, a worker can efficiently transfer a view to another worker. This support for efficient ***view transferal*** allows workers to perform reductions without extra memory mapping.

I implemented the memory-mapping reducer mechanism in the Cilk-M runtime system, which supports a much more efficient reducer lookup than the existing hypermap approach. Figure 4-1 graphs the overheads of ordinary accesses, memory-mapped reducer lookups, and hypermap reducer lookups on a simple microbenchmark that performs additions on four memory locations in a tight `for` loop, executed on a single processor. The memory locations are declared to be `volatile` to avoid the compiler from optimizing the memory accesses into register accesses. Thus, the microbenchmark measures the overhead of L1-cache memory accesses. For the memory-mapped and hypermap reducers, one reducer per memory location is used. The figure also includes the overhead of locking for comparison purpose — one `pthread_spin_lock` per memory location is employed, where the microbenchmark performs lock and unlock around the memory updates on the corresponding locks.[1] The microbenchmark was run on a AMD Opteron processor 8354 with 4 quad-core 2 GHz CPU's with a total of 8 GBytes of memory and installed with Linux 2.6.32. As the figure shows, a memory-mapped reducer lookup is roughly 3× slower than an ordinary L1-cache memory access and almost 4× faster than the hypermap approach (and as we shall see in Section 7.4, the differences between the two increases with the number of reducers). The overhead of locking is similar but slightly worse than the overhead of a hypermap reducer lookup.

A memory-mapped reducer admits a lookup operation that essentially translates to two memory accesses and a predictable branch, which is more efficient than that of a hypermap reducer. An unexpected byproduct of the memory-mapping approach is that it provides greater locality than the hypermap approach, which leads to more scalable performance.

As an orthogonal issue, the reducer mechanisms in Cilk++ and Cilk Plus do not support parallelism within a reduce operation. In Cilk-M, this limitation has been lifted. This chapter also explores runtime support necessary to enable parallelism within a reduce operation. Since there is no fundamental reason why the hypermap approach cannot support a parallel reduce operation, one should be able to apply the same runtime support to allow for parallel reduce operations for the hypermap approach.

[1] The use of locks does not exactly solve the same problem as the use of reducers, i.e., determinacy races, because locking does not guarantee a deterministic ordering in updates to a shared variable. Nevertheless, locking is a commonly used synchronization mechanism. Thus, I include the overhead for locking here for comparison purposes.

```
std::list<Node *> l;
bool has_property(Node *n);
// ...
void traverse(Node *n) {
    if(n) {
        if(has_property(n)) {
            l.push_back(n);
        }
        traverse(n->left);
        traverse(n->right);
    }
}
```

Figure 4-2: C++ code to traverse a binary tree and create a list of all nodes that satisfy a given property in pre-ordering.

The rest of the chapter is organized as follows. Section 4.1 provides the necessary background on reducer semantics, which includes the reducer interface and guarantees. Section 4.2 reviews runtime support for the hypermap approach, as implemented in Cilk Plus and Cilk++. Section 4.3 describe the design and implementation of the memory-mapped reducers, addressing each of the four questions raised above in detail. Section 4.4 presents the empirical evaluation of the memory-mapped reducers by comparing it to the hypermap reducers. Finally, Section 4.5 provides some concluding remark.

4.1 Reducer Linguistics

The use of ***nonlocal variables***, variables bound outside of the scope of the function, method, or class in which they are used, is prevalent in serial programming. While the use of nonlocal variable is considered bad practice in general [137], programmers often find them convenient; for instance, using nonlocal variables avoids parameter proliferation — allowing a leaf routine to access a nonlocal variable eliminates the need of passing the variable as parameters through all function calls that lead to the leaf routine.

In parallel programming, the use of nonlocal variables may prohibit otherwise independent "strands" from executing in parallel, because they constitute a source of races. Henceforth, we shall use ***strand*** to refer to a piece of serial code that contains no keywords for parallel control. When one naively parallelizes a serial program that uses nonlocal variables, the use of nonlocal variables tends to introduce ***determinacy races*** [42] (also called ***general races*** [116]), where logically parallel strands access some shared memory location.

As an example, let's consider parallelizing the code shown in Figure 4-2 that traverses a binary tree and creates a list of all nodes that satisfy some given property in a *pre-order* fashion. The code checks and appends the current node onto the output list if the node satisfies the given property and subsequently traverses the node's left and right children. Ideally, one would like to parallelize this program by simply traversing the left and right children in parallel; care must be taken, however, to resolve the determinacy race on the list `l`, because now the left- and right-subtree traversals may potentially append to the list in parallel.

One may wish to avoid the race by protecting the accesses to the list `l` using a mutual-exclusion lock. This solution does not work correctly, however, since the code no longer maintains the pre-ordering among nodes inserted into the list. Furthermore, even if one does not care about the ordering of nodes in the list, the contention on the lock limits parallelism and may create a performance bottleneck if there are many nodes in the tree that satisfy the given property.

```
 1  bool has_property(Node *n);
 2  list_append_reducer<Node *> l;
 3  // ...
 4  void traverse(Node *n) {
 5     if(n) {
 6         if(has_property(n))
 7            l->push_back(n);
 8         cilk_spawn traverse(n->left);
 9         traverse(n->right);
10         cilk_sync;
11     }
12  }
```

Figure 4-3: A correct parallelization of the C++ code shown in Figure 4-2 using a reducer hyperobject with the original reducer interface.

A possible fix is to duplicate the list — one can restructure the code such that the `traverse` function creates a new list at every recursion level, so that every subtree has its own local copy of the list for insertion. The `traverse` function can then insert itself and appends the lists returned by its left and right children to create the final list to return, with nodes in proper order. While this strategy works correctly, it requires restructuring the code, and creating lists and moving nodes from one list to another at every recursion level, which can become expensive rather quickly.

Reducer hyperobjects (or reducers for short) proposed by Frigo et al. [48] provide a linguistic mechanism to avoid such determinacy races in a dynamically multithreaded computation. By declaring the nonlocal variable to be a reducer, the underlying runtime system coordinates parallel updates on the reducer variable, thereby avoiding determinacy races. Figure 4-3 shows a correct parallelization of the `traverse` function that employs a reducer to avoid determinacy race — the code simply declares `l` to be a reducer that performs list append (line 2). By declaring the list `l` to be a reducer, parallel accesses to `l` are coordinated, and the code produces deterministic output that is identical to the result from a serial execution.

Intuitively, the reducer mechanism works almost like the strategy that duplicates the output list, except more efficiently and that the programmer is not required to restructure the code. That is, copies of the list are created lazily only when necessary, and the underlying runtime system handles the list combining implicitly.

Not every type of object can be declared as a reducer and produce deterministic output. Conceptually, a reducer is defined in terms of an algebraic ***monoid***: a triple $(T, \otimes, e)$, where T is a set, and $\otimes$ is an binary associative operation over T with identity e. Example monoids include summing over integers with identity 0, logical `AND` with identity `true`, and list append with identity empty list such as in the example. Nevertheless, concurrent accesses to a reducer are coordinated, and the output is guaranteed to retain serial semantics as long as the reducer is updated using only its corresponding associative binary operator.

4.2 Support for Reducers in Cilk Plus

This section overviews the implementation of the Cilk++ [94] and Cilk Plus [69] reducer mechanism, which is based on hypermaps. Support for reducers was first proposed in [48] and implemented in Cilk++, and the implementation in Cilk Plus closely follows that in Cilk++. This section summarizes the runtime support relevant for comparing the hypermap approach to the memory-mapping approach. I refer interested readers to the original article [48] for full details on the hypermap approach.

The reducer library and runtime API

Support for reducers in Cilk Plus is implemented purely as a C++ template library without compiler involvement. The user invokes functions in the runtime system, and the runtime system calls back to user-defined functions according to an agreed-upon API [70]. The type of a reducer is dictated by the monoid it implements and the type of data set that the monoid operates on. The reducer library implements the monoid interface and provides two important operations that the runtime invokes: IDENTITY, which creates an identity view for a given reducer, and REDUCE, which implements the binary associative operator that reduces two views. A user can override these operations to define her own reducer types.

Maintenance of views

During parallel execution, concurrent accesses to a reducer cause the runtime to generate and maintain multiple views for a given reducer hyperobject, thereby allowing each worker to operate on its own local view. A reducer is distinctly different from the notion of ***thread-local storage*** (or ***TLS***) [129], however. Unlike TLS, a worker may create and operate on multiple local views for a given reducer throughout execution. New identity views for a given reducer may be created whenever there is parallelism, because the runtime must ensure that updates performed on a single view retain serial semantics. In that sense, a local view is associated with a particular execution context but not with a particular worker. Consequently, a hypermap that contains local views is not permanently affixed to a particular worker, but rather to the execution context.

To see how local views are created and maintained, let's consider how views are maintained with respect to the the main keywords for parallel control, **cilk_spawn** and **cilk_sync**.[2] Upon a **cilk_spawn**, the spawned child owns the view l owned by its parent before the **cilk_spawn**. On the other hand, the continuation of the parent owns a new view l', initialized to identity using IDENTITY. When a spawned child returns, the parent owns the child's view l, which is reduced with the parent's previous view l' sometime before **cilk_sync**, where l is assigned with $l \otimes l'$ and l' is destroyed. Once **cilk_sync** executes successfully, the parent owns the same view it owned before all the **cilk_spawn** statements, and any newly created view has been reduced into it.

The Cilk Plus runtime, like Cilk-5, follows the lazy task creation strategy [80] — whenever a worker encounters a **cilk_spawn**, it invokes the child and suspends the parent, pushing the parent frame onto its deque, so as to allow the parent frame to be stolen.[3] If the parent is never stolen, once the spawned child return, the continuation of the parent resumes with child's view l. In this case, the new view l' from the parent's continuation is essentially an identity, in which case, no reduce operation is necessary. Thus, the runtime is able to perform a key optimization that in a serial execution, no new views are ever created. Since a worker's behavior mirrors the serial execution between each successful steal, no new views are created when a worker is executing within a ***trace***, i.e., a sequence of consecutive strands that a worker executes between steals.

The following concrete example illustrates when views are created. Imagine an execution of the traverse code in Figure 4-3 on an input binary tree with 15 nodes, executed by three workers, W_1 (gray), W_2 (white), and W_3 (black). Figure 4-4 illustrates how the execution unfolds under one possible scheduling, where the nodes' coloring indicates which worker invoked the `traverse` function on a given node. This particular execution divides the computation into four traces, and four views exist at the end of the execution: the leftmost view l, and three additional views created via

[2] The **cilk_for** construct is effectively desugared into code containing **cilk_spawn** and **cilk_sync**, so we don't need to consider **cilk_for** here.

[3] See Section 2.1 for a more thorough review of how a work-stealing scheduler operates.

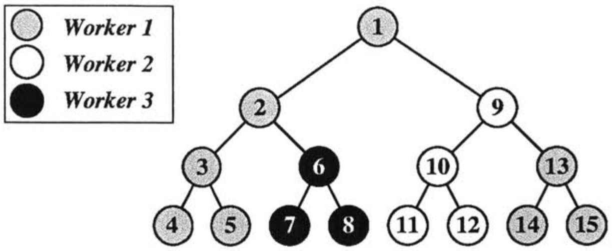


Figure 4-4: The graphical representation of an execution of the code shown in Figure 4-3. The input binary tree has 15 nodes. The coloring of the nodes indicate which worker initiate the traversal at a given node.

IDENTITY for traces that traverse nodes 6–8 (l_2), nodes 9–12 (l_3), and nodes 13–15 (l_4) respectively. The serial semantics is preserved on the final output as long as the views are combined in the order of $l \otimes l_2 \otimes l_3 \otimes l_4$, disregarding which pair gets reduced first. Note that, even though W_1 (gray) traversed nodes 1–5 and nodes 13–15, it acquired a new view for the latter traversal, because the semantic guarantee of a reducer dictates that updates must be accumulated in the order that respects the serial semantics.

Maintenance of hypermaps

A worker's behavior precisely mimics the serial execution between successful steals. Logical parallelism morphs into true parallelism when a thief steals and resumes a function (the continuation of the parent created by a spawn). Whenever a Cilk function is stolen, its frame is ***promoted*** into a ***full frame***, which contains additional bookkeeping data to handle the true parallelism created, including hypermaps that contain local views. Specifically, each full frame may contain up to 3 hypermaps — the ***user hypermap***, ***left-child hypermap***, and ***right-sibling hypermap*** — each of which respectively contains local views generated from computations associated with the given frame, its leftmost child, and its right sibling.

During parallel execution, a worker performs reducer-related operations on the user hypermap stored in the full frame sitting on top of its deque (since everything below the full frame mirrors the serial execution). The hypermap maps reducers to their corresponding local views that the worker operates on. Specifically, the address of a reducer is used as a key to hash the local view. Whenever a full frame is stolen, its original user hypermap is left with its child executing on the victim, and an empty user hypermap is created, which corresponds to the fact that new identity views must be created for the stolen frame. When the worker encounters a reducer declaration which creates a reducer hyperobject, the executing worker inserts a key-value pair into its hypermap, with the key being the address of the reducer and the value being the initial view created along with the initialization of the reducer, referred as the ***leftmost view***. When a reducer goes out of scope, at which point only its leftmost view should remain reflecting all updates, the worker removes the key-value pair from its hypermap. Finally, whenever the worker encounters an access to a reducer in the user code, the worker performs a lookup in its hypermap and returns the corresponding local view. If nothing is found in the hypermap (the user hypermap starts out empty when the frame is first promoted), the worker creates and inserts an identity view into the hypermap and returns the identity.

The other two hypermaps are placeholders. They store the accumulated values of the frame's terminated right siblings and terminated children, respectively. Whenever a frame is promoted, an additional set of local views may be created to accumulate updates from the computation associated with the frame. These views must be reduced either with views from its left sibling or parent at some point, in the order that retain serial semantics. When a frame F_1 executing on W_1 is terminating (i.e., returning), however, its sibling or parent F_2 may still be running, executed by another worker W_2. To avoid interfering with W_2 executing F_2, W_1 simply deposits its set of local views stored in F_1's user hypermap into F_2's left-child or right-sibling hypermap placeholder, depending on the relation between F_1 and F_2. The process of one worker depositing its local views into a frame running on another worker is referred to as the ***view transferal***, which more generally, refers to the process of transferring ownership of local views from one worker to another. Similarly, before W_1 can perform view transferal from F_1 to F_2, it may find a second set of local views stored in F_1's left-child or right-sibling hypermap placeholders. If so, W_1 must reduce the two sets of views together — iterate through each view from one hypermap, lookup the corresponding view in another hypermap, and reduce the two into one. This process is referred as the ***hypermerge*** process.

To facilitate hypermerges, a full frame F also contains pointers to its left sibling (or parent if F is the leftmost child), right sibling and first child. These pointers form a left-child right-sibling representation of the spawn tree, referred as the ***steal tree***, since hypermerges always occur between an executing full frame and its parent or its siblings.

There are three possible scenarios when hypermerges can occur. The first scenario is, as described above, when a full frame F returns from a **cilk_spawn**. The executing worker must perform hypermerges until it has only one set of local views left to deposit, which involves merging F's user hypermap with F's right-sibling hypermap (if not empty), and/or merging F's user hypermap with another hypermap already stored in its left-sibling or parent's placeholder, where the view transferal must occur. The second scenario is when a full frame F executes a **cilk_sync** successfully. F's executing worker must hypermerge F's left-child hypermap with F's user hypermap and store the resulting views into F's user hypermap, so as to allow F to continue execution after **cilk_sync** using views stored in the user hypermap. The last scenario is after a successful joining steal, where the last spawned child returning resumes the execution of its parent function, passing the **cilk_sync** at which the parent was stalled. A successful joining steal is semantically the same as executing a **cilk_sync** successfully. Thus, the hypermerges that occur here are the same as the hypermerges that occur when a child returns and executes a **cilk_sync** successfully.

In all cases, an executing worker performs a hypermerge in ways that maintain the serial semantics. During a hypermerge between a full frame F's user hypermap and F's right-sibling hypermap, local views in F's right-sibling hypermap are always reduced as the right of the binary associative operators, because these views come logically after the ones stored in the user hypermap. On the other hand, during a hypermerge between a full frame F's left-child hypermap and F's user hypermap, local views in F's left-child hypermap are reduced as the left of the binary associative operators, because these views come logically before the views stored in the user hypermap. Finally, hypermerges occur during view transferal follow the same logical. A hypermap being deposited in a full frame F's left-child or right-sibling hypermap placeholder comes logically after any hypermap already stored in the placeholder, and thus its local views are reduced as the right to the binary associative operators.

Preventing races during frame elimination

Workers eliminating $F.lp$ and $F.r$ might race with the elimination of F. To resolve these races, Frigo et al. [48] describe how to acquire abstract locks between F and these neighbors, where an abstract lock is a pair of locks that correspond to an edge in the steal tree. Since Frigo et al. assumes that REDUCE is a constant operation, their locking protocol holds locks during the hypermerges that must be performed before elimination. Leiserson and Schardl [96] describe a modified locking protocol to allow hypermerges to take place without holding the locks while preventing races.

4.3 Support for Reducers in Cilk-M

This section describes the memory-mapping reducer mechanism implemented in Cilk-M. The implementation of the memory-mapping reducer mechanism partially follows what's described in Section 4.2, such as the reducer library and runtime API, how views are maintained with respect to the keywords for parallel control, and how the runtime maintains the ordering in which the sets of views are reduced. Nevertheless, to enable a memory-mapped reducer, the Cilk-M runtime system must address the four questions raised at the beginning of the chapter. This section describe in detail Cilk-M's strategy for addressing each of these questions. Finally, as a orthogonal issue, this section also presents how the Cilk-M runtime supports a parallel REDUCE operation.

A reducer region using thread-local memory mapping

The first question is what operating-system support is required to allow the virtual-memory hardware to map reducers to their local views. The premise of the memory-mapping reducer mechanism is to utilize the virtual-address hardware to perform the address translation, mapping a reducer to different local views for different worker. That means, different workers must be able to map different physical pages within the same virtual address range, so the same global virtual address can map to different views for different workers. On the other hand, part of the address space must be shared to allow workers to communicate with each other and enable parallel branches of the user program to share data on the heap. In other words, this memory-mapping approach requires part of the virtual address space to be ***private***, in which workers can map independently with different physical pages, while the rest being ***shared***, in which different workers can share data allocated on the heap as usual. This mixed sharing mode is precisely what the thread-local memory mapping mechanism (TLMM) provides. Cilk-M, which already employs TLMM to build a cactus stack, provides an ideal platform for experimenting with the memory-mapping reducer mechanism.

I added the memory-mapping reducer mechanism to Cilk-M, which now utilizes the TLMM region for both the cactus stack and memory-mapped reducers. Since a stack naturally grows downward, and the use of space for reducers is akin to the use of heap space, at system start-up, the TLMM region is divided into two parts — the cactus stack is allocated at the highest TLMM address possible, growing downwards, and the space reserved for reducers starts at the lowest TLMM address possible, growing upwards. The two parts can grow as much as needed, since as a practical matter in a 64-bit address space, the two ends will never meet.

Thread-local indirection

The second question is how the memory-mapping reducer mechanism handles a variety of reducers with different types, sizes, and life spans. We shall first examine a seemingly straightforward approach for leveraging TLMM to implement reducers and see what problems can arise. In this scheme, whenever a reducer is declared, the runtime system allocates the reducer at a virtual address in the TLMM region that is globally agreed upon among all workers. It instructs each worker to map the physical page containing its own local view at that virtual address. Thus, accesses to the reducer by a worker operate directly on the worker's local view.

Although this approach seems straightforward, it fails to address two practical issues: the overhead of mapping can be great due to fragmentation arising from allocations and deallocations of reducers in the TLMM region, and performing a hypermerge of views in TLMM regions is complicated and may incur heavy mapping overhead. We discuss each of these issues in turn.

If views are allocated within a TLMM region, the runtime system needs to manage the storage in the region separately from its normal heap allocator. Since reducers may be allocated and deallocated throughout program execution, the TLMM region may become fragmented with live reducer hyperobjects scattered across the region. Consequently, when a worker maps in physical pages associated with a different worker's TLMM region, as must occur for a hypermerge, multiple physical pages may need to be mapped in, each requiring two kernel crossings (from user mode to kernel mode and back). Even though the remapping overhead can be amortized against steals, and the Cilk-M runtime already performs a sys_pmap call upon a successful steal to maintain the cactus stack, if the number of sys_pmap calls is too great, the kernel crossing overhead can become a scalability bottleneck, which might outweigh the benefit of replacing the hash-table lookups of the hypermap approach with virtual address translations.

The second issue involves the problem of performing hypermerges. Consider a hypermerge of the local views in two workers W_1 and W_2, and suppose that W_1 is performing the hypermerge. To perform a monoid operation on a given pair of views, both views must be mapped into the same address space. Consequently, at least one of the views cannot be mapped to its appropriate location in the TLMM region, and the code to reduce them with the monoid operation must take that into account. For example, if W_2's view contains a pointer, W_1 would need to determine whether the pointer was to another of W_2's views or to shared memory. If the former, it would need to perform an additional address translation. This "pointer swizzling" could be done when W_1 maps W_2's views into its address space, but it requires compiler support to determine which locations are pointers, as well as adding a level of complexity to the hypermerge process.

Since "any problem in computing can be solved by adding another level of indirection,"[4], the Cilk-M runtime employs ***thread-local indirection***. The idea is to use the TLMM region to store pointers to local views which themselves are kept in shared memory visible to all workers. When a reducer is allocated, a memory location is reserved in the TLMM region to hold a pointer to its local view. If no view has yet been created, the pointer is null. Accessing a reducer simply requires the worker to check whether the pointer is null, and if not, dereference it, which is done by the virtual-address translation provided by the hardware. In essence, the memory-mapping reducer mechanism replaces the use of hypermaps with the use of the TLMM region.

The two problems that plague the naive scheme are solved by thread-local indirection. The TLMM region contains a small, compact set of pointers, all of uniform size, precluding internal fragmentation and making storage management of reducers simple, avoiding pointer swizzling. The TLMM region requires only a simple scalable[5] memory allocator for single-word objects (the pointers). Since local views are stored in shared memory, the job of handling them is conveniently delegated to the ordinary heap allocator. Thread-local indirection also solves the problem of one worker gaining access to the views of another worker in order to perform hypermerge. Since the local views are allocated in shared memory, a worker performing the hypermerge can readily access the local views of a different worker. The only residual problems are one, how to manage the storage for the pointers in the TLMM region, and two, how to determine which local views to merge, which will be addressed in turn next.

Organization of worker-local views

The third question is how a worker's local views can be organized compactly. Recall that after a steal, the thief resuming the stolen frame starts with an empty set of views, and whenever the thief

[4] Quotation attributed to David Wheeler in [87].

[5] To be scalable, the memory allocator allocates a local pool per worker and occasionally rebalances the fixed-size slots among local pools when necessary in the manner of Hoard [10].

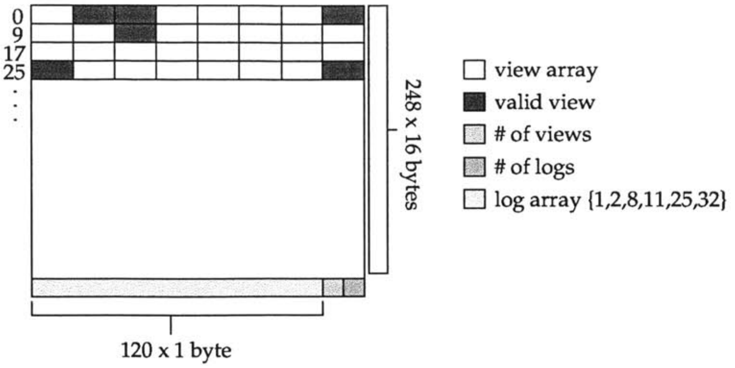


Figure 4-5: An example of a SPA map in which locations 1, 2, 8, 11, 25, and 32 in the view array are occupied.

accesses a reducer for the first time, a new identity view is created lazily. Once a local view has been created, subsequent accesses to the reducer return the local view. Moreover, during a hypermerge, a worker sequences through two sets of local views to perform the requisite monoid operations. Specifically, a worker's local views must be organized to allow the following operations:

- given (the address of) a reducer hyperobject, perform a constant-time ***lookup*** of the local view of the reducer; and
- ***sequence*** through all of a worker's local views during a hypermerge in linear time and ***reset*** the set of local views to the empty set.

To support these activities efficiently, the Cilk-M runtime system employs a "sparse accumulator (spa)" data structure [50] to organize a worker's local views. A traditional ***sparse accumulator (SPA)*** consists of two arrays:[6] an array of values, and an array containing an unordered "log" of the indices of the nonzero elements. The data structure is initialized to an array of zeros at start-up time. When an element is set to a nonzero value, its index is recorded in the log, incrementing the count of elements in the SPA (which also determines the location of the end of the log). Sequencing is accomplished in linear time by walking through the log and accessing each element in turn.

Cilk-M implements the SPA idea by arranging the pointers to local views in a ***SPA map*** within a worker's TLMM region. A SPA map is allocated on a per-page basis, using 4096-byte pages on x86 64-bit architectures. Each SPA map contains the following fields:

- a ***view array*** of 248 elements, where each element is a pair of 8-byte pointers to a local view and its monoid,
- a ***log array*** of 120 bytes containing 1-byte indices of the valid elements in the view array,
- the 4-byte number of valid elements in the view array, and
- the 4-byte number of logs in the log array.

Figure 4-5 illustrates an example of a SPA map.

Cilk-M maintains the invariant that empty elements in the view array are represented with a pair of null pointers. Whenever a new reducer is allocated, a 16-byte slot in the view array is allocated, storing pointers to the executing worker's local view and to the monoid. When a reducer goes out of scope and is destroyed, the 16-byte slot is recycled. The simple memory allocation for the

[6] For some applications, a third array is used to indicate which array elements are valid, but for some applications, invalidity can be indicated by a special value in the value array.

TLMM region described earlier keeps track of whether a slot is assigned or not. Since a SPA map is allocated in a worker's TLMM region, the virtual address of an assigned 16-byte slot represents the same reducer for every worker throughout the life span of the reducer and is stored as a member field `tlmm_addr` in the reducer object.

A reducer lookup can be performed in constant time, requiring only two memory accesses and a predictable branch. A lookup entails accessing `tlmm_addr` in the reducer (first memory access), dereferencing `tlmm_addr` to get the pointer to a worker's local view (second memory access), and checking whether the pointer is valid (predictable branch). The common case is that the `tlmm_addr` contains a valid local view, since a lookup on an empty view occurs only once per reducer per steal. As we shall see when discussing view transferal, however, a worker resets its SPA map by filling it with zeros between successful steals. If the worker does not have a valid view for the corresponding reducer, the `tlmm_addr` simply contains zeros.

Sequencing through the views can be performed in linear time. Since a worker knows exactly where a log array within a page starts and how many logs are in the log array, it can efficiently sequence through valid elements in the view array according to the indices stored in the log array. The Cilk-M runtime stores pointers to a local view and the reducer monoid side-by-side in the view array, thereby allowing easy access to the monoid interface during the hypermerge process. In designing the SPA map for Cilk-M, a 2 : 1 size ratio between the view array and the log array is explicitly chosen. Once the number of logs exceed the length of the log array, the Cilk-M runtime stops keeping track of logs. The rationale is that if the number of logs in a SPA map exceeds the length of its log array, the cost of sequencing through the entire view array, rather than just the valid entries, can be amortized against the cost of inserting views into the SPA map.

View transferal

The fourth question is how a worker can efficiently gain access to another worker's local views and perform view transferal efficiently. The Cilk-M runtime system, which employs thread-local indirection and SPA maps, also includes an efficient view-transferal protocol that does not require extra memory mapping.

In the hypermap approach, view transferal simply involves switching a few pointers. Suppose that worker W_1 is executing a full frame F_1 that is returning. It simply deposits its local views into another frame F_2 executing on worker W_2 that is either F_1's left sibling or parent, at the appropriate hypermap placeholder. In the memory-mapping approach, more steps are involved. In particular, even though all local views are allocated in the shared region, their addresses are only known to W_1, the worker who allocated them. Thus, W_1 must ***publish*** pointers to its local views, making them available in a shared region.

There are two straightforward strategies for W_1 to publish its local views. The first is the ***mapping strategy***: W_1 leaves a set of page descriptors corresponding to the SPA maps in its TLMM region in F_2, which W_2 later must map in its TLMM region to perform the hypermerge. The second strategy is the ***copying strategy***: W_1 simply copies those pointers from its TLMM region into a shared region. Cilk-M employs the copying strategy because generally the number of reducers used in a program is small, and thus the overhead of memory mapping greatly outweighs the cost of copying a few pointers.

For W_1 to publish its local views, which are stored in the ***private SPA maps*** in its TLMM regions, W_1 simply allocates the same number of ***public SPA maps*** in the shared region, and ***transfers*** views from the private SPA maps to the public ones. As W_1 sequences through valid indices in a view array to copy from a private SPA map to a public one, it simultaneously zeros out those valid indices in the private SPA map. All transfers are complete, the public SPA maps contain all the references to W_1's local views, and the private SPA maps are all empty (the view array contains all zeros). Zeroing out W_1's private SPA maps is important, since W_1 must engage in work-stealing next, and the empty private SPA maps ensure that the stolen frame is resumed with an empty set of local views.

Since a worker must maintain space for public SPA maps throughout its execution, Cilk-M explicitly configures SPA maps to be compact and allocated on the per-page basis. Each SPA map holds up to 248 views, making it unlikely that many SPA maps are ever needed. As mentioned earlier, the Cilk-M runtime system maintains the invariant that an entry in a view array contains either a pair of valid pointers or a pair of null pointers indicating that the entry is empty. Thus, a newly allocated (recycled) SPA map is empty.[7] The fact that a SPA map is allocated on the per-page basis allows the Cilk-M runtime to easily recycle empty SPA maps by maintaining memory pools[8] of empty pages solely for allocating SPA maps.

In the memory-mapping approach, a frame contains placeholders to SPA maps instead of to hypermaps, so that W_1 in our scenario can deposit the populated public SPA maps into F_2 without interrupting W_2. Similarly, a hypermerge involves two sets of SPA maps instead of hypermaps. When W_2 is ready to perform the hypermerge, it always sequences through the map that contains fewer views and reduces them with the monoid operation into the map that contains more views. After the hypermerge, one set of SPA maps contain the reduced views, whereas the other set (assuming they are public) are all empty and can be recycled. Similar to the transfer operation, when W_2 performs the hypermerge, as it sequences through the set with fewer views, it zeros out the valid views, thereby maintaining the invariant that only empty SPA maps are recycled.

View transferal in the memory-mapping approach incurs higher overhead than that in the hypermap approach, but this overhead can be amortized against steals, since view transferals are necessary only if a steal occurs. As Section 4.4 shows, even with the overhead from view transferal, the memory-mapping approach performs better than the hypermap approach and incurs less total overhead during parallel execution.

Support for parallel REDUCE *operations*

The reducer mechanism in Cilk-M supports parallelism in REDUCE operations. That means, the Cilk-M runtime must set up the invocation to a REDUCE in a way which allows the REDUCE to be stolen. Furthermore, once all necessary hypermerges complete, the executing worker must resume the user code at the appropriate execution point.

To allow a REDUCE operation to be stolen, the executing worker must perform hypermerges on its TLMM stack. In Cilk-M, every worker juggles between two stacks, its TLMM stack allocated in the TLMM region for executing user code, and its runtime stack for executing runtime code. The runtime stack is necessary — recall from Section 3.2, a worker always remaps its TLMM stack upon a successful steal, and the worker must use an alternative stack during the remapping. In two out of three scenarios where hypermerges may occur, i.e., returning from a **cilk_spawn** or performing a successful joining steal, the executing worker is operating on its runtime stack. In such scenarios, the worker must switch from its runtime stack and execute the hypermerge on its TLMM stack so as to allow a REDUCE operation to be stolen.

To ensure that a worker completing a hypermerge resumes the user code at the appropriate execution point is more complicated. Since a REDUCE operation may contain parallelism, the worker who finishes the hypermerge may differ from the worker who initiated the hypermerge.

[7] To be precise, only the number of logs and the view array must contain zeros.

[8] The pools for allocating SPA maps are structured like the rest of pools for the internal memory allocator managed by the runtime. Every worker has its own local pool, and a global pool is used to rebalance the memory distribution between local pools in the manner of Hoard [10].

Thus, when a worker initiates a hypermerge, it must set up its TLMM stack and runtime data structure for bookkeeping (i.e., its deque) to correctly correspond to each other and to allow the hypermerge, once complete, to naturally resume the right execution point in the user code.

Let's examine the three scenarios when hypermerges may occur one by one. The first scenario is when a full frame F_1 returns from a **cilk_spawn**. In this case, the executing worker performs hypermerges and a view transferal as part of F_1's return protocol. If F_1 happens to be the last child returning, the worker also performs a joining steal as part of the return protocol. If the joining steal is successful (which may trigger more hypermerges), the executing worker should resume the parent, say F_2, passing the **cilk_sync** at which F_2 was stalled. Thus in this scenario, the worker who initiated the hypermerges must ensure that once all hypermerges are complete, whichever worker finishes the last hypermerge must execute the return protocol again. The Cilk-M runtime achieves this by setting up the worker's TLMM stack and deque as if the parent frame F_2 *spawned* the function M that performs the hypermerges after the child F_1 returns. Note that a possible alternative is to set up the worker's TLMM stack and deque as if F_1 *called* the function M. This strategy may work but seems to be messier, because now F_1 would need to return twice (the second time after M finishes), and the second return must somehow trigger the return protocol. Furthermore, unlike the current strategy which provides a natural resumption point in the user code for the worker after M completes (i.e., after **cilk_sync** in the parent function if the joining steal is successful), this strategy does not. The second scenario is when a full frame F executes a **cilk_sync** successfully. In this case, the executing worker is already on its TLMM stack. Thus, it seems natural to simply allow the **cilk_sync** function to perform the hypermerge, which may contain parallelism if the REDUCE operations triggered by the hypermerge contain parallelism. The fact that Cilk-M supports SP-reciprocity makes this strategy feasible. Finally, the last scenario is during a successful joining steal, which is the combination of the first two scenarios. Assuming the first two scenarios are handled correctly, this scenario should just work.

In all scenarios, the runtime must set up the hypermerge process so that no worker-specific data is captured on the TLMM stack across **cilk_spawn** and **cilk_sync**, because a worker who resumes the hypermerge after a call to REDUCE may differ from the worker who initiated the call. That means, a worker must also perform a view transferal on its own set of local views before a hypermerge, and perform the hypermerge between the two public SPA maps stored in the heap.

Finally, since a worker executing hypermerges should not be holding any locks that belong to part of the runtime system bookkeeping, Cilk-M employs a locking protocol similar to the modified locking protocol due to Leiserson and Schardl [96] as discussed in Section 4.2.

This implementation of the reducer mechanism treats a REDUCE operation like a piece of user code that may spawn. Therefore, a parallel REDUCE operation can employ yet another reducer, and the reducer will work as expected. As we shall see in Chapter 5, the reducer array library employs a parallel REDUCE operation that uses another reducer.

4.4 An Empirical Evaluation of Memory-Mapped Reducers

This section compares the memory-mapping approach used by Cilk-M to implement reducers to the hypermap approach used by Cilk Plus. The evaluation quantifies the overheads of the two systems incurred during serial and parallel executions on three simple synthetic microbenchmarks and one application benchmark. Experimental results show that memory-mapped reducers not only admit more efficient lookups than hypermap reducers, they also incur less overhead overall during parallel executions, despite the additional costs of view transferal.

Recall from Section 4.3, in order to allow parallelism in a REDUCE operation, the Cilk-M run-

Name	*Description*
`add-n`	Summing 1 to x into n add-reducers in parallel
`min-n`	Processing x random values in parallel to find the min and accumulate the results in n min-reducers
`max-n`	Processing x random values in parallel to find the max and accumulate the results in n max-reducers

Figure 4-6: The three microbenchmarks for evaluating lookup operations. For each microbenchmark, the value x is chosen according to the value of n so that roughly the same number of lookup operations are performed.

time system must perform additional work as part of the hypermerge process, such as setting up a worker's TLMM stack and deque before invoking a REDUCE operation. This section as well evaluates the overhead for supporting parallel REDUCE operations, but will not evaluate the performance of a reducer with a parallel REDUCE operation. Since the reducer array library employs a parallel REDUCE operation, we shall delay the evaluation of a reducer with a parallel REDUCE operation until next chapter. While none of the benchmarks shown in this section employ parallel REDUCE operations, the current implementation always performs steps necessary to support parallel REDUCE operations. One could design a reducer interface to provide the runtime system information on its REDUCE operation, thereby allowing the runtime to skip these steps if all reducers involved in a hypermerge employ serial REDUCE. Nonetheless, experimental results show that these steps incur negligible overhead; the execution times of the same program that uses serial REDUCE with and without these steps are comparable.

General setup. The evaluation compares the two approaches using a few microbenchmarks using reducers included in the Cilk Plus reducer library and one application benchmark. Figure 4-6 shows the list of microbenchmarks and their descriptions. All microbenchmarks are synthetic, designed to perform lookup operations repeatedly with simple REDUCER operations that perform addition, finding the minimum, and finding the maximum. The value n in the name of the microbenchmark dictates the number of reducers used, determined at compile-time. The value x is an input parameter chosen so that a given microbenchmark with different n performs roughly the same number of lookup operations. The application benchmark is a parallel breath-first search program [96] called PBFS.

The application benchmark used is the parallel breadth-first search algorithm (referred to as PBFS) due to Leiserson and Schardl [96]. I obtained the code base for PBFS from the authors and made a few small modifications to fix minor bugs and improve the performance. Specifically, I modified the application to use the scalable memory allocator library released as part of TBB [126] instead of the default memory allocator. In addition, I manually performed a ***lookup optimization*** — lifting reducers' lookup operations out of serial **for** loops — so that a given loop body accesses a reducer's underlying view directly instead of accessing the reducer, which causes a lookup operation to be performed. Since all lookup operations within a single strand (in this case, across **for** loop iterations) return the same view, one lookup operation before entering the **for** loop to obtain the view suffices.

All benchmarks were compiled using the Cilk Plus compiler version 12.0.0 with `-O2` optimization. The experiments were run on an AMD Opteron system with 4 quad-core 2 GHz CPU's having a total of 8 GBytes of memory. Each core on a chip has a 64-KByte private L1-data-cache, a 512-KByte private L2-cache, and a 2-MByte shared L3-cache.

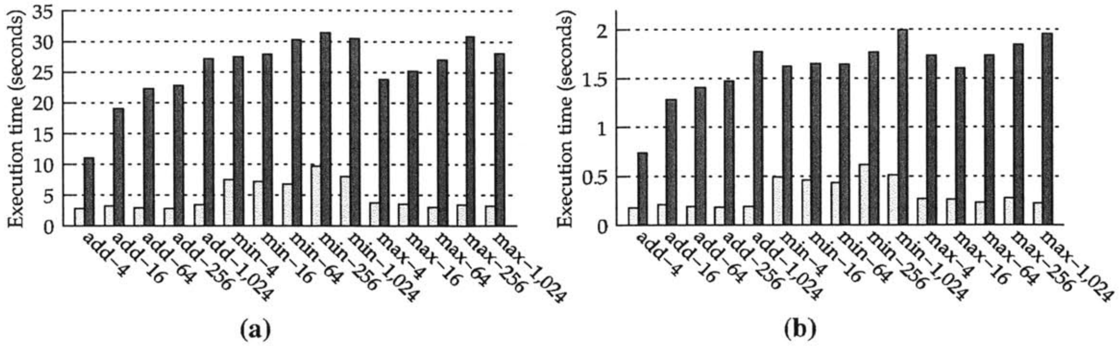


Figure 4-7: Execution times for microbenchmarks with varying numbers of reducers using Cilk-M and Cilk Plus, running on **(a)** a single processor and **(b)** on 16 processors, respectively.

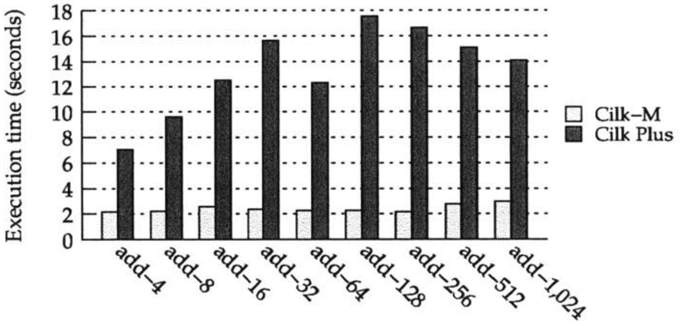


Figure 4-8: Reducer lookup overhead of Cilk-M and Cilk Plus running the microbenchmark using add reducers on a single processor. A single cluster in the x-axis shows the overheads for both systems for a given n, and the y-axis shows the overheads in execution time in seconds.

Performance overview using microbenchmarks

Figure 4-7 shows the microbenchmark execution times for a set of tests with varying number of reducers running on the two systems. Figure 4-7(a) shows the execution times running on a single processor, whereas Figure 4-7(b) shows them for 16 processors. Each data point is the average of 10 runs with standard deviation less than 5%. Across all microbenchmarks, the memory-mapped reducers in Cilk-M consistently outperform the hypermap reducers in Cilk Plus, executing about 4–9 times faster for serial executions, and 3–9 times faster for parallel executions.

Lookup overhead. Figure 4-8 presents the lookup overheads of Cilk-M 1.0 and Cilk Plus on `add-n` with varying n. The overhead data was obtained as follows. First, I ran the `add-n` with x iterations on a single processor. Then, I ran a similar program called `add-base-n`, which replaces the accesses to reducers with accesses to a simple array, also running x iterations. Since hypermerges and reduce operations do not take place when executing on a single processor, `add-base-n` essentially performs the same operations as `add-n` minus the lookup operations. Figure 4-8 shows the difference in the execution times of `add-n` and `add-base-n` with varying n. Each data point takes the average of 10 runs with standard deviation less than 2% for Cilk-M and less than 12% for Cilk Plus.

While the lookup overhead in Cilk-M stays fairly constant as n varies, the lookup overhead

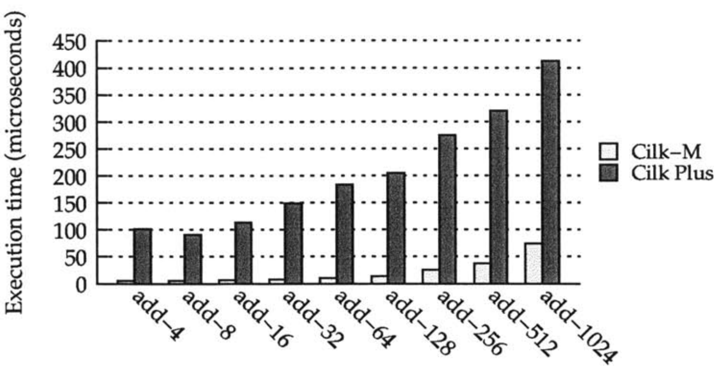


Figure 4-9: Comparison of the reduce overheads of Cilk-M and Cilk Plus running add-n on 16 processors. A single cluster in the *x*-axis shows the overheads for both system for a given *n*, and the *y*-axis shows the reduce overheads in milliseconds.

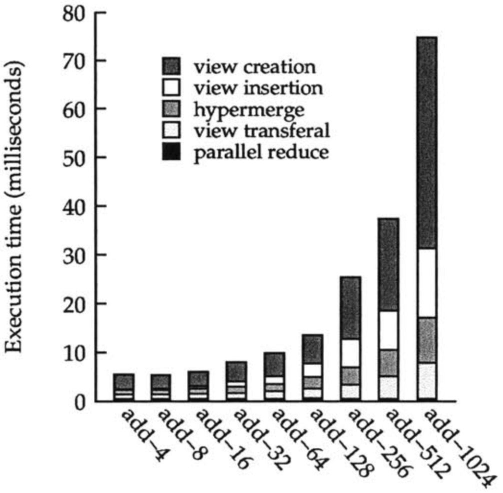


Figure 4-10: The breakdown of the reduce overhead in Cilk-M for add-n on 16 processors with varying *n*.

in Cilk Plus varies quit a bit. This makes sense, since a lookup operation in Cilk-M translates into two memory accesses and a branch disregarding what *n* is, whereas a lookup operation in Cilk Plus translates into a hash-table lookup whose time depends on how many items the hashed bucket happens to contain, as well as whether it triggers a hash-table expansion. Even though the implementation of Cilk Plus rehashes the hash table from time to time to keep the items in a bucket roughly constant, the lookup overhead still visibly varies.

Reduce overhead during parallel execution

Besides the lookup overhead, this section also studies the other overheads incurred by the use of reducers during parallel executions. We refer to the overheads incurred only during parallel executions as the ***reduce overhead***, which includes overheads in performing hypermerges, creating views, and inserting views into a hypermap in Cilk Plus or a SPA map in Cilk-M. For Cilk-M, this overhead also includes view transferal. For both systems, additional lookups are performed during a hypermerge, and they are considered as part of the overhead as well.

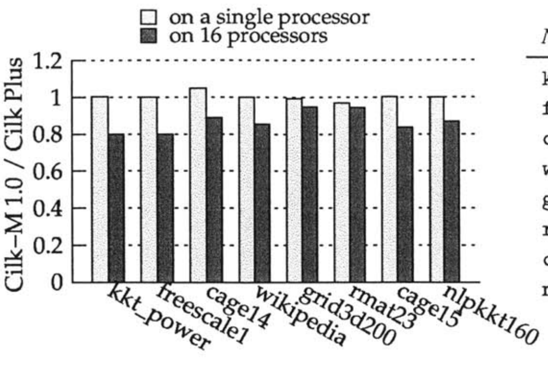
Figure 4-9 compares the reduce overhead of the two systems. The data was collected by running `add-rdcers-n` with varying n on 16 processors for both systems and instrumenting the various sources of reduce overhead directly inside the runtime system code. In order to instrument the Cilk Plus runtime, I obtained the open-source version of the Cilk Plus runtime, which was released with ports of the Cilk Plus language extensions to the C and C++ front-ends of gcc-4.7 [1]. I downloaded only the source code for the runtime system (revision 181962), inserted instrumentation code, and made it a plug-in replacement for the Cilk Plus runtime released with the official Intel Cilk Plus compiler version 12.0.0. This open-source runtime system is a complete runtime source to support the Linux operating system [1], and its performance seems comparable to the runtime released with the compiler. Given the high variation in the reduce overhead when memory latency plays a role, the data represents the average of 100 runs. Since the reduce overhead is correlated with the number of (successful) steals, I also verified that in these runs, the average numbers of steals for the two systems are comparable.

As can be seen in Figure 4-9, the reducer overhead in Cilk Plus is much higher than that in Cilk-M, and the discrepancy increases as n increases. It makes sense that the overhead increases as n increases, because a higher n means more views are created, inserted, and must be reduced during hypermerges. Nevertheless, the overhead in Cilk Plus grows much faster than that in Cilk-M. It turns out that the Cilk Plus runtime spends much more time on view insertions (inserting newly created identity views into a hypermap), which dominates the reduce overhead, especially as n increases, resulting a much higher reduce overhead, even though the Cilk-M runtime has the additional overhead of view transferal. In contrast, Cilk-M spends much less time on view insertions than Cilk Plus, which makes sense. A view insertion in Cilk-M involves writing to one memory location in a worker's TLMM region, whereas in Cilk Plus, it involves inserting into a hash table. Moreover, a SPA map in Cilk-M store views much more compactly than does a hypermap, which helps in terms of locality during a hypermerge.

For Cilk-M, I was interested in studying the breakdown of the reduce overhead, as shown in Figure 4-10, which attributes the overhead to five activities: view creation, view insertion, view transferal, hypermerge, which includes the time to execute the monoid operation, and setup necessary for parallel REDUCE operations. As can be seen from the breakdown, overhead from view transferal grows rather slowly as n increases, demonstrating that the SPA map allows efficient sequencing. Furthermore, the dominating overhead turns out to be view creations, which inspires confidence in the various design choices made in the memory-mapping approach. The overhead in supporting parallel REDUCE operations is an orthogonal issue, although the overhead is negligible compared to all other overheads. This result is consistent with the fact that in all experiments I ran with microbenchmarks, the executions times with support for parallel REDUCE operations enabled are comparable to that when the support is disabled.

Performance evaluation using PBFS

Lastly, this section presents the evaluation using a real-world application, the parallel breath-first search (PBFS) due to Leiserson and Schardl [96]. In PBFS, given an input graph $G(V,E)$ and a starting node $v_0 \in V$, the algorithm finds the shortest distance between v_0 and every other node in V. The algorithm explores the graph "layer-by-layer", where the d-th layer is defined to contain the set of nodes in V that are d-distance away from v_0. While the algorithm explores the d-th layer, it discovers nodes in the $d+1$-th layer. The set of nodes in a layer is kept in a data structure referred as a ***bag***, which is a container of an unordered set that allows efficient insert, merge, and split. The algorithm alternates between two bags to insert throughout execution — as it explores nodes stored in one bag that belongs to the same layer, it inserts newly discovered nodes that belongs to the next



Name	$\|V\|$	$\|E\|$	*D*	*# of lookups*
kkt_power	2.05M	12.76M	31	1027
freescale1	3.43M	17.1M	128	1748
cage14	1.51M	27.1M	43	766
wikipedia	2.4M	41.9M	460	1631
grid3d200	8M	55.8M	598	4323
rmat23	2.3M	77.9M	8	71269
cage15	5.15M	99.2M	50	2547
nlpkkt160	8.35M	225.4M	163	4174

(a) **(b)**

Figure 4-11: **(a)** The relative execution time of Cilk-M to that of Cilk Plus running PBFS on a single processor and on 16 processors. Each value is calculated by normalizing the execution time of the application on Cilk-M with the execution time on Cilk Plus. **(b)** The characteristics of the input graphs for parallel breath-first search. The vertex and edge counts listed correspond to the number of vertices and edges.

layer into another bag. Since the algorithm explores all nodes within a given layer in parallel, the bags are declared to be reducers to allow parallel insertion.

Figure 4-11(a) shows the relative execution time between Cilk-M and Cilk Plus on a single processor and on 16 processors. Since the work and span of a PBFS computation depend on the input graph, we evaluated the relative performance with 8 input graphs whose characteristics are shown in Figure 4-11(b). These input graphs are the same ones used in [96] to evaluate the algorithm. For each data point, I measured the mean of 10 runs, which has a standard deviation of less than 1%. Figure 4-11 shows the mean for Cilk-M normalized by the mean for Cilk Plus.

For single-processor executions, the two systems perform comparably, with Cilk-M being slightly slower. Since the number of lookups in PBFS is extremely small relative to the input size, the lookups constitute a tiny fraction of the overall work (measured by the size of the input graph). Thus, it's not surprising that the two systems perform comparably for serial executions. On the other hand, Cilk-M performs noticeably better during parallel executions, which is consistent with the results from the microbenchmarks. Since the reduce overhead in Cilk-M is much smaller than that in Cilk Plus, PBFS scales better.

4.5 Conclusion

This chapter lays out a different way of implementing reducer hyperobjects, namely, using the memory mapping approach. As we have seen in Section 4.4, experimental results show that the memory-mapping approach admits a more efficient implementation, demonstrating the utility of the TLMM mechanism for building memory abstractions.

There is one particular downside about the memory-mapping approach, however, which is that a view transferal incurs overhead proportional to the number of active reducers in the computation. This particular overhead is inherent to the memory-mapping reducer mechanism in that a worker's local views (at least pointers to them) are stored within a region private to the worker and therefore is difficult to avoid.

Nevertheless, in most applications, the number of active reducers tends to be small given that each reducer represents a nonlocal variable shared among workers. Furthermore, whether the reducer mechanism constitutes a useful memory abstraction when a large number of reducers are used is still an open question. Recall that during parallel execution, the use of reducers generates a nondeterministic amount of additional work. If a large number of reducers are used, this additional work may become a scalability bottleneck. We explore this topic further in the next chapter.

Chapter 5

Library Support for Reducer Arrays

A natural extension for reducers is to allow array types. Thus far, we have focused our attention on scalar reducers, where a reducer represents a scalar type object and the REDUCE operation for combining two views takes constant time. If one wishes to parallelize an application that employs a nonlocal variable that is an array type, there are two possible approaches one may take in order to allow sharing without introducing determinacy races. The first approach is to declare an ***array of reducers***, which allows one to employ existing reducer library support for scalar reducer types. The second approach is to write library support for an ***array reducer***, that supports a reducer whose underlying view is an array. Either approach has its pros and cons. This chapter explores a third approach, referred to as a ***reducer array***, which attempts to combine the advantages of the first two approaches. Since the use of reducers generates a nondeterministic amount of additional overhead for creating, managing, and combining views during parallel execution, if the REDUCE operation takes nonconstant time or a large number of reducers are used, this additional overhead may have an impact on performance. This chapter also studies the theoretical framework for analyzing computations that employ reducers due to Leiserson and Schardl [96] and extends the analysis to better understand the overhead of using reducer arrays.

We shall first examine the difference between these approaches before we dive into the implementation of a library for reducer arrays. The first two approaches, an array of reducers versus an array reducer, have some fundamental differences in terms of their semantics, whereas the third approach, a reducer array, is somewhat of a hybrid between the first two. Figure 5-1 summarizes their differences. The first approach, array of reducers, associates each array element with its own reducer whose REDUCE operation combines two elements together, thereby allowing each array element to have its own view. We shall refer to this view representation as the ***element view***. With the element view representation, views are created only for elements accessed during parallel execution, and a hypermerge process operates only on elements accessed. The second approach, an array reducer, associates the entire array with one reducer and employs an ordinary array as its underlying view, which we shall refer as the ***array view*** representation. With the array view representation, whenever a new view is created during parallel execution, the view created represents the entire array, and its REDUCE operation must reduce two array views. As a result, a hypermerge process invoking this REDUCE operation simply operates on every element in the array. Finally, the third approach investigated in this chapter, a reducer array, combines features from the first two approaches. A reducer array, like the array reducer approach, associates the entire array with a single reducer, but it employs the SPA data structure [50] (described in Section 4.3) as its underlying view, which we shall refer as the ***SPA view***. With the SPA view representation, a single view still represents the entire array, but a hypermerge process combining two SPA views needs to operate only on elements

Approach	*View representation*	REDUCE *operation*	*Hypermerge process*
1. Array of reducers	element view	reduce two elements	operate on elements accessed
2. Array reducer	array view	reduce two arrays	operate on every element
3. Reducer array	SPA view	reduce two arrays	operate on elements accessed

Figure 5-1: Summary of semantic differences between the three approaches.

accessed since the views were created.

Each of the first two approaches has its respective pros and cons. In terms of the view representation, the array view in the array reducer approach has a couple of advantages over the element view in the array of reducers approach. First, the array view likely leads to a better utilization of space. The array view employs a single reducer for the entire array, whereas the element view employs an array of reducers. Even though the element view representation causes only views for accessed elements to be created, whereas the array view requires a newly created view to allocate space for the entire array, the array of reducers required by the element view takes up much more space during parallel execution, for the following reasons. A reducer typically is larger in size and has a longer life span than its corresponding views. As the implementation currently stands, a reducer contains 96 bytes of bookkeeping data in addition to its leftmost view. Moreover, a reducer requires space for both the private SPA maps in the TLMM reducer region and the public SPA maps allocated during hypermerges. Every reducer hyperobject alive (and accessed) takes up 16 bytes of space in a worker's TLMM reducer region and another 16 bytes in every public SPA map created for a hypermerge throughout its lifetime. Assuming the original nonlocal array contains elements of some primitive type or pointers to objects, the array view will consume less space than the element view throughout execution.

Perhaps more importantly, the array view has a second advantage over the element view in that its natural array structure allows an important optimization which cannot be done with the element view. Since the array view representation allocates elements for an array in contiguous memory locations, only one reducer lookup operation suffices for all the corresponding array reducer accesses within a single strand. Henceforth, we shall refer to this optimization as the ***lookup optimization***. By contrast, since view allocation occurs when an element is first accessed, the element view representation tends to allocate views for elements in a given array in noncontiguous memory locations. Consequently, every access to a given element must translate into a reducer lookup operation, even when multiple elements are accessed within a single strand. Moreover, the array view's natural array structure may also lead to better locality during a hypermerge.

Nevertheless, the array of reducers approach has an advantage over the array reducer approach, which is that it operates only on elements accessed during a hypermerge. With an array of reducers, an element view is initialized only upon access. Consequently, only views corresponding to elements accessed need to be reduced during a hypermerge. By contrast, upon the first access, an array reducer creates and initializes the entire array view, and a hypermerge access invoking its REDUCE operation touches every element in the array. This advantage is especially pronounced when the nonlocal array is sparsely accessed.

The third approach, the library implementation of reducer arrays described in this chapter, attempts to combine the most advantageous features of the first two approaches. In particular, a reducer array employs the SPA view, which associates the entire array with one reducer and allocates elements for a given array in contiguous memory locations, thereby allowing the lookup optimization and obtaining better locality during hypermerges. The SPA view costs more space compared to the array view; additional space is needed for bookkeeping sake. Using the SPA view, however, a reducer array is able to initialize an element with the identity value only upon its first access and therefore operates only on elements that have been accessed during a hypermerge. To mitigate the additional space overhead inherent in the SPA view, the library also employs per-worker memory pools to recycle views. Experimental results show that a computation that uses a reducer array consumes less space and performs $2\times$ faster or more than that with an array of reducers during parallel execution (the exact performance difference depends on the array size and the access density of the array).

Although reducer arrays are faster than arrays of reducers, the inherent overhead incurred by the use of reducers is significant for large arrays. Depending on the specific computation and the size of the reducer array used in the computation, this overhead can become a scalability bottleneck. This brings us to the question, what kind of applications may benefit from reducer arrays, or more pointedly, do reducer arrays constitute a useful memory abstraction? We will examine these questions by studying the overhead in using reducer arrays, extending the theoretical framework for analyzing computations that employ reducers due to Leiserson and Schardl [96]. The analysis gives an upper bound on the execution time and provides us with some insights as to what kind of scalability we may expect out of a computation that uses a reducer array. As a case study, the parallel breadth-first search due to Leiserson and Schardl [96] is augmented with parent computations, which uses a reducer array of size n, where n is the number of vertices in the input graph. This chapter also analyzes the theoretical bound of this application and evaluates its scalability empirically. The analysis tells us that we should not expect the application to scale, and indeed we see little scalability empirically — the speedup plateaus around 12 processors, achieving 2–$3\times$ speedup depending on the input graph. An application can benefit from a reducer array if the application contains enough work besides accessing the reducer array such that the work dominates the additional overhead incurred by the use of reducers. Currently, I don't know of an application that exhibits such characteristics, however, and whether a reducer array constitutes a useful memory abstraction remains an open question.

The rest of this chapter is organized as follows. Section 5.1 describes the library support for reducer arrays. Section 5.2 studies the theoretical overhead of a computation that employs a reducer array. Section 5.3 evaluates the empirical performance of reducer arrays, comparing them to arrays of reducers and examining one case study using parallel breadth-first search with parent computations. Finally, Section 5.4 gives concluding remarks.

5.1 Library Support for Reducer Arrays

This section describes an implementation of library support for reducer arrays. Unlike an array of reducers, a reducer array uses one reducer to represent the entire array. Thus, during parallel execution, whenever a local view is created, the view represents the entire array in its full length n, where n being the length of the original nonlocal array. Unlike an array reducer, however, a reducer array initializes elements to its identity value lazily and minimizes the overhead during hypermerges, reducing only elements that have been accessed. This section provides the implementation details of the reducer array library.

The reducer pointer library

Before I present the implementation of the reducer array library, I digress for a moment to describe a new reducer interface, referred to as the ***reducer pointer*** interface. Chapter 4 presents the memory-mapped reducers assuming the reducer interface as originally documented in [48] and implemented

```
1  bool has_property(Node *n);
2  std::list<Node *> l;
3  reducer_ptr<reducer_list_append<Node *>::Monoid> lptr(&l);
4  // ...
5  void traverse(Node *n) {
6     if(n) {
7        if(has_property(n)) {
8           lptr->push_back(n);
9        }
10       cilk_spawn traverse(n->left);
11       traverse(n->right);
12       cilk_sync;
13    }
14 }
```

Figure 5-2: The same code as shown in Figure 4-3 which uses the new reducer pointer interface.

in Cilk++ [94]. The implementation in Cilk Plus closely resembles that in Cilk++, although the linguistic interface has since evolved — Pablo Halpern, one of the original designers of reducers and a Cilk Plus developer, investigated in a new interface, referred to as the ***reducer pointer*** interface. Even though the reducer pointer interface is not officially released by the Cilk Plus compiler,[1] this chapter studies and evaluates the reducer array assuming the reducer pointer interface, because the reducer pointer interface makes more sense in the context of a reducer array, as I explain shortly.

Recall the tree traversal example studied in Section 4.1, where the code traverses a binary tree and creates a list of all nodes that satisfy some given property in a *pre-order* fashion. We have seen a correct parallelization of the code in Figure 4-3 using the reducer interface. Figure 5-2 shows the same code parallelized the say way but uses the new reducer pointer interface.

Using the new reducer pointer interface, one turns the list `l` into a reducer by declaring a reducer pointer to manage the list `l` such as shown in line 3. We say that the list `l` is ***hyperized***, referring to the fact that now the list is managed by a reducer pointer. Then, instead of updating `l` directly, the code updates `l` via the reducer pointer interface in line 8, since the list `l` may be updated in parallel.

Hyperizing a nonlocal variable using the reducer pointer interface provides the same guarantees as employing a reducer in place of the nonlocal variable. Just like the reducer interface, a reducer pointer implements the monoid interface and provides the two important operations that the runtime invokes: IDENTITY and REDUCE.

The main distinction between the two interfaces is that, whether the underlying view is exposed. The reducer pointer interface is designed so that the underlying view is exposed, and the reducer pointer simply serves as a wrapper for coordinating parallel updates to the reducer. Note that using the reducer pointer interface, the user explicitly declares the leftmost view for the reducer and creates a reducer pointer to wrap around the leftmost view. Exposing the underlying view can be beneficial for performance reasons. For instance, if the user code knows that a view is updated repeatedly within a single strand, it can obtain the underlying view once for the entire strand and update the view directly instead of going through the reducer interface for the updates, which can be slower. This "optimization" must be exercised with extreme caution, however, since if the programmer is not careful, she may write code that races with the runtime system on the underlying view.

Using the reducer interface, on the other hand, the library implementer may choose to never expose the underlying view. Doing so results in a cleaner semantics, which comes with a cost — a reducer object must define update functions to allow the user code to indirectly perform updates on

[1] Thanks to Pablo who graciously provided me the implementation of reducer pointers so that I could experiment with the new interface before it is officially released.

```
int sum[100];
for(int i=0; i < 100; i++) {
    sum[i] = 0;
}
reducer_array<reducer_opadd<int>::Monoid> rArray(100, sum);
```

Figure 5-3: A declaration of reducer array that hyperizes a nonlocal array with length 50. The type of the reducer array is initialized by a monoid that performs addition with identity 0.

the underlying views, which results a more cumbersome syntax and a slower reducer update than what the reducer pointer interface would allow.

The reducer array library

Just like the other reducer libraries provided by Cilk Plus [69], the reducer array library is implemented as a C++ library without the compiler involvement. The interface of the reducer array library follows that of the reducer pointer library — to employ a reducer array, the user program hyperizes a nonlocal array by initializing a reducer array with the array length and the address of the nonlocal array.

In the case of a reducer array, the reducer pointer interface makes more sense than the original reducer interface for the following reason. Once the leftmost view becomes stable, i.e., being the only view remains reflecting all updates, the user code likely wishes to process the final data in some fashion. If the user code is only reading the array, it makes sense to read the array in parallel without generating additional views of the array. This is not possible with the reducer interface where the underlying view is not exposed. Thus, the reducer array library is implemented using the reducer pointer interface.[2]

The type of the reducer array is dictated by its type parameter, which specifies the monoid type for managing an element in the array. The monoid only specifies the IDENTITY and the REDUCE operations for an element in the array, and the library applies the monoid across elements in the array when appropriate. To clarify the terminology, henceforth whenever we refer to the IDENTITY and the REDUCE operations for a reducer array, we mean that the operations that are applied to the entire reducer array.

Figure 5-3 illustrates an example of hyperizing a nonlocal array of `int[]` type. Each element in the nonlocal array can be used to accumulate sums (for example, to compute a histogram), which is indicated by the type parameter that initializes the reducer array type, in this case by a monoid that performs addition with identity 0.

In general, one should not directly access the hyperized variable without going through the reducer pointer interface, unless the leftmost view is stable. This is particularly true in the case of a reducer array, because the hyperized nonlocal array does not constitute the leftmost view, but only a part of the leftmost view. Instead, the user code should either update the array either via the reducer array interface (which overloads the operator `[]` for accessing array elements), or obtain the underlying view returned by the reducer array and update the view.[3] We will come back to this point later when we explore the internals of the library implementation.

[2] Note that the point here is not the syntax used, but rather whether the library allows the underlying view to be exposed.

[3] Note that this does not preclude the optimization I mentioned earlier, in which the code accesses the view directly; it simply means that one should access the underlying view instead of the hyperized array.

Use of the SPA data structure

The underlying view of a reducer array is a SPA data structure [50]. Recall from Section 4.3 that a SPA data structure allows both random accesses to elements in an array and sequencing through the occupied array positions in constant time per element. In the reducer array library, a SPA view consists of an uninitialized ***value array*** of length n, where n is the length of the hyperized array, a ***log array*** of length $n/2$ which stores indices of elements accessed, and an array of length n for ***occupied flags*** which indicate the occupied position of the value array.

The use of a SPA view minimizes the overhead of the REDUCE operation for a reducer array. The reducer array library overloads the array subscript operator `[]` so that whenever an element is accessed, its corresponding occupied flag is set to TRUE and its index is logged. Thus, when two SPA views are reduced, only accessed elements are reduced. Like the use of the SPA in the Cilk-M runtime, once the number of accessed elements exceeds the length of the log array, the library stops logging the accessed indices and simply marks the occupied flags. At that point, however, enough work has been performed on the given view to justify sequencing through the entire array according to the occupied flags. Unlike the use of the SPA in the Cilk-M runtime system, however, the reducer array library must include an array of occupied flags to indicate which elements have been accessed. Since the type of the elements depends on the base type of the hyperized array, there is no general default value that can distinguish whether an element has been accessed of not. Each SPA view also keeps a counter to record the number of accessed elements for the given view. When two views are reduced together, the library always reduces the view with fewer accessed elements into the one with more, thereby reducing the number of elements that must be reduced or moved. The only exception is when one of the two views is the leftmost view, because the user program captures the reference to the underlying value array for the leftmost view.

Since the library depends on the SPA view to log every element accessed in order to correctly perform the IDENTITY and the REDUCE operations, it is critical that an element is accessed via the reducer library interface or the SPA view interface (the SPA view is a public type exported by the reducer array library and accessible to the user code). If the user program accesses an element by accessing the value array directly,[4] incorrect executions may result.

Recycling SPA views

Besides minimizing the REDUCE overhead, we would also like to minimize view creation overhead. To allow a SPA view to be created as efficiently as possible, the reducer array library implements a list of memory pools indexed by worker IDs to store SPA views. The memory pools are specific to a given instance of a reducer array. When a worker needs to create a SPA view for a given reducer array, it first checks in the local memory pool indexed by its ID. If the pool is empty, it allocates new memory for a new SPA view. Otherwise, a SPA view is retrieved from the pool. When two views are reduced together, one of the views is recycled and returned to the memory pools. To avoid memory drifting [10], each SPA view is marked with the worker ID which corresponds to the worker who created the view, so that a recycled view always gets returned to the worker who created the view initially.

The reducer array library maintains the invariant that a SPA view in a memory pool is not initialized except that all its occupied flags are set to FALSE. This invariant allows a worker to determine whether an element in the value array for a given view has been accessed and therefore initialize elements in the value array lazily. Whenever a new view is retrieved from the pool, the

[4] Although the value array is a private field of the SPA object, there are ways in which a user program can capture a reference to the value array.

executing worker does not initialize elements in the value array to identity. Rather, only upon first access of a given element does the executing worker initialize the element to identity.

Parallel REDUCE *operations*

When the array size is large, it is beneficial to allow the REDUCE operation for a reducer array to contain parallelism, enabling elements in two value arrays to be reduced in parallel. When combining two SPA views together in parallel, some care must be taken in order to combine the log arrays correctly. Let's walk through the REDUCE operation for a reducer array to make the explanation more concrete. Without loss of generality, let's assume that the REDUCE operation is reducing the right SPA view into the left SPA view, because the right SPA view has a smaller log (i.e., fewer elements have been accessed). We will also assume that logs from both views have not exceeded their respective length and that the resulting view must still keep track of the logs. Conceptually, the REDUCE operation walks the log array from the right SPA view, and for every index found in the log, the corresponding element in the right view's value array is reduced with or moved into the corresponding element in the left view's value array, depending on whether that particular element has been accessed in the left view. If the particular element has not been accessed, the index for this element must be inserted into the left view's log, and its corresponding occupied flag must be marked as TRUE. Since the REDUCE operation walks the right view's log array in parallel so as to reduce elements in parallel, we now have a determinacy race on the left view's log array.

To avoid the determinacy race on the log array, the reducer array library uses yet another reducer for the log array in the REDUCE operation for a reducer array. As mentioned in Section 4.3, Cilk-M's implementation of reducer mechanism treats a REDUCE operation as a piece of user code that may spawn, and so a parallel REDUCE operation can employ yet another reducer. In this case, hyperizing the log array avoids the determinacy race. Since we are hyperizing the log array and walking the log array in parallel, ideally the log array should support efficient split and merge, in addition to insert. The split operation allows the library to traverse the logs in parallel in a divide and conquer fashion. The merge operation allows the library to combine two logs together quickly. A vanilla implementation of an array does not support merge efficiently, however. Thus, instead of using a vanilla array, now the log is kept in a bag data structure (as described in [96] and summarized in Section 7.4) that supports efficient insert, split, and merge, which is ideal for our purpose.

Since keeping a log as a bag instead of a vanilla array and walking the logs in parallel incur additional overhead, performing the REDUCE operation in parallel is beneficial only if the array size is large enough. Thus, there are two implementations for the reducer array library. Henceforth, we will refer to the one without parallel REDUCE as the ***ordinary reducer array library***, and the one with parallel REDUCE as the ***parallel reducer array library***. As we will see in later sections, the parallel reducer array outperforms the ordinary reducer array empirically.

5.2 Analysis of Computations That Employ Reducer Arrays

As emphasized earlier, the use of reducers generates a nondeterministic amount of additional work. In the case of a reducer array, if the array size is large, the additional work may constitute a scalability bottleneck. How much additional work is generated? When does it become a bottleneck? The theoretical analysis presented in this section provides some insights. This section studies the theoretical framework due to Leiserson and Schardl [96] for analyzing a computation that uses a reducer with a nonconstant-time REDUCE operation and extends the framework to analyze a computation that uses a parallel reducer array with a parallel REDUCE operation. Leiserson and Schardl's

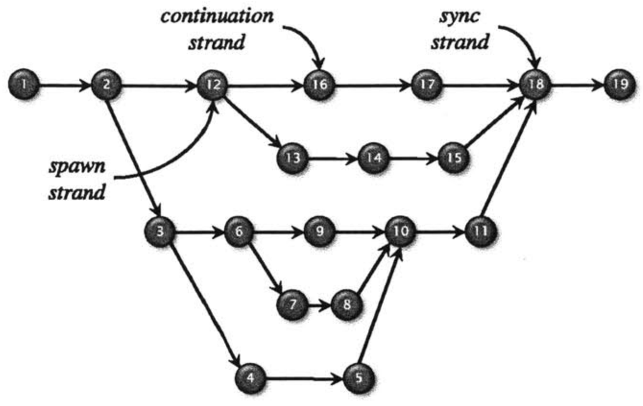


Figure 5-4: A dag representation of a multithreaded execution. The vertices represent strands, and edges represent dependencies between strands.

framework follows the framework of Blumofe and Leiserson [20] for analyzing a dynamically multithreaded computation using a work-stealing scheduler, which models a Cilk computation as a dag, and extends the analysis to handle the nondeterminism due to the use of a reducer. This section first reviews the dag model due to Blumofe and Leiserson [20], summarizes how Leiserson and Schardl [96] extend the analysis to handle reducers, and finally extends the model to analyze computations with parallel reducer arrays. Analysis presented in this section is joint work with Tao B. Schardl and Charles E. Leiserson. A portion of the text presented in this section is adapted from [96] with permission from the authors.

The dag model

The dag model for multithreading introduced by Blumofe and Leiserson [20] views the execution of a multithreaded program[5] as a ***dag (directed acyclic graph)*** *D*, where the vertex set consists of ***strands*** — sequences of serially executed instructions containing no parallel control — and the edge set represents parallel-control dependencies between strands.

Figure 5-4 illustrates such a dag, which represents a program execution in that it involves executed instructions, as opposed to source instructions. In particular, it models an execution that contains spawns and syncs. As illustrated in Figure 5-4, a strand that has out-degree 2 is a ***spawn strand***, and a strand that resumes the caller after a spawn is called a ***continuation strand***. A strand that has in-degree at least 2 is a ***sync strand***. A strand can be as small as a single instruction, or it can represent a longer computation. Generally, we shall slice a chain of serially executed instructions into strands in a manner convenient for the computation we are modeling. We shall assume that strands respect function boundaries, meaning that calling or spawning a function terminates a strand, as does returning from a function. Thus, each strand belongs to exactly one function instantiation. For simplicity, we shall assume that programs execute on an ***ideal parallel computer***, where each instruction takes unit time to execute, there is ample memory bandwidth, there are no cache effects, etc. A strand's ***length*** is defined as the time a processor takes to execute all instructions in the strand.

[5] When we refer to the execution of a program, we shall generally assume that we mean "on a given input."

Work and span

The dag model admits two natural measures of performance which can be used to provide important bounds [19,23,40,53] on performance and speedup. The ***work*** of a dag $\mathcal{D}$ is the sum of the lengths of all the strands in the dag. The ***span*** of $\mathcal{D}$ is the length of the longest path in the dag. Assuming for simplicity that it takes unit time to execute a strand, the span for the example dag in Figure 5-4 is 10, realized by the path $\langle 1,2,3,6,7,8,10,11,18,19 \rangle$, and the work is 19.

Recall that Section 2.1 defines the work to be T_1, the execution time of a given computation on one processor, and the span to be T_∞, the execution time of the computation on an infinite number of processors. Section 2.1 also provides an execution-time bound on P processors in terms of T_1 and T_∞. For a program that is ***deterministic*** on a given input, where every memory location is updated with the same sequence of values in every execution, one can use T_1 and work or T_∞ and span interchangeably, since a deterministic program always behaves the same and results in the same execution dag on a given input, no matter how the program is scheduled. That is, the execution dag on a given input (and hence its work and span) for a deterministic program executing on a single processor is the same as the dag executing on multiple processors. For a ***nondeterministic*** program, however, where a memory location may be updated with a different sequence of values from run to run, different executions may result in different dags depending on the scheduling. Thus, we can no longer directly relate the work and span for a parallel execution to that of the serial execution. Rather, we must relate the work and span of a parallel execution to the resulting dag of the execution. Therefore, henceforth, we shall use the notation Work($\mathcal{D}$) and Span($\mathcal{D}$) to denote the work and span of a dag $\mathcal{D}$.

To generalize the bounds we have from earlier chapters for both deterministic and nondeterministic programs, we shall define the Work Law and the Span Law based on a given execution dag. Suppose that a program produces a dag $\mathcal{D}$ in time T_P when run on P processors of an ideal parallel computer. We have the following two lower bounds on the execution time T_P:

$$T_p \geq \text{Work}(\mathcal{D})/P\,, \qquad (5.1)$$

$$T_P \geq \text{Span}(\mathcal{D})\,. \qquad (5.2)$$

Similarly, the ***parallelism*** of the dag $\mathcal{D}$ is defined to be Work($\mathcal{D}$)/Span($\mathcal{D}$). Based on the dag, a work-stealing scheduler achieves the expected running time

$$T_P \leq \text{Work}(\mathcal{D})/P + O(\text{Span}(\mathcal{D}))\,, \qquad (5.3)$$

where we omit the notation for expectation for simplicity. This bound, which is proved in [20], assumes an ideal computer, but it includes scheduling overhead. As Section 2.1 explains, the computation exhibits linear speedup when the number of processors P is much smaller than the parallelism, since the first term dominates.

Copying with the nondeterminism of reducers

The bound shown in Inequality (5.3) applies to both deterministic and nondeterministic computations. Obtaining bounds on performance and speedup for a nondeterministic program can be more challenging, however. Unlike a deterministic program, we cannot readily relate the execution dag for a nondeterministic program resulting from a parallel execution to that of a serial execution.

A computation that uses a reducer generates a nondeterministic amount of work during a parallel execution. The question is, how much additional work, and how does it affect the work and span of the resulting dag. Leiserson and Schardl [96] provide a theoretical framework for analyzing an execution dag for a program that contains nondeterminism due to the use of a reducer, which allows us to obtain an upper bound on the additional work generated due to the use of a reducer and how the additional work impact the span of the computation, thereby obtaining bounds on performance and speedup. We will overview their framework and extend it to analyze a computation that uses a parallel reducer array.

The use of a reducer generates a nondeterministic amount of additional work, because accessing a reducer during parallel execution may implicitly cause the runtime system to create additional views for the reducer, which must be reduced later. The number of views created depends on the scheduling and cannot be determined solely by the execution dag from a serial execution, which is the only observable part from a user's perspective. To capture the nondeterminism due to a reducer, Leiserson and Schardl define two types of dags. First, they define the ***user dag*** $\mathcal{D}_\upsilon$ for a computation $\mathcal{D}$ in the same manner that we define an ordinary dag for a deterministic program. The user dag consists of only ***user strands***, which are observable during serial executions. Next, they define the ***performance dag*** $\mathcal{D}_\pi$, which is obtained by augmenting the user dag $\mathcal{D}_\upsilon$ with additional sets of ***runtime strands*** that the runtime system implicitly generates for managing a reducer. That is, given a parallel execution of a program with a user dag $\mathcal{D}_\upsilon = (V_\upsilon, E_\upsilon)$, one can obtain the performance dag $\mathcal{D}_\pi = (V_\pi, E_\pi)$, where

- $V_\pi = V_\upsilon \cup V_\iota \cup V_\rho$
- $E_\pi = E_\upsilon \cup E_\iota \cup E_\rho$,

where V_ι and E_ι represent the added ***init strands*** corresponding to view creations triggered by accessing or updating a reducer, and V_ρ and E_ρ represent the added ***reduce strands*** corresponding to instructions needed to reduce those views.

The vertex sets V_ι and V_ρ are based on the given parallel execution. The edge sets E_ι and E_ρ, on the other hand, are constructed a posteriori. For each init strand $v \in V_\iota$, we include (u, v) and (v, w) in E_ι, where $u, w \in V_\upsilon$ are the two strands comprising the instructions whose execution caused a view to be created (by invoking IDENTITY) corresponding to v. The construction of E_ρ is more complicated. To insert reduce strands, the edges in E_ρ are created in groups corresponding to the set of REDUCE functions that must execute before a given sync. Suppose that $v \in V_\upsilon$ is a sync strand, that k user strands $u_1, u_2, \ldots, u_k \in \mathcal{D}_\upsilon$ join at v, and that $k' < k$ reduce strands $r_1, r_2, \ldots, r_{k'} \in \mathcal{D}_\rho$ execute before the sync. One can define an ordering among the $k' + 1$ views seen by the k strands based on when the views are created. Leiserson and Schardl describe a construction for incorporating the reduce strands by repeatedly joining together two strands that have the "minimal" and "adjacent" views. The construction results in a ***reduce tree*** that incorporates all reduce strands between the k user strands and the sync node v, where the user strands are at the leaves, the reduce strands constitute intermediate nodes, and the sync node serves as the root. I omit the details of the construction here and refer interested readers to [96].

With this construction, the resulting graph $\mathcal{D}_\pi$ is indeed a dag. More importantly, one can apply the "delay-sequence" argument[6] due to Blumofe and Leiserson [20] to analyze the constructed $\mathcal{D}_\pi$ and show that every "critical" instruction is either sitting on top of some worker's deque or is being executed, including the reduce strands in V_ρ. The crucial observation is that, if an instruction in a reduce strand is critical, then its sync node (at the root of the reduce tree) has been reached, and thus a worker must be executing the critical instruction, since reduces are performed eagerly when nothing impedes their execution. Thus, whenever a worker steals, it has $1/P$ chance of executing a critical instruction. With constant probability, P steals suffice to reduce the span of the performance dag $\mathcal{D}_\pi$ by 1. Consequently, one can bound the expected running time of a computation $\mathcal{D}$ that uses

[6] This includes augmenting the performance dag $\mathcal{D}_\pi$ with additional "deque edges".

a reducer as

$$T_P(\mathcal{D}) \le \text{Work}(\mathcal{D}_\pi)/P + O(\text{Span}(\mathcal{D}_\pi)) \ . \tag{5.4}$$

and the expected number of steals is $O(P \cdot \text{Span}(\mathcal{D}_\pi))$.

Handling parallel REDUCE *operations*

In their analysis [96], Leiserson and Schardl assume that that the computation uses one reducer that has a nonconstant-time serial REDUCE operation. For our purpose, however, we shall assume that the computation uses a reducer that has a nonconstant-time parallel REDUCE operation, since our goal is to analyze a computation that uses a reducer array, whose REDUCE operation contains parallelism. Previously, with a serial REDUCE operation, when we construct the performance dag $\mathcal{D}_\pi$ for an execution, each REDUCE operation executed translates into either a single reduce strand or a chain of reduce strands between a user strand and a sync node in $\mathcal{D}_\pi$. Now with a parallel REDUCE operation, an executed REDUCE operation translates into a subdag between a user strand and a sync node in $\mathcal{D}_\pi$. This difference does not affect the delay-sequence argument, and Inequality (5.4) still holds. The main difference in the analysis for a serial REDUCE operation and a parallel REDUCE operation is how we relate the work and span of a performance dag to its user dag, which we shall discuss next.

Analyzing the work and span of a performance dag

Now we examine how one can relate the work and span of a performance dag to the user dag. The analysis we will discuss here closely follows the analysis described by Leiserson and Schardl [96] modified to handle a reducer with a parallel REDUCE operation. In particular, Leiserson and Schardl in their analysis assume τ to be the worst-case cost of any REDUCE or IDENTITY for the particular execution. In the case of a serial REDUCE, this τ parameter represents both the work and span of the worst-case cost of a REDUCE operation. In our analysis, we shall assume two distinct parameters τ_W and τ_S to represent the work and span of the worst-case cost of a REDUCE operation. For simplicity, we shall first assume that the parallel REDUCE operation we consider does not use yet another reducer. We shall come back to this point later. In addition, throughout the analysis, we shall assume that the computation uses a single reducer. Nevertheless, it is straightforward to use the same framework to analyze a computation that uses multiple reducers — simply assume τ_W and τ_S are the work and span of the worst-case cost of a hypermerge process.

First let's analyze and bound the additional work involved in joining strands together, which includes the REDUCE operations necessary before a sync node. Operationally, joining strands together corresponds to frames returning. Recall from Chapter 4, a returning frame must perform a locking protocol to prevent racing with its sibling frames who may also be returning. Once locks are acquired successfully, the frame returning obtains the necessary SPA maps to perform hypermerges until there is only one set of views left to deposit. Once the view transferal is done, a frame may eliminate itself from the steal tree. The next lemma bounds the work involved in joining strands together by considering the work involved in each elimination attempt and the total numbers of elimination attempts.

Lemma 5.1 *Consider the execution of a computation $\mathcal{D}$ on a parallel computer with P processors using a work-stealing scheduler. The total work involved in joining strands is $O(\tau_W P \cdot \text{Span}(\mathcal{D}_\pi))$, where τ_W is the work of the worst-case cost of any* REDUCE *or* IDENTITY *for the given input.*

PROOF. First, we shall bound the work involved in lock acquisition during an elimination attempt. Since we use the same locking protocol for acquiring SPA maps from siblings as described in [96], a lock is held only for a constant amount of time. Furthermore, as shown in [96], the time for the ith abstract lock acquisition by some worker w is independent of the time for w's jth lock acquisition for all $j > i$. Thus, by the analysis in [48], the total time a worker spends in lock acquisitions is proportional to the number of elimination attempts.

Next, we shall bound the total number of elimination attempts. Since each successful steal creates a frame in the steal tree that must be eliminated, the number of elimination attempts is at least as large as the number M of successful steals. Each elimination of a frame may force two other frames to repeat this protocol. Therefore, each elimination increases the number of elimination attempts by at most 2. Thus, the total number of elimination attempts is no more than $3M$.

Finally, let's consider the amount of work involved per elimination attempt. The total time spent acquiring abstract locks and performing the necessary operations while the lock is held is $O(M)$. Each failed elimination attempt triggers at most two hypermerge processes (each hypermerge combines two SPA maps into one) and at most view transferal. The work involved in a hypermerge and a view transferal is proportional to the number of reducers used. Assuming the computation employs a single reducer whose REDUCE operation involves τ_W amount of work in the worst-case, the total amount of work involved per elimination attempt is $O(\tau_W)$.

Putting everything together, we can bound the total expected work spent joining strands, which is $O(\tau_W M)$. Following the analysis on the number of steals from [20], which bounds the number of steals for a given dag $\mathcal{D}$ to be $O(P \cdot \text{Span}(\mathcal{D}))$, we have that the total work spent on joining strands is $O(\tau_W P \cdot \text{Span}(\mathcal{D}_\pi))$. □

Next, we shall bound the work and span of the performance dag in terms of the span of the user dag. We will consider the span (Lemma 5.2) first and the work (Lemma 5.3) separately.

Lemma 5.2 *Consider a computation $\mathcal{D}$ with user dag $\mathcal{D}_\upsilon$ and performance dag $\mathcal{D}_\pi$, and let τ_S be the span of the worst-case cost of any* CREATE-IDENTITY *or* REDUCE *operation for the given input. Then, we have* $\text{Span}(\mathcal{D}_\pi) = O(\tau_S \cdot \text{Span}(\mathcal{D}_\upsilon))$.

PROOF. Each successful steal in the execution of $\mathcal{D}$ may force one view to be created via an invocation of IDENTITY, which must be reduced later via REDUCE. Thus, each successful steal may lead to at most one IDENTITY and one REDUCE operation. Since each spawn in $\mathcal{D}_\upsilon$ provides an opportunity for a steal to occur, in the worst case, every spawn in $\mathcal{D}_\upsilon$ may increase the length of the path that contains the spawn by $2\tau_S$.

Consider a critical path in $\mathcal{D}_\pi$, and let p_υ be the corresponding path in $\mathcal{D}_\upsilon$. Suppose that k steals occur along the path p_υ. The length of that corresponding path in $\mathcal{D}_\pi$ is at most $2k\tau_S + |p_\upsilon| \leq 2\tau_S \cdot \text{Span}(\mathcal{D}_\upsilon) + |p_\upsilon| \leq 3\tau_S \cdot \text{Span}(\mathcal{D}_\upsilon)$. Therefore, we have $\text{Span}(\mathcal{D}_\pi) = O(\tau_S \cdot \text{Span}(\mathcal{D}_\upsilon))$. □

Lemma 5.3 *Consider a computation $\mathcal{D}$ with user dag $\mathcal{D}_\upsilon$. Let τ_W and τ_S be the work and span, respectively, of the worst-case cost of any* IDENTITY *or* REDUCE *operation for the given input. Then, we have* $\text{Work}(\mathcal{D}_\pi) = \text{Work}(\mathcal{D}_\upsilon) + O(\tau_W \tau_S P \cdot \text{Span}(\mathcal{D}_\upsilon))$.

PROOF. The work in $\mathcal{D}_\pi$ is the work in $\mathcal{D}_\upsilon$ plus the work represented in the runtime strands, i.e., init strands and reduce strands. The total work in reduce strands equals the total work to join stolen strands, which is $O(\tau_W P \cdot \text{Span}(\mathcal{D}))$ by Lemma 5.1. Similarly, each steal may create one init strand, and by the analysis of steals from [20], the total work in init strands is $O(\tau_W P \cdot \text{Span}(\mathcal{D}))$. Thus, we have $\text{Work}(\mathcal{D}_\pi) = \text{Work}(\mathcal{D}_\upsilon) + O(\tau_W P \cdot \text{Span}(\mathcal{D}_\pi))$. Applying Lemma 5.2 yields the lemma. □

Theorem 5.4 bounds the runtime of a computation whose nondeterminism arises from reducers.

Theorem 5.4 *Consider the execution of a computation $\mathcal{D}$ on a parallel computer with P processors using a work-stealing scheduler. Let $\mathcal{D}_\upsilon$ be the user dag of $\mathcal{D}$. The total running time of $\mathcal{D}$ is* $T_P(\mathcal{D}) \leq \text{Work}(\mathcal{D}_\upsilon)/P + O(\tau_W\tau_S \cdot \text{Span}(\mathcal{D}_\upsilon))$.

PROOF. By Inequality (5.4) and Lemmas 5.2 and 5.3, we have $\text{Work}(\mathcal{D}_\upsilon)/P + O(\tau_W\tau_S \cdot \text{Span}(\mathcal{D}_\upsilon)) + O(\tau_S \cdot \text{Span}(\mathcal{D}_\upsilon))$. We can omit the third term $O(\tau_S \cdot \text{Span}(\mathcal{D}_\upsilon))$, since it is dominated by the second term $O(\tau_W\tau_S \cdot \text{Span}(\mathcal{D}_\upsilon))$. □

In the case of a parallel reducer array, since its REDUCE operation uses a bag reducer, the REDUCE operation generates a nondeterministic amount of work during parallel execution. Thus, we must recursively apply the analysis to the work and span for the REDUCE operation for the parallel reducer array in order to obtain the appropriate bounds for τ_W and τ_S. That is, consider the subdag $\mathcal{D}'$ that corresponds to the worst-case cost of a REDUCE operation for the parallel reducer array. We are looking for the work and span of $\mathcal{D}'_\pi$, which correspond to the terms τ_W and τ_S from Theorem 5.4. Let τ'_W and τ'_S be the work and span of $\mathcal{D}'_\upsilon$ respectively, and let τ''_W and τ''_S be the work and span of the worst-case cost of the REDUCE operation from the bag reducer used in the REDUCE operations of the parallel reducer array. By applying Lemma 5.2 and Lemma 5.3, we have

$$\begin{aligned} \tau_S &= O(\tau'_S\tau''_S) , \\ \tau_W &= \tau'_w + O(\tau'_s \cdot \tau''_w\tau''_s P) . \end{aligned} \tag{5.5}$$

With this bound, we define the ***effective parallelism*** as $\text{Work}(\mathcal{D}_\upsilon)/(\tau_W\tau_S \cdot \text{Span}(\mathcal{D}_\upsilon))$. Just as with the parallelism defined for deterministic computations, if the effective parallelism exceeds the number P of processors by a sufficient margin, the P-processor execution can obtain near-linear speedup over the serial execution. The second term in the time bound gives an upper bound on the overhead incurred by all the REDUCE operations in the computation, which stays the same no matter how many processors are used to execute $\mathcal{D}$, since the maximum number of views created is proportional to the number of processors used for execution. As the effective parallelism implies, this bound gives us an intuition as to whether one can expect a computation to scale when a reducer array is used. Specifically, it depends on the total work in $\mathcal{D}_\upsilon$ and how much work is involved in REDUCE operations (which corresponds to the size of the reducer array used). If the overall work of the computation is comparable to the work and span involved in the REDUCE operations for the reducer used in the computation, one should not expect to see linear speedup when running the computation on multiple processors. On the other hand, when the work involved in the REDUCE operations is large, parallelism in REDUCE indeed helps. As we shall see in our case study in Section 5.3, experimental results bear out these observations.

5.3 An Empirical Evaluation of Reducer Arrays

This section empirically evaluates the library implementations of reducer arrays by comparing the space utilization and performance of reducer arrays to that of arrays of reducers. Recall from Section 5.1 that there are two library implementations — an ordinary reducer array which keeps the logs in a vanilla array and employs a serial REDUCE operation, and a parallel reducer array which keeps the logs in a bag reducer and employs a parallel REDUCE operation. In terms of space usage, experimental results show that both implementations of reducer arrays use less space than an array of reducers. In terms of execution time, both implementations of reducer arrays perform about 2× better than an array of reducers when one enables the lookup optimization. Without the lookup optimization, the performance difference is negligible when the array size is small but becomes noticeable as array size increases, especially during parallel executions. Furthermore, the use of a bag reducer in a parallel reducer array has negligible overhead compared to the use of a vanilla array in a reducer array, and its parallel REDUCE operation indeed helps in the event when the array size is large.

General setup. The library implementations of reducer arrays are evaluated using one microbenchmark and one application benchmark. The microbenchmark is synthetic, designed to perform random array accesses repeatedly. The evaluation uses the microbenchmark to compare space utilization and performance of reducer arrays and arrays of reducers. This evaluation also includes a case study using a real-world application, parallel breadth-first search (or PBFS) [96], modified to include "parent computations" that employ a reducer array.

Both the microbenchmark and the PBFS application benchmark were compiled using the Cilk Plus compiler version 12.0.0 using `-O2` optimization. All experiments were performed on an AMD Opteron system with 4 quad-core 2 GHz CPU's having a total of 8 GBytes of memory. Each core on a chip has a 64-KByte private L1-data-cache and a 512-KByte private L2-cache, but all cores on a chip share a 2-MByte L3-cache. With 4 quad-cores, the system has a total of 8-MByte L3-cache.

All experiments were conducted with the Cilk-M runtime system (specifically, Cilk-M 1.0). This evaluation does not include performance comparison with Cilk Plus [68]; although the libaray implementations of reducer arrays work with Cilk Plus, the reducer mechanism in Cilk Plus does not support parallel REDUCE operations. Please refer to Section 4.4 for performance comparisons for the reducer mechanisms between Cilk-M and Cilk Plus.

Reducer pointer interface. Both the microbenchmark and the PBFS application are coded using the reducer pointer interface (see Section 5.1 for a description of the reducer pointer interface). There isn't fundamental performance difference bwteen the reducer interface and the reducer pointer interface. When a program uses the reducer pointer interface, however, it may suffer from ***false sharing***, where different workers compete for a cache line when they write to different memory locations that happen to be allocated on the same cache line. In the case of using a reducer pointer, the false sharing occurs when the leftmost view is small enough to share a cache line with other (possibly read-only) variables. Thus, when multiple workers inevitably update the leftmost view during parallel execution, variables which happen to lie on the same cache line get bounced between private L1-caches of different cores, and incur significantly more overhead compared to single-processor executions. The false sharing does not occur if one uses the reducer interface, because the leftmost view is allocated as part of the reducer object, which is large enough to occupy its own cache line. Nevertheless, this false-sharing problem can be easily fixed with padding once the programmer realizes what is causing the slowdown during parallel executions and where the false sharing occurs.

In the absense of false sharing, performance between the two interfaces is comparable when the nubmer of reducers used is moderate. When the number of reducers used is large, however, the reducer pointer interface has a slight advantage in that it requires less space. For instance, an add reducer (which includes its leftmost view) takes up 192 bytes, whereas a reducer pointer (which excludes its leftmost view) takes up 96 bytes. Even accounting space taken up by the leftmost view, a reducer pointer still uses less space. This advantage is evident when the microbenchmark is evaluated with a large array of reducers, since the microbenchmark becomes memory-bandwidth bound in this case. Thus, all experimental results shown in this section employ the reducer pointer interface and include the fix to false sharing.

Evaluation using the microbenchmark

The microbenchmark works as follows — it generates an array of random indices and updates the array of reducers or the reducer array repeatedly using the random indicies. The parallelism comes from recursivly subdividing the iteration space and traversing the iteration space in parallel, so different workers are updating the array of reducers or the reducer array in parallel, writing to the same array indices according to the random index array.

There are two input parameters to the microbenchmark that can be adjusted. The first is the size of the random index array, which dictates how densely the array of reducers or the reducer array is accessed. The microbenchmark is evaluated with density values ranging from 0.1–0.9 (sparse to dense) with a 0.1 increment. The second parameter is the size of the array of reducers or the reducer array. The micobenchmark is evaluated with three different array sizes — 8192 (small), 32768 (medium), and 262144 (large). The number of iterations in the benchmark was chosen according to the array size and access density in such a way that the benchmark runs

Space usage. For either reducer arrays or arrays of reducers, the space overhead includes the folliwng:

1. space allocation for private SPA maps in workers' TLMM reducer regions throughout reducers' lifespan,
2. space allocation for public SPA maps during hypermerges,
3. space allocations for their corresponding reducer pointers,
4. space allocations for the leftmost views, and
5. space allocations for newly allocated local views due to parallel execution.

In the case of serial executions, the runtime uses zero space for overheads 2 and 5. Thus, it is easy to see that an array of reducers consums more space than a reducer array (for either implementation), because an array of reducers incurs high cost in overheads 1 and 3 simply due to the high nubmer of reducer pointers that it employs.

During parallel executions, it is no longer a clear cut which variant uses more memory. Even though an array of reducers incurs high costs in overheads 1, 2, and 3, it incurs relatively lower cost in overhead 5 than a reducer array, because an array of reducers only creates views for elements accessed whereas a reducer array creats a SPA view for the entire array.

I measured the space usage for array of reducers, ordinary reducer arrays (with and without the lookup optimization), and parallel reducer arrays (with and without the lookup optimization) during parallel executions, using the microbenchmark with three different array sizes and across access densities. Experimental results show that both implementations of reducer arrays consume less space than an array of reducers. In particular, when the array size is large, a parallel reducer array uses the least amount of memory of the three.

Figure 5-5 summarizes the experimental results in three graphs, one for each array size tested. Within each graph, three different variants are shown, grouped into a cluster: an array of reducers, an ordinary reducer array, and a parallel reducer array. For both implementations of reducer arrays, the space usages with and without the lookup optimization are pretty comparable. Thus, Figure 5-5 shows only data obtained with the lookup optimization. Within each variant, Figure 5-5 selectively shows the space usages on executions with access densities of 0.1, 0.3, 0.6, and 0.9 to simplify the presentation. For each access density, the bar presents a breakdown of the space utilization into three different categories. The first category is overhead for private SPA maps, which corresponds to overhead 1, calculated by the number of physical pages mapped in workers' TLMM reducer regions. The second category is overhead for allocating public SPA maps, which corresponds to

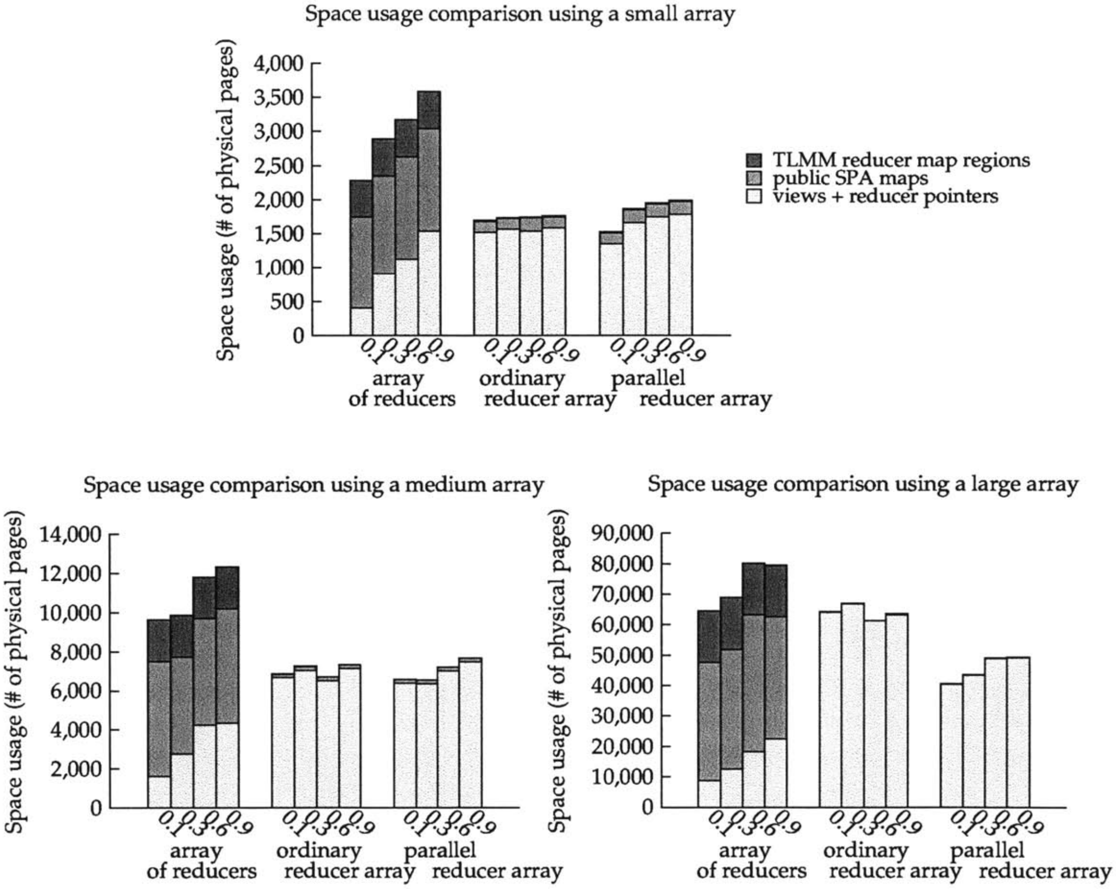


Figure 5-5: The breakdown of space usage of the microbenchmark using a small array, a medium array, and a large array. In each graph, the space usage for three different variants are shown, one per cluster: an array of reducers, an ordinary reducer array with the lookup optimization, and a parallel reducer array with the lookup optimization. Within each cluster, the x-axis labels the access density. For all graphs, the y-axis labels the space usage in the number of physical pages.

overhead 2, calculated by the total number of physical pages the runtime system requested from the operation system for public SPA maps in a given execution. Since pages for public SPA maps are recycled in the runtime system, this number shows the maxinum number of pages needed for public SPA maps during the execution. The third category is overhead for allocating views and reducer pointers, which corresponds to overheads 3–5, inclusively. For both implementations of reducer arrays, the SPA views are recycled on the per-worker basis, so the number shows the maxinum number of pages needed for SPA views during the execution.[7] Since the space usage for these categories differ from run to run due to scheduling, for each data point, the microbenchmark was run 10 times and recorded the maximum number of pages used.

As Figure 5-5 shows, even though an array of reducers tends to use less space in creating views than both implementations of reducer arrays, its space usage is dominated by allocating SPA maps during hypermerges. Once the space for (public and private) SPA maps is accounted for, reducer arrays end up using less space. In particular, the parallel reducer array consumes about 60%–70% of the space consumed by the array of reducers in the test cases.

Somewhat surprisingly, an ordinary reducer does not necessarily save on space compared to a parallel reducer, even though a parallel reducer uses a bag reducer in its REDUCE operation, which generates more views during parallel execution. The reason is that a parallel reducer array uses the bag data structure to store logs, and the bag allocates space lazily, whereas an ordinary reducer array uses a vanilla array to store logs, which is allocated when a view is created. That means the bag has a much more compact representation than an array when the number of logs is small and the array size is large.

Performance comparison. The same microbenchmark was used to evaluate the performance of the three variants. For reducer arrays, I was interested in seeing how much the lookup optimization helps, where one lookup is performed within a single strand instead of multiple lookups (i.e., one lookup per array element accessed), so the evaluation also includes time measurements of reducer arrays with and without the lookup optimization.

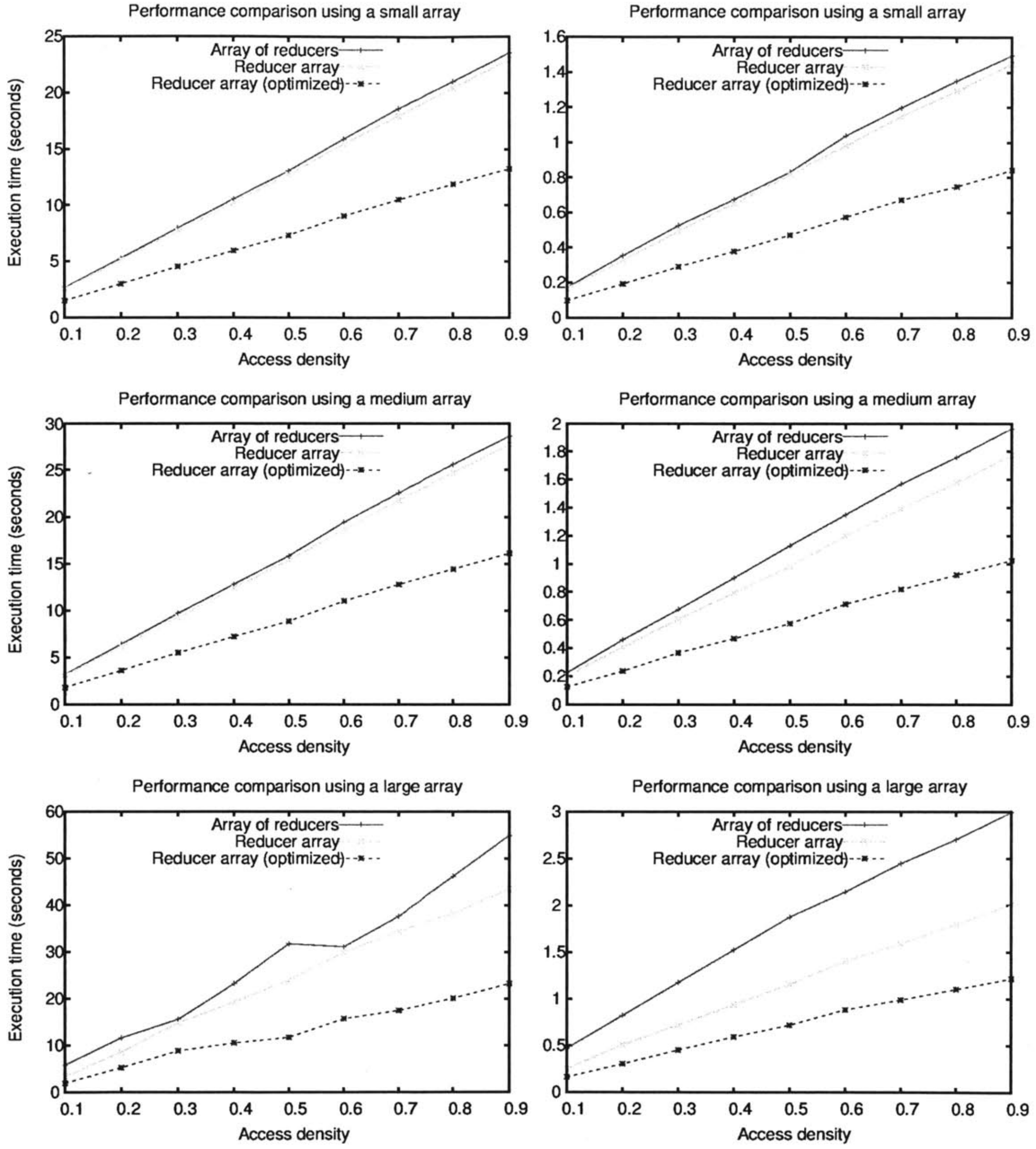
It turns out that, with arrays of the sizes tested for the microbenchmark, the performance of an ordinary reducer array and a parallel reducer array are quite comparable, and so figures include only the execution times of benchmarks using arrays of reducers and parallel reducer arrays.[8] We shall defer the discussion on the difference between reducer arrays and parallel reducer arrays until the case study.

Figure 5-6 column (a) shows the performance comparison between benchmark executions that use an array of reducers, a parallel reducer array without the lookup optimization, and a parallel reducer array with the lookup optimization running on a single processor. Figure 5-6 column (b) shows the same performance comparison when running on 16 processors. Three different array sizes are shown in each column.

Let's first examine Figure 5-6 column (a) for the single-processor executions. The performance difference between reducer arrays with and without the lookup optimization stays constant across different array sizes, where the reducer array with the optimization runs about 1.8× faster. This makes sense, since these two variants use about the same amount of memory, and the performance difference results purely from the optimization.

[7] Although the SPA views are recycled, a parallel reducer array uses a bag reducer in its REDUCE operation, and views for the bag reducer are not recycled.

[8] The ordinary reducer array performs slightly better than the parallel reducer array when the microbenchmark uses a small or medium array, whereas the parallel reducer array performs slightly better than the ordinary reducer array when the microbenchmark uses a large array. In all cases, the performance difference is small enough that including the timing on both does not add much information to Figure 5-6.



(a) single-processor executions **(b) 16-processor executions**

Figure 5-6: The execution times of the microbenchmark using a small array, a medium array, and a large array executing on **(a)** a single processor and on **(b)** 16 processors. The y-axis labels the execution times in seconds, and the x-axis labels the varying access densities ranging from 0.1–0.9. There are three variants of the benchmark — one using an array of reducers, one using a parallel reducer arrays without the lookup optimization, and one using a parallel reducer array with the lookup optimization.

On the other hand, there is no performance difference between an array of reducers and a reducer array without the optimization when the array size is small or medium. For the small and medium arrays, since only one view per array element is ever created during single processor executions, the amount of memory used by either variant (which includes the left most view and the reducer pointers created) fits comfortably in the L3-cache (total of 8 MBytes). Even though the array of reducers ends up using more memory and incurs more L1- and L2-cache misses, the additional cache misses does not impact the performance in a significant way. For the large array, however, the amount of memory used by either variant no longer fits in the L3-cache, and the amount of memory used by the two variants differ enough to make a performance impact, although not too significant.

For 16-processor executions, again, the performance difference between reducer arrays with and without the lookup optimization stays about constant across array sizes. On the other hand, performance difference between an array of reducers and a reducer array without the optimization starts to show in the medium-sized array test case, and the gap widens when the array size increases. An array of reducers consumes more space, and the large-sized array does not fit in the main memory, so the space consumption probably impacts the performance. Another important factor is the reduce overhead incurred during parallel executions. An array of reducers incurs much higher overhead in performing view transferal than a reducer array, simply due to its use of many reducer pointers. Furthermore, a reducer array likely has an advantage in locality during the hypermerge process when the access density is above 0.5. Even though the microbenchmark accesses the array using random indices, for a reducer array, a hypermerge process involves simply combining two SPA views. When the access density is above 0.5, the SPA view no longer keeps the access logs, and its REDUCE operation walks the underlying value arrays in order. For an array of reducers, on the other hand, a hypermerge process involves reducing multiple pairs of views together (one pair per element accessed), and there is not much locality among the pairs of views. Finally, in the large array test case, a parallel reduce array has an advantage in that its REDUCE operation contains parallelism — it does perform slightly better compared to its counterpart, an ordinary reducer array. All these reasons contribute to the lower reduce overhead in a reducer array than an array of reducers during parallel executions. Indeed, instrumentation in the runtime system indicates that an array of reducers spends much more time performing view transferals and hypermerges than a reducer array when the array size is large.

A case study using PBFS with parent computations

The case study used to evaluate the performance of reducer arrays is parallel breadth-first search [96], or PBFS. The base algorithm is summarized in Section 4.4. For the purpose of evaluating reducer arrays, I modified the algorithm to perform parent computations, which requires either an array of reducers or a reducer array in order to compute parents in a deterministic fashion.

PBFS with parent computations works as follows. As the algorithm discovers the shortest path from the starting node v_0 to some node v_n, it records v_n's ***parent***, the ancestor node that leads to v_n in the shortest path. The algorithm records the parents of all nodes in a nonlocal array of size $|V|$, i.e., the size of vertex set of the input graph. As workers discover different paths that lead to the same node, two worker may potentially update the same element in the array in parallel (assuming the two paths have the same distance from starting node v_0). In such a case, the algorithm breaks the tie between the two parents having the same distance according to their vertex IDs, where the parent with a smaller ID gets recorded. To do so, the algorithm employs an array of reducers or reducer array whose REDUCE operation is a min operation.

The application is evaluated using both implementations of reducer arrays to examine the impact of the parallel REDUCE operations on the overall performance. Experiements using the microbenchmark have established that a reducer array works equally well or better than an array of reducers. In particular, the lookup optimization indeed helps. Thus, this case study focuses on evaluating the difference between an ordinary reducer array (with a serial REDUCE operation) and a parallel reduce array (with a parallel REDUCE operation), where the difference is only evident empirically when the application requires large reducer arrays, which is the case for PBFS with parent computations.

Theoretical bound. We shall first examine how the execution time bounds compare when the application uses an ordinary reducer array versus a parallel reducer array. Recall from Section 5.2 Theorem 5.4 that a computation $\mathcal{D}$ that uses a reducer array executing on P processors has a time bound $T_P(\mathcal{D}) \leq \text{Work}(\mathcal{D}_\upsilon)/P + O(\tau_W \tau_S \cdot \text{Span}(\mathcal{D}_\upsilon))$. The work and span for PBFS with parent computations is asymptotically the same as the work and span for PBFS, since the parent computations simply add additional constant overhead per vertex processed. Thus, given an input graph $G = (V,E)$ with diameter D, the work of PBFS with parent computations is $O(V+E)$, and the span is $O(D\lg(V/D) + D\lg\Delta)$, where Δ is the maximum out-degree of any vertex in V [96].[9]

Consider a PBFS computation $\mathcal{D}$ that uses an ordinary reducer array for parent computations. With a serial REDUCE operation, in the worst-case, both the work and span of a REDUCE operation[10] can be as much as V, since V is the size of the parent array. Thus, a PBFS computation $\mathcal{D}$ that uses an ordinary reducer array has the following time bound:

$$T_P(\mathcal{D}) \leq O(V+E)/P + O(V^2 \cdot (D\lg(V/D) + D\lg\Delta)). \tag{5.6}$$

Recall from Section 5.2 that the second term constitutes the worst-case overhead for performing all REDUCE operations. While this overhead is an upper bound, the fact that the second term dominates the first term tells us that one should not expect PBFS with parent computations using a reducer array to scale well.

If the computation uses a parallel reducer array that supports a parallel REDUCE operation, the work and span for the worst-case REDUCE operation without considering the overhead from using the bag reducer, are V and $\lg V$ respectively (which corresponds to the terms τ'_W and τ'_S in Equation (5.5)). The bag reducer used in the array's REDUCE operation has the worst case work and span of $O(\lg V)$ for its own REDUCE operation (which corresponds to the terms τ''_W and τ''_S in Equation (5.5)), because a bag may contain as many as $O(V)$ nodes. Then, the worst-case work and span for a REDUCE operation, including the overhead of using a bag reducer are $\tau_W = O(V + P\lg^3 V)$ and $\tau_S = O(\lg^2 V)$, respectively. Thus, a PBFS computation $\mathcal{D}$ that uses a parallel reducer array has the following time bound:

$$\begin{aligned} T_P(\mathcal{D}) &\leq O(V+E)/P + O((V\lg^2 V + PV\lg^5 V) \cdot (D\lg(V/D) + D\lg\Delta)) \\ &= O(V+E)/P + O(PV\lg^5 V \cdot (D\lg(V/D) + D\lg\Delta)). \end{aligned} \tag{5.7}$$

Even though the reduce overhead in Inequality (5.7) grows slower asymptotically than the reduce overhead in Inequality (5.6), I cannot sensibly compare the two bounds, because I don't know the constant factor involved in the various terms, and the bound is only an upper bound on execution time. Moreover, with the input sizes used to evaluate PBFS, the slower asymptotic growth of the $lg^5 V$ term than the V term does not kick in until V becomes fairly large. The only thing one can

[9] The notation for set cardinality is omitted within the time bound for clarity.

[10] Even though the analysis in Section 5.2 considers the work and span of the worst case of RECUCE or IDENTITY operations, we simply drop the IDENTITY in the discussion here for simplicity. This does not affect the correctness of the analysis for PBFS with parent computations, since for this particular application, the work and span of a REDUCE operation is the same as that of an IDENTITY operation.

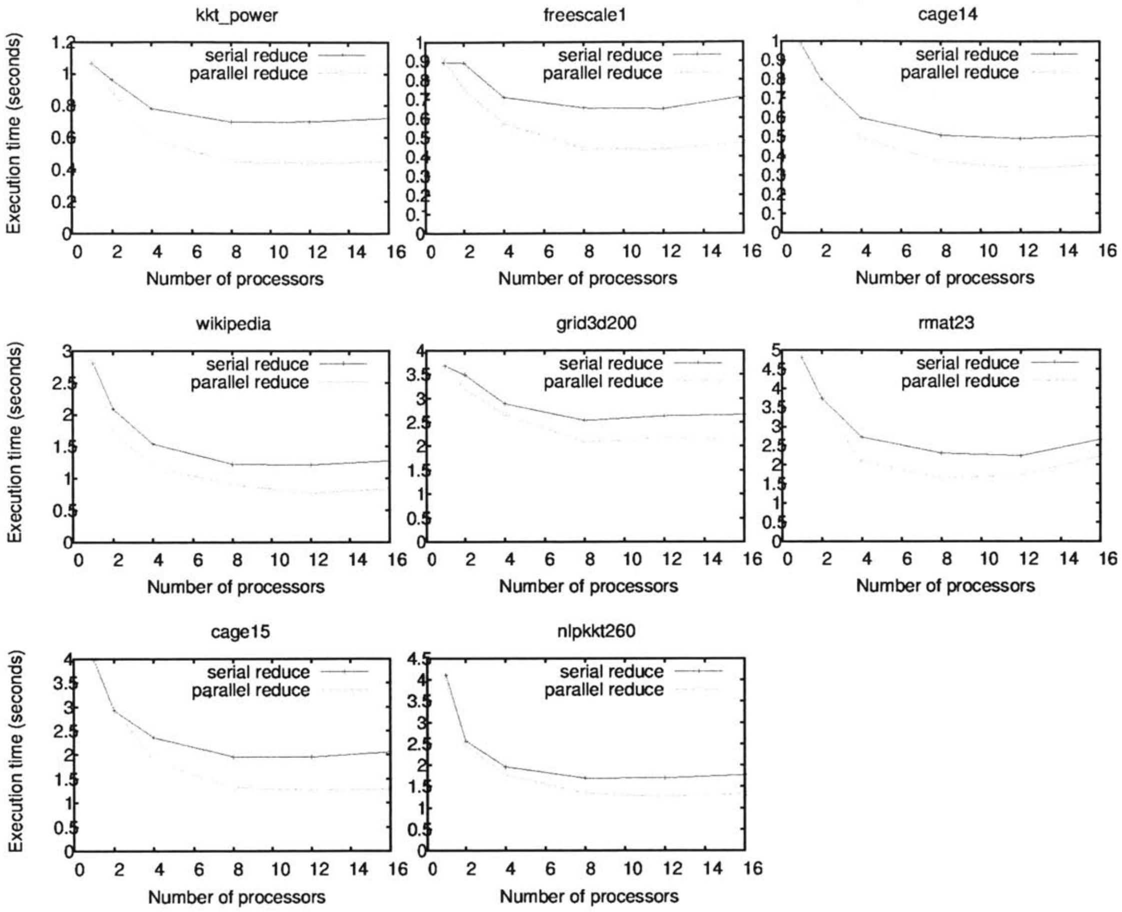


Figure 5-7: The execution times of PBFS with parent computations running on 1, 2, 4, 8, 12, and 16 processors using 8 different input graphs. For each configuration, two variants of reducer arrays are used — the reducer array with a serial REDUCE operation and the parallel reducer array with a parallel REDUCE operation. The lookup optimization is employed for both variants. Each figure shows the execution times for a given input graphs. The y-axis labels the execution times in seconds, and the x-axis labels the number of processors used.

conclude from this bound is that, the second (reduce overhead) term still dominates the first (work) term, and so one should not expect PBFS with parent computations using a parallel reducer array to scale, either.

Empirical results. Now we examine the empirical results of PBFS using an ordinary reducer array and a parallel reducer array. I evaluated PBFS using 8 different input graphs, each with the number of vertices on the order of millions (the sizes of vertex- and edge-sets can be found in Figure 4-11). That means that each execution uses a reducer array of size in the order of millions. I also evaluated PBFS with parent computation using an array of reducers in Cilk-M, but the results are not shown here — when using an array of reducers, PBFS gets linear slowdown, and it sometimes runs out of memory when executed on 12 or 16 processors.

Figure 5-7 shows the execution times for each input graph executing on 1, 2, 4, 8, 12, and 16 processors using either a reducer array or a parallel reducer array. Each data point represents the average of 10 runs with standard deviation equal or less than 3%, except for the 16-processor executions, which have standard deviation ranging from 0.33%–11.21% depending on the input graph. The computation using a parallel reducer array consistently performs better than the computation using an ordinary reducer array, especially when the number of processors increases. As the bounds predict, however, neither computation scales — the best speedup one gets on any graph is at most 3×. Furthermore, the execution time curve tends to plateau around 12 processors, sometimes with a 16-processor execution taking longer time.

The fact that PBFS with parent computations does not scale well, even when using a parallel reducer array, poses a question of whether it is a good idea to use large number of reducers or a reducer array with large size. This is not to say that an application using a large reducer array cannot possibly scale. For instance, if an application has quadratic amount of work with logarithmic span in the user computation and uses a reducer array with size less than linear with respect to the input size, the computation could scale. I have yet to find such a computation that requires a reducer array with such work and span profiles, however.

5.4 Concluding Remarks

Reducer hyperobjects seem to be a useful memory abstraction. As told by the practitioners in the field — researchers and engineers who have worked on parallelizing large applications using Cilk++ and Cilk Plus — it would have been difficult to parallelize some of the large applications which they encountered without the use of reducer hyperobjects. The use of reducer hyperobjects, like any synchronization mechanism I know of, has its own shortcomings, in particular, the inherent overhead associated with managing views. While this shortcoming is small when the computation uses only a constant number of reducers or the overall REDUCE operations take constant time, in the case of a reducer array, this overhead may constitute a scalability bottleneck. As we have seen both theoretically and empirically in Sections 5.2 and 5.3, this is indeed the case if the work involved in REDUCE operations dominates or even is simply comparable to the work involved in the user computation. The particularly troubling bit is that the number of views created, and hence the reducer overhead involved, grows proportionally to the number of workers executing the computation.

Whether a reducer array constitutes a useful memory abstraction remains an open question. Even though reducer arrays seem to be a natural extension to reducer hyperobjects, I have yet to find an application that requires a reducer array to compute deterministically and scales well at the same time. The PBFS example used in our case study neither scales well nor does it require a reducer array to compute deterministically. Given that the type of reduce operation used in the application is both associative and commutative, one could simply allocate a nonlocal array for parent computations and use compare-and-swap (CAS) to update an element in the array as it discovers different shortest paths to a given vertex. The final result ought to be deterministic still, assuming the algorithm simply uses vertex IDs to break tie — the parent with a smaller ID wins out in the end. This CAS implementation would conceivably scale better than using a reducer array. Of course, this strategy only works because the operation on the parent array is both associative and commutative. Until we find an application that absolutely requires a reducer array, we cannot say that a reducer array constitutes a useful memory abstraction. Even if we do find such an application, it may be fruitful to consider other alternatives for avoiding determinacy races that is as general as reducer arrays but incurs less overhead, which in turn may lead us to a more efficient reducer-like mechanism.

Part II:
Other Memory Abstractions

Chapter 6

Ownership-Aware Transactional Memory

Transactional memory (TM) [64], another type of memory abstraction, is meant to simplify concurrency control in parallel programming by providing a transactional interface for accessing memory; the programmer simply encloses the critical region inside an `atomic` block, and the TM system ensures that this section of code executes atomically. When using TM, one of the issues that the programmer must deal with is the semantics of "nested" transactions. Previous proposals for handling nested transactions either create a large memory footprint and unnecessarily limit concurrency, or fail to guarantee "serializability" [121], a correctness condition often used to reason with TM-based programs, and possibly produce anomalous program behaviors that are tricky to reason about. This chapter explores a new design of a TM system which employs "ownership-aware transactions" (OAT) that admit more concurrency and provide provable safety guarantees, referred to as "abstract serializability."

Without considering the semantics of nested transactions, the basic concept of transactional memory is fairly straightforward. A TM system enforces atomicity by tracking the memory locations that transactions access (using ***read sets*** and ***write sets***), finding transaction "conflicts," and aborting and retrying transactions that conflict. Two executing transactions are said to ***conflict*** if they access the same memory location, with (at least) one of the accesses being a write. If a transaction completes without generating a conflict, the transaction is said to be ***committed***, at which point its updates are reflected in the global memory. If a transaction generates a conflict, the TM system may choose to ***abort*** the transaction in order to resolve the conflict. Any update to memory from an aborted transaction is not "visible" to other transactions, and the transaction is rolled back to the beginning, possibly being retried later. By aborting and retrying transactions that conflict, the TM system guarantees that all committed transactions are ***serializable*** [121]; that is, transactions affect global memory as if they were executed one at a time in some order, even if in reality, several executed concurrently.

Transactions may be ***nested***, where a transaction Y is dynamically enclosed by another transaction X. If Y is ***closed nested*** [112] inside X, then for the purpose of detecting conflicts, the TM system considers any memory locations accessed by Y as conceptually also being accessed by its parent X. Thus, when Y commits, the TM system merges Y's read and write sets into the read and write sets of X.[1] TM with closed-nested transactions guarantees that transactions are serializable at

[1] Y can also be ***flat-nested*** inside of X. Flat-nesting has similar semantics to close-nesting in the sense that memory locations accessed by Y are conceptually also being accessed by X, but instead of merging Y's read and write sets into X's when Y commits, the transaction Y is simply eliminated and executed as part of X. While this is a subtle difference,

Transaction X_1

```
1  //compute k1
2   ...
3  atomic {  //Transaction Y1
4     if(tree.contains(k1)==false)
5       tree.insert(k1);
6  }
7  //other long computation
8   ...
```

Transaction X_2

```
10  //compute k2
11   ...
12  atomic {  //Transaction Y2
13     if(tree.contains(k2)==false)
14       tree.insert(k2);
15  }
16  //other long computation
17   ...
```

Figure 6-1: Two transactions X_1 and X_2 from a user program that may execute concurrently. Each transaction performs some computation to calculate the key to insert into a shared balanced binary search tree. The user program first checks that the key is not already present before inserting it into the tree. To avoid duplicate keys, the invocations to `contains` and `insert` ought to be executed in an atomic fashion. The user program express this intent by surrounding the calls with an `atomic` block, which generates inner transactions Y_1 and Y_2 of X_1 and X_2 respectively.

the level of memory. Researchers have observed, however, that closed nesting might unnecessarily restrict concurrency in programs because it does not allow two "high-level" transactions to ignore conflicts due to "low-level" memory accessed by nested transactions.

A simple scenario illustrates why closed nested transactions may unnecessarily restrict concurrency in programs. Consider a user program that processes a set of data, performs some computation to generate keys, inserts the generated keys into a balanced binary search tree, and performs some other computation. The code that processes data is enclosed by an `atomic` block, which generates transactions X_1 and X_2 shown in Figure 6-1. The balanced binary search tree instance is provided by a library, which supports functions such as `insert`, `contain`, and `remove`. At the end of each insert or remove operation, the tree performs rotations to rebalance itself. From the user program's perspective, it does not care about the order in which the keys are inserted, as long as no duplicates exist. This intention is expressed by another `atomic` block in lines 3 and 12, ensuring that the invocations to `contains` and `insert` execute atomically. The `atomic` block generates inner transactions inside of X_1 and X_2, referred to as Y_1 and Y_2 respectively.

Since the user program does not care about the order in which the keys are inserted, it does not care whether Y_1 occurs before or after Y_2, as long as each of them appears to execute as an atomic unit. That is, assuming no other conflicts occur in the prefixes and suffixes of X_1 and X_2, the following schedule would be an acceptable outcome from the user's perspective: lines 1–2, lines 10–11, lines 3–6, lines 12–15, lines 7–8, and lines 16–17. Using closed-nesting, however, if subtrees accessed by Y_1 and Y_2 happen to overlap, this schedule will not allowed. Without loss of generality, let's assume that Y_1 causes rotations in the subtree it accessed but commits before Y_2 begins. If Y_2 happens to traverse through nodes modified during rotations performed in Y_1, Y_2 will generate a conflict with X_1, because Y_1 merges its read and write sets with that of X_1, its parent, when it commits, and the underlying TM system must abort one of X_1 or Y_2 to resolve the conflict. A user may find this need to abort undesirable because it unnecessarily limits concurrency; even though the schedule given above is not serializable at the level of memory, it is "abstractly serializable" from the level of program semantics. Once Y_1 commits, X_1 operates at the level of the user program and no longer cares about the low level changes made to the tree nodes, provided that Y_1 completed execution as a atomic unit. Using closed nesting, transactions X_1 and X_2 cannot execute concurrently, unless they access separate parts of the binary tree.

flat-nesting would not work as expected if one allows parallelism inside a transaction. For the purpose of describing the problem addressed by ownership-aware transactions, we will simply focus our attention on closed-nesting.

```
1 bool contains(Key k) {
2   bool empty = false;
3   open_atomic{
4     empty = (this.size == 0);
5   }
6   if(empty) return false;
7   //otherwise search the tree
8     ...
9 }
```

Figure 6-2: An erroneous implementation of the `contains` method of the binary search tree library, where the read of the `size` field is enclosed in a open-nested transaction.

To allow more concurrency of transactions in such examples, researchers have proposed the ***open-nested commit mechanism*** [106, 113, 114]. When an open-nested transaction Y (enclosed within another transaction X) commits, Y's changes are committed to memory and become visible to other transactions immediately, independent of whether X later commits or aborts. Once Y commits, its read and write sets are discarded without merging into X's read and write sets.[2] Thus, the TM system no longer detects conflicts with X due to memory accessed by Y. In other words, the open-nested commit mechanism provides a loophole in the strict guarantee of transaction serializability by allowing the outer transaction to ignore memory operations performed by its open-nested subtransactions. Going back to our example scenario, if Y_1 and Y_2 are open-nested inside X_1 and X_2 instead, the TM system will no longer detect conflicts between X_1 and Y_2 (assuming Y_1 commits before Y_2 begins), since the TM system no longer keeps track of Y_1's read and write sets as part of X_1 once Y_1 commits.

Once the TM system supports open-nested commits, however, it can permit nonserializable schedules, some of which may be considered desirable by the programmer, while others may lead to incorrect executions. For instance, imagine that the library implementer of the balanced binary search tree decides to add a field `size` to keep track of the number of items in the tree, and subsequently uses it in the `contains` method as shown in Figure 6-2. The `contains` method first checks whether the tree is empty, and only searches the tree if it is not empty. Given that the `size` field can be highly contended, the library implementer mistakenly decides that it will be a good "optimization" to enclose this read of the `size` field in an open-nested transaction, call it transaction Z (lines 3–5), which would exclude conflict on this read of `size` field if Z is enclosed within another transaction. An unintended consequence of this "optimization" is that a transaction from the user program calling both `contains` and `insert` can still commit even though the transaction no longer appears to execute atomically — assuming the tree is empty when Y_1 begins, another transaction may come in and insert the same key as Y_1 and commit between lines 4 and 5, and Y_1 can still commit successfully, inserting a duplicate key.

As Moss [113] suggests, the use of an open-nested commit mechanism requires the programmer to reason about the program at multiple levels of abstraction, and that the use of open-nested commit mechanism ought to be incorporated with an ***open-nesting methodology***, in which if Y is open-nested inside of X, X should not care about the memory operations performed by Y when checking for conflicts. That is, the programmer considers Y's internal memory operations to be at a "lower level" than X. Thus, instead of detecting conflicts at the memory level, X should acquire an ***abstract lock*** based on the high-level operation that Y represents, so as to allow the TM system to perform

[2] The open-nested mechanism proposed in [106] suggests that if X has previously accessed any location later written by Y, X receives the updated value when Y commits. Alternative treatments to the parent transaction's read and write sets for handling this scenario have been suggested in [114] and [113]. Since Moss [113] also suggests adopting the same scheme as in [106], we will go by the scheme as in [106].

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 [113] illustrates the use of open nesting with an application that employs a B-tree. Ni et al. [117] describe a software TM system that supports the open-nesting methodology.

Unfortunately, a gap exists between between the proposed high-level programming methodology of open nesting [113, 117] and the memory-level open-nested commit mechanism [106, 114]. Given that the TM system has no knowledge of discerning different levels of abstraction, the burden falls on the programmer to carefully reason through the memory-level semantics of the program to figure out exactly which nonserializable schedules are allowed in order to apply the methodology correctly. Nevertheless, as shown by Agrawal et al. [5], an unconstrained use of the open-nested commit mechanism can lead to anomalous program behaviors that are tricky to reason about.

One potential reason for the apparent complexity of open nesting is that the mechanism and the methodology make different assumptions about memory. Consider a transaction Y 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 cannot distinguish between these two accesses, and will commit both in an open-nested manner, leading to anomalous behavior.

Researchers *have* demonstrated specific examples [25, 117] 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.

This chapter describes the ***ownership-aware TM system***, or the ***OAT system*** for short, which bridges the gap between memory-level mechanisms for open nesting and the high-level view by explicitly integrating the notions of "transactional modules" and "ownership" into the TM system. The OAT system allows the programmer to apply the methodology of open nesting in a more structured fashion, expressing the levels of abstraction explicitly to allow the underlying runtime to behave in a way that more closely reflects the programmer's intent. Specifically, the programmer uses ***transactional modules***, or ***Xmodules*** for short, to specify levels of abstraction, and expresses ownership of data for Xmodules using parametric ownership types [22]. The OAT system employs an ***ownership-aware commit mechanism*** that is a hybrid between an open-nested and a closed-nested commit. When a transaction X commits, access to a memory location ℓ is committed globally if ℓ belongs to the same Xmodule as X; otherwise, the access to ℓ is propagated to X's parent transaction. Unlike an ordinary open-nested commit, the ownership-aware commit treats memory locations differently depending on which Xmodule owns the location. The ownership-aware commit is still a mechanism, however, and programmers must still use it in combination with abstract locks and compensating actions to implement the full open-nesting methodology.

Besides the ownership-aware commits, another distinct feature of the OAT system is that it imposes a structure on the program using the ownership types, thereby allowing the compiler and runtime to enforce properties needed to provide provable guarantees of "safety" to the programmer. Using the OAT system, the programmer is provided with a concrete set of guidelines for sharing of data and interactions between Xmodules. This chapter explains these guidelines, describes how the Xmodules and ownership can be specified in a Java-like language and proposes a type system that enforces most of the above-mentioned guidelines in the programs written using this language extension. Furthermore, this chapter presents an operational model for the ownership-aware transactions, referred to as the ***OAT model***, with which the chapter shows the following theorems. First, 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 [136]. Second, under certain restricted conditions, a computation executing under the OAT model cannot enter a semantic deadlock. Finally, the ownership-aware commit is the same as open-nested commit if no Xmodule ever accesses data belonging to other Xmodules. Thus, one corollary of our theorem is that open-nested transactions are serializable when Xmodules do not share data. This observation explains why researchers [25, 117] have found it natural to use open-nested transactions in the absence of sharing, in spite of the apparent semantic pitfalls.

Throughout this chapter, we shall distinguish between the variations of nested transactions as follows. When I refer to a nested transaction X in the OAT system which employs the ownership-aware commit mechanism, I say that X is ***safe nested***. When I refer to a nested transaction X in an ordinary TM that employs the open-nested commit mechanism, I say that X is ***open nested***. One should not confuse the term open-nested commit with the term open-nesting methodology. The open-nesting methodology includes the use of abstract locks and compensating actions, which can and should be incorporated with both safe-nested and open-nested commit mechanisms.

The rest of this chapter is organized as follows. Section 6.1 presents an overview of ownership-aware transactions and highlight key features using an example application. Section 6.2 describes language constructs for specifying Xmodules and ownership. Section 1.3 describes the OAT model in detail, and Section 6.4 gives a formal definition of serializability by modules and shows that the OAT model guarantees this definition. Section 6.5 provides conditions under which the OAT model does not exhibit semantic deadlocks. Section 6.6 discusses related work on improving the use of open-nesting. Finally, Section 6.7 provides concluding remarks.

6.1 Ownership-Aware Transactions

This section gives an overview of the ownership-aware transactions. To motivate the need for the concept of ownership in TM, this section presents an example application which may benefit from the open-nesting methodology. Illustrating using the application example, this section introduces the notion of an Xmodule and informally explains the programming guidelines when using Xmodules. This section as well highlights some of the key differences between ownership-aware TM and a TM with open-nested commit mechanism. This section serves to provide the concept of ownership-aware TM in a intuitive but informal way; we defer the formal definitions until later sections.

The book application

We shall use an example application, referred to as the book application, to illustrate the concept of ownership-aware transactions. This book application is similar to the one described by Moss [113], but it includes data sharing between nested transactions and their parents, and contains more than two levels of nesting.

Since the open-nesting methodology is designed for programs that have multiple levels of abstraction, the book application is a modular application. The book application is designed to concurrently access 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 or 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 book 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 book application is able to request a checkpoint operation which flushes the buffer to disk.

The book application can be naturally decomposed into five natural modules — the user application (`UserApp`), the database (`DB`), the binary search tree (`BST`), the hashmap (`Hashmap`), 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 the user application and the database may call methods from the `Logger` module.

Using open-nested transactions, the modules can produce 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 generates transaction X, calls an insert method in the `DB` module and passes the `Book` object b to be inserted. This insert method generates an vanilla open-nested transaction Y. Suppose Y writes to some field of the book b, which corresponds to memory location ℓ_1, and also writes some internal database metadata, which corresponds to memory 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`, i.e., 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.[3] To increase concurrency, it is desirable for Y, generated by the method from `DB`, to commit changes to its own internal data; it is not desirable, however, for Y to commit the data that `UserApp` cares about.

The notion of ***ownership of data*** can help enforcing this kind of restriction: if the TM system is aware of the fact that the book object "belongs" to the `UserApp`, it can decide not to commit `DB`'s change to the book object globally. For this purpose, the OAT system incorporates the notion of data ownership and transactional modules, or Xmodules. When a programmer explicitly defines Xmodules and specifies the ownership of data, the OAT system can make the correct judgment about which data to commit globally.

Xmodules and the ownership-aware commit mechanism

The OAT system requires that programs be organized into Xmodules. Intuitively, an Xmodule M is 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 Xmodules. During program execution, a call to a method from an Xmodule M generates a transaction instance, say X. If this method in turn calls another method from an Xmodule N, N generates an additional transaction Y, safe nested inside X, but only if $M \neq N$. Therefore, defining an Xmodule automatically specifies safe-nested transactions.

In the OAT system, every memory location is owned by exactly one Xmodule. If a memory location ℓ is in a transaction X's read or write set, the ownership-aware commit of a transaction X commits this access globally only if X is generated by the same Xmodule that owns ℓ; in this case, we say that X is ***responsible*** for that access to ℓ. Otherwise, the read or write to ℓ is propagated up

[3] Abstract locks [113] alone do not address this problem. Abstract locks are meant to disallow other transactions from noticing that the book was inserted into the `DB`, but they do not protect the individual fields of the book object itself.

to the read or write set of X's parent transaction; that is, the TM system behaves as though X was a closed-nested transaction with respect to location ℓ.

In order to guarantee that ownership-aware transactions behave "nicely," the OAT system 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 properties might not be satisfied.

One way of restricting Xmodules is to allow a transaction to access only objects that belongs to its own Xmodule. This condition might severely restrict the expressiveness of the program, however, since it does not allow an Xmodule to pass an object that it owns as a parameter to a method that belongs a different Xmodule. The OAT system is able to impose a weaker restriction on the interactions between Xmodules and at the same time guarantee these desirable properties.

Rules for Xmodules

The OAT system employs 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 the OAT system, Xmodules are arranged as a ***module tree***, denoted as $\mathcal{D}$. In $\mathcal{D}$, an Xmodule N is a child of M if N is "encapsulated by" M. The root of $\mathcal{D}$ is a special Xmodule called `world`. Each Xmodule is assigned an `xid` by visiting the nodes of $\mathcal{D}$ in a pre-order traversal, and assigning `xid`s in increasing order, starting with `xid(world)` $= 0$. Thus, `world` has the minimum `xid`, and "lower-level" Xmodules have larger `xid` numbers.

Definition 6.1 *The OAT system imposes two rules on Xmodules based on the module tree:*

1. **Rule 1***: A method of an Xmodule M can access a memory location ℓ directly only if ℓ is owned by either M or an ancestor of M in the module tree. This rule states that an ancestor Xmodule N of M may pass data down to a method belonging to M, but a transaction from module M cannot directly access any "lower-level" memory.*
2. **Rule 2***: A method from M can call a method from N only if N is the child of some ancestor of M, and that* $\mathtt{xid}(N) > \mathtt{xid}(M)$ *(i.e., if N is "to the right" of M in the module tree). This rule states that an Xmodule can call methods of some, but not all, lower-level Xmodules.*[4]

The intuition behind these rules is as follows. Xmodules have methods to provide services to other higher-level Xmodules, and Xmodules maintain their own data in order to provide these services. Therefore, a higher-level Xmodule can pass its data to a lower-level Xmodule and ask for services. A higher-level Xmodule should not directly access the internal data belonging to a lower-level Xmodule.

If Xmodules satisfy Rules 1 and 2, the ownership-aware transactions are well-defined — some transaction is always responsible for every memory access (proved in Section 1.3). In addition, these rules and the ownership-aware commit mechanism guarantee that transactions satisfy the property of serializability by modules (proved in Section 6.4).

One potential limitation of ownership-aware TM is that 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 M can call Xmodule N, then conceptually M is at a higher level than N

[4] An Xmodule can, in fact, call methods within its own Xmodule or from its ancestor Xmodules, but these calls are modeled differently. We shall come back to visit these cases at the end of this section.

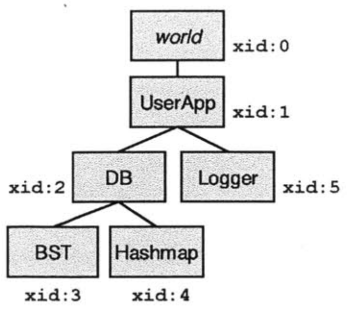


Figure 6-3: A module tree $\mathcal{D}$ for the program described in Section 6.1. The `xid`'s are assigned according to a pre-order traversal, numbering Xmodules in increasing order, starting with xid(world) = 0.

(i.e., $\texttt{xid}(M) < \texttt{xid}(N)$), and thus N cannot call M. If two components of the program call each other, then, conceptually, neither of these components is at a higher level than the other, and the OAT system requires that these two components be combined into one Xmodule.

Xmodules in the book application

Consider a Java implementation of the book application described earlier. The book application may contain 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`.

Using ownership-aware transactions, not all of a program's classes are meant to be Xmodules; some classes only wrap data. In the book example, one can identify 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.

Then, one can organize the Xmodules of the book application into the module tree shown in Figure 6-3. `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 6.1 for the example. By Rule 1, all of `DB`, `BST`, `Hashmap`, and `Logger` can directly access data owned by `UserApp`, but `UserApp` cannot directly access data owned by any of the other Xmodules. This rule corresponds to standard software-engineering rules for abstraction; the "high-level" Xmodule `UserApp` should be able to pass its data down, allowing lower-level Xmodules to access that data directly, but `UserApp` itself should not be able to directly access data owned by lower-level Xmodules. By Rule 2, `UserApp` may invoke methods from `DB`, `DB` may invoke methods from `BST` and `Hashmap`, and every other Xmodule may invoke methods from `Logger`. That is, Rule 2 allows all the operations required by the book application. As expected, `UserApp` can call the `insert` and `search` methods from `DB` and can even pass its data to `DB` for insertion. More importantly, notice the relationship between `BST` and `Logger` — `BST` can call methods from `Logger`, but `BST` cannot pass data it owns directly into `Logger`. `BST` can, however, pass data owned by the `UserApp` to `Logger`, as required by the book application.

Advantage of ownership-aware transactions

One of the major problems with ordinary open-nested commit is that some transactions can see inconsistent data. For instance, consider a transaction Y 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 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 (as written by X's inner transaction Y). The programmer may see this behavior as counterintuitive.

Now consider the same scenario for ownership-aware transactions. Without loss of generality, assume that X is generated by a method of Xmodule M and Y is generated by a method of Xmodule N. There are two cases to consider:

- **Case 1:** N owns ℓ. By Rule 2 in Definition 6.1, we know that $\texttt{xid}(M) < \texttt{xid}(N)$. Since by Rule 1 in Definition 6.1, no transaction from a higher-level module can access data owned by a lower-level module, X cannot access ℓ. Thus, the problem does not arise.
- **Case 2:** N does not own ℓ. In this case, the ownership-aware commit of Y will not commit the changes to ℓ globally, and ℓ will be propagated to X's write set. Hence, if Z tries to access ℓ before X commits, the OAT system will detect a conflict. Therefore, X cannot see an inconsistent value for ℓ.[5]

To make the scenario more concrete, think of the book application when a method from `UserApp` A calls the insert method from `DB` to insert book b. The method A generates a transaction X, which calls the insert method, which generates a transaction Y, safe nested inside X. When Y commits, it commits the data owned by `DB`, thereby increasing the concurrency; other transactions may now access data belonging to `DB` without generating conflicts with X. Y does not commit the changes made to the book b (if any), however. Thus, no other parallel transaction Z can modify b before X commits, causing X to see inconsistent state.

Callbacks

At first glance, it appears that the OAT system prohibits callbacks, where an Xmodule M is not allowed to call another transactional method in the same Xmodule M or provided by M's proper ancestor, which seems restrictive. On the contrary, the OAT system does allow some forms of callbacks, which are simply modeled differently.

More precisely, if a method X from Xmodule M calls another method Y provided by an ancestor Xmodule N, this call does not generate a new safe-nested transaction instance. Instead, Y is subsumed in X using closed nesting. Recall that Rule 1 in Definition 6.1 allows a method from a Xmodule to directly access data belonging to the same Xmodule or to any of the Xmodule's ancestors. Thus, we can treat any data access by the closed-nested transaction from Y as being directly accessed by X, provided that Y and any calls made by Y access only memory belonging to N or N's ancestors. Henceforth, we refer to such method Y as a ***proper callback*** method of Xmodule N, where Y's nested calls are themselves proper callback methods belonging to Xmodules which are ancestors of N. The formal model for ownership-aware transactions described in Section 6.3

[5] For simplicity, I 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.

assume that the computation contains only proper callbacks and models the these callbacks as direct memory accesses, allowing us to ignore callbacks in the formal definitions. The OAT type system does not enforce that the computation practice proper-callback discipline. Nevertheless, the proper-callback discipline can be enforced dynamically.

Closed-nested transactions

Using the OAT system, every method call that crosses an Xmodule boundary automatically generates a safe-nested transaction. The OAT system can effectively provide closed-nested transactions, however, with appropriate specifications of ownership. If an Xmodule M owns no memory, but only operates on memory belonging to its proper ancestors, then transactions of M will effectively be closed-nested. In the extreme case, if the programmer specifies that all memory is owned by the `world` Xmodule, then all changes in any transaction's read and write sets are propagated upwards; thus all ownership-aware commits behave exactly as closed-nested commits.

6.2 Ownership Types for Xmodules

When using ownership-aware transactions, the Xmodules and data ownership in a program must be specified, for two reasons. First, the ownership-aware commit mechanism depends on these concepts. Second, we can guarantee some notion of serializability only if a program has Xmodules which conform to the rules in Definition 6.1. This section describes the language constructs for specifying Xmodules and ownership in a Java-like language and its corresponding type system, referred to as the ***OAT type system***, which statically enforces some of the restrictions described in Definition 6.1. The OAT type system extends the type system for checking parametric ownership types due to Boyapati, Liskov, and Shrira [22], henceforth referred to as the ***BLS type system***. This section first reviews the BLS type system, then describes how the OAT type system extends the BLS type system in order to enforce most of the rules described in Definition 6.1. Lastly, this section discusses the restrictions required by Definition 6.1 which the OAT type system does not enforce statically and how these restrictions may be enforced dynamically.

The BLS type system

The BLS type system [22] provides a mechanism for specifying ownership of objects and enforces certain properties, as stated in the following lemma.

Lemma 6.2 *The BLS type system enforces the following properties:*

1. *Every object has a unique owner.*
2. *The owner can be either another object, or* `world`*.*
3. *The ownership relation forms an* ***ownership tree*** *(of objects) rooted at* `world`*.*
4. *The owner of an object does not change over time.*
5. *An object a can access another object b directly only if b's owner is either a, or one of a's proper ancestors in the ownership tree.*

The BLS type system requires ownership annotations to class definitions and type declarations to guarantee properties stated in Lemma 6.2. Every class type T_1 has a set of associated ownership tags, denoted $T_1\langle f_1, f_2, \ldots f_n\rangle$. 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 T_1 is instantiated. By parameterizing class and method declarations with ownership tags, the BLS type system permits owner polymorphism. Thus, one can define a class type once, but instantiate multiple instances of that class with different owners in different parts of the program.

The BLS type system enforces the properties stated in Lemma 6.2 with the following checks:

1. Within the class definition of type T_1, only the tags $\{f_1, f_2, \ldots f_n\} \cup \{\texttt{this}, \texttt{world}\}$ are visible. The `this` ownership tag represents the object itself.
2. A variable v_2 with type $T_2\langle f_2, \ldots \rangle$ can be assigned to a variable v_1 with type $T_1\langle f_1, \ldots \rangle$ if and only if T_2 is a subtype of T_1 and $f_1 = f_2$.
3. If an object v's tags are instantiated to be $o_1, o_2, \ldots o_n$ when v 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).

Boyapati et al. [22] show that these type checks guarantee the properties of Lemma 6.2.

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 T_1 has formal tags $\langle f_1, f_2, f_3 \rangle$, and inside $T1$'s definition, some type T_2 object is instantiated with ownership tags $\langle f_2, f_3 \rangle$. The type system cannot determine whether or not $f_2 \preceq f_3$. To resolve this ambiguity, the programmer must specify `where` $(f_2$ `<=` $f_3)$ at the class declaration of type T_1. When an instance of type T_2 object is instantiated, the type system then checks that the `where` clause is satisfied.

The OAT type system

The ownership tree described by Boyapati et al. [22] exhibits some of the same properties as the module tree described in Section 6.1. Nevertheless, the BLS type system does not enforce two major requirements needed by the OAT system:

- In the BLS type system, any object can own other objects. The OAT system, however, requires that only Xmodules own other objects.
- In the BLS type system, an object can call any of its ancestors' siblings. Rule 2 in Definition 6.1, however, dictate that an Xmodule can only call its ancestor's siblings to the right.

Thus, the OAT type system extends the BLS type system to handle these additional requirements.

Handling the first requirement is straightforward. The OAT type system explicitly distinguishes objects and Xmodules by requiring that an Xmodule extend from a special `Xmodule` class. The OAT type system only allows classes that directly extend `Xmodule` to use `this` as an ownership tag. This restriction creates a ownership tree where all the internal nodes are Xmodules objects and all leaves are non-Xmodule objects. If we ignore the ordering requirement on the children of an Xmodule, the module tree described in Section 6.1 is essentially the ownership tree with all non-Xmodule objects removed.

The second requirement involves more complexity to enforce. First, the OAT type system extends each owner instance o to have two fields: ***name***, represented as $o.name$, and ***index***, represented as $o.index$. The name field is conceptually the same as an ownership instance in the BLS type system. The index field is added to allow 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 just `this`. Similarly, one can use `world[i]` as an ownership tag. Indices enable the OAT type system to infer an ordering between two sibling

Xmodules, where the Xmodule initiated with owner `this[i]` is treated as appearing to the left of the Xmodule initiated with owner `this[i+1]` in the module tree.

Finally, for technical reasons, the OAT type system prohibits all Xmodules from declaring fields that are primitive types. If the OAT type system had allowed an Xmodule M to have fields with primitive types, these fields would be owned by M's parent. Since this property seems counter-intuitive, the OAT type system opted to disallow fields with primitive-types for Xmodules.

In summary, the OAT type system performs these checks:

1. Within the class definition of type T_1, only the tags $\{f_1, f_2, \ldots f_n\} \cup \{$`this`$,$`world`$\}$ are visible.
2. A variable v_2 with type $T_2\langle f_2, \ldots\rangle$ can be assigned to a variable v_1 with type $T_1\langle f_1, \ldots\rangle$ if and only if $T_1 = T_2$, and all the formals are initialized to the same owners with the same indices, if indices are specified.
3. A type $T\langle o_1, o_2, \ldots o_n\rangle$ 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.[6]
4. The ownership tag `this` can only be used within the definition of a class that directly extends `Xmodule`.
5. Xmodule objects cannot have fields with primitive types.

The first three checks are analogous to the checks in the BLS 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 OAT 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 Appendix B.

The book application using the OAT type system

Figure 6-4 illustrates how one can specify Xmodules and ownership for the book application described in Section 6.1 using the OAT system. The programmer specifies an Xmodule by creating a class which extends from a special `Xmodule` class. The `DB` class has three formal owner tags — `dbOwner` which is the owner of the `DB` Xmodule instance (`db`), `logO` which is the owner of the `Logger` Xmodule instance used by the `DB` Xmodule (`logger`, and `dataO` which is the owner of the user data being stored in the database. When an instance of `UserApp` initializes Xmodules that it employs in lines 6–7, it declares itself as the owner of the `Logger` Xmodule instance (`logger`), `DB` Xmodule instance (`db`), and the user data being passed into `db`. The indices on `this` indicate the ordering of Xmodules in the module tree, i.e., the user data is lower level than `Logger`, and `Logger` is lower level than `DB`. lines 16–18 illustrate how the `DB` class can initialize the Xmodules that it employs and propagate its formal owner tags, such as `logO` and `dataO`, down the module tree .

In order for this code to type check, the `DB` class must declare `logO < dataO` using the `where` clause in line 15, otherwise the type check would fail at line 16, due to ambiguity of their ordering in the module tree. The `where` clause in line 15 is checked whenever an instance of `DB` is created, i.e. at line 7.

The OAT type system's guarantees

The following lemma about the OAT type system can be proved in a reasonably straightforward manner using Lemma 6.2.

[6] In the ownership tree, for any Xmodule M, the OAT type system implicitly assigns non-Xmodule children of M higher indices than the Xmodule children of M, unless the user specifies otherwise.

```
public class UserApp<appO> extends Xmodule<appO> {
   private Logger<this[1], this[2]> logger;
   private DB<this[0], this[1], this[2]> db;

   public UserApp() {
      logger = new Logger<this[1], this[2]>();
      db = new DB<this[0], this[1], this[2]>(logger);
   }

   // rest of the class definition
    ...
}

public class DB<dbO, logO, dataO>
extends Xmodule<dbO> where (logO < dataO) {
   private Logger<logO, dataO> logger;
   private BST<this[0], logO, dataO> bst;
   private Hashmap<this[1], logO, dataO> hashmap;

   public DB(Logger<logO, dataO> logger) {
      this.logger = logger;
      // rest of the constructor
       ...
   }

   // rest of the class definition
    ...
}
```

Figure 6-4: Specifying Xmodules and ownership for the book application described in Section 6.1.

Lemma 6.3 *The OAT type system guarantees the following properties.*

1. *An Xmodule M can access a (non-Xmodule) object b with ownership tag o_b only if $M \preceq o_b.name$.*
2. *An Xmodule M can call a method in another Xmodule N with owner o_N only if one of the following is true:*
   - *(a) $M = o_N.name$ (i.e. M owns N);*
   - *(b) The least common ancestor of M and N in the module tree is $o_N.name$; or*
   - *(c) $N \succeq M$ (i.e. N is an ancestor of M).*

Lemma 6.3 does not, however, guarantee all the properties that the OAT system requires from Xmodules described in Definition 6.1. In particular, Lemma 6.3 does not consider any ordering of sibling Xmodules. The OAT type system can, however, provide stronger guarantees for a program that satisfies the following properties:

- ***unique owner indices***: For all Xmodules M, all children of M in the module tree are instantiated with ownership tags with unique indices that can be statically determined.
- ***localized use of the `world` ownership tag***: the `world` ownership tag is only used to instantiate owners inside the function `main`, or some top-level function that serves as an entry point to the user program that is executed only once.

These properties allow the OAT type system to statically determine, with local checking only, the ordering among children of a given Xmodule for all Xmodules, including `world`, thereby assigning the appropriate `xid` to every Xmodule in the modules tree, as described in Section 6.1. Then, the following result holds:

Theorem 6.4 *In the execution of a program with unique owner indices and localized use of* `world` *ownership tag, consider two Xmodules M and N. Let L be the least common ancestor Xmodule of M and N, and let o_N be the ownership tag that N is instantiated with. If $L = o_N.name$, then M can call a method in N only if* $\texttt{xid}(M) < \texttt{xid}(N)$.

PROOF. We prove (by contradiction) that if $L = o_N.name$, and $\texttt{xid}(M) > \texttt{xid}(N)$, then M cannot have a formal tag with value o_N. That means, it cannot declare a type with owner tag o_N and thus cannot access N.

Since $L = o_N.name$, we know that L is N's parent in the module tree. Given L is the least common ancestor of M and N, we know that Q exists that is N's sibling. Let o_Q be what the Q's ownership tag is instantiated with. Since N and Q have the same parent (i.e. L) in the module tree, we have $o_N.name = o_Q.name = L$. Since $\texttt{xid}(M) > \texttt{xid}(N)$, M is to the right of N in the ownership tree. Therefore, Q, which is an ancestor of M, is to the right of N in the ownership tree. Therefore, assuming the program satisfies the property of unique ownership indices, we have $o_Q.index > o_N.index$.

Assume for the purpose of contradiction that M does have o_N as one of its tags. Using Lemma 6.2, one can show that the only way for M to receive tag o_N is if Q also has a formal tag with value o_N. Thus, Q's first formal owner tag has value o_Q and another one of its formals has value o_N.

Consider the chain of Xmodule instantiations $P_k, \ldots, P_0$, where P_i instantiates P_{i-1} ending at $P_0 = Q$, and the class type of each P_i has formal ownership tags of $\langle f_1^i, f_2^i, \ldots \rangle$. P_1 must have instantiated $P_0 = Q$ with values $f_1^0 = o_Q$, and some other formal, without loss of generality say, the second formal $f_2^0 = o_N$. (We must have $f_1^0 = o_Q$, since o_Q is the owner of Q; without loss of generality, we can assign $f_2^0 = o_N$, since the OAT type system does not care about the ordering of formal tags after the first one.)

Since $o_N.name = o_Q.name = L$, assuming $L \neq \texttt{world}$, this chain of instantiations must lead back to L, since that is the only Xmodule that can create ownership tags with values o_N and o_Q in its class definition using the keyword `this`. On the other hand, if $L = \texttt{world}$, assuming the program satisfies the property of localized use of the `world` ownership tag, both o_N and o_Q must be created within the `main` function (or an entry-point function with a single execution) using the `world` keyword. Without loss of generality, we can assume that function execution is part of P_k. Then, for each instantiation P_i for $1 \leq i < k$, the following must be true.

- P_i must have some formals f_a^i and f_b^i, with values o_Q and o_N, respectively, and P_i must pass these formals into the instantiation of P_{i-1}.
- The class definition of P_i must specify the constraint $f_a^i < f_b^i$ on its formal tags explicitly through a `where` clause declaring that $f_a^i < f_b^i$.[7]

The first condition must hold to allow both o_N and o_Q to be passed down to $P_0 = Q$. The second condition is true for the Xmodules in the chain of instantiations 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, P_{k-1} is instantiated by Xmodule $P_k = L$ (or if $L = \texttt{world}$, instantiated within the function that contains the localized use of the `world` tag). In the instantiation of P_{k-1} in P_k, P_k must instantiate P_{k-1}'s formal f_a^{k-1} with value o_Q by using $\texttt{this}[x]$ (or $\texttt{world}[x]$). Similarly, P_k must instantiate P_{k-1}'s formal f_b^{k-1} with value o_N by using $\texttt{this}[y]$ (or $\texttt{world}[y]$). Assuming the instantiation in P_k type checks, we must have $x < y$, which contradicts our original assumption that $o_Q.index >$

[7] Even though the constraint $f_a^i < f_b^i$ could be implicitly specified by P_i having f_a^i as the first ownership tag, the program would no longer satisfy the unique owner indices property if that were the case.

o_N.*index* however, since if o_Q.*index* $>$ o_N.*index*, the program should not type check. Therefore, we must have o_Q.*index* $<$ o_N.*index*. □

Theorem 6.4 only modifies the Condition 2b of Lemma 6.3. Therefore, Lemma 6.3 along with Theorem 6.4 imposes restrictions on every Xmodule M which are only slightly weaker than the restrictions required by Definition 6.1. Condition 1 in Lemma 6.3 corresponds to Rule 1 of Definition 6.1. Conditions 2a and 2b are the cases permitted by Rule 2. Condition 2c, however, corresponds to the special case of callbacks or calling a method from the same Xmodule, which is not permitted by Definition 6.1. This case is modeled differently, as we explained in Section 6.1.

The OAT type system is a best-effort type system to check for the restrictions required by Definition 6.1. The OAT type system cannot fully guarantee, however, that a type-checked program does not violate Definition 6.1. Specifically, the OAT type system does not detect the following violations statically. First, if the program does not have unique owner indices, then L may instantiate both M and N with the same index. Then, by Lemma 6.3, M and N, can call each other's methods, and we can get cyclic dependencies between Xmodules.[8] Second, the program may perform improper callbacks. Say a method from M calls back to method B from L. An improper callback B can call a method of N, even though M is to the right of N. Finally, if the program does not satisfy the property of localized use of the `world` tags, M can obtain access to another Xmodule N which belongs to the `world` and to the left of M. In these cases, the OAT type system allows a program with cyclic dependency between Xmodules to pass the type checks, which is not allowed by Definition 6.1.

While the OAT type system may strictly enforce the unique indices and localized use of `world` properties, it may be overly restrictive. Instead, it may be better to employ dynamic checks and have the runtime system report an error when an execution violates the rules described in Definition 6.1. 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 there are no cyclic dependencies among Xmodules and that the execution contains only proper callbacks.

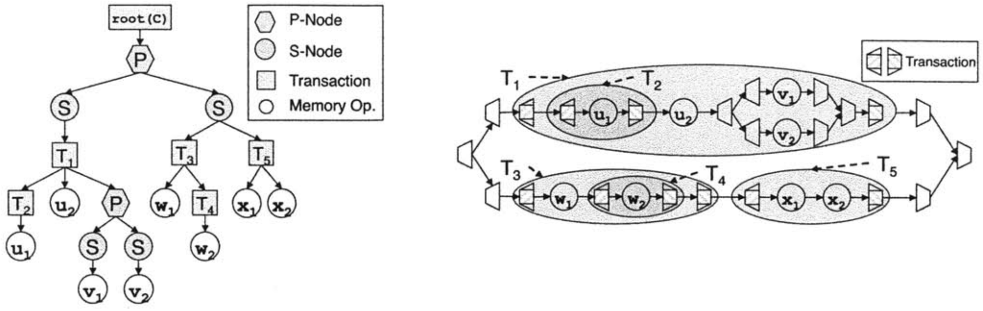
6.3 The OAT Model

The OAT model models the behavior of the OAT system as it executes a program with ownership-aware transactions. To model a program execution with ownership-aware transactions, this section extends the transactional computation framework due to Agrawal, Leiserson, and Sukha [5] to incorporate the concepts of Xmodules and ownership of data, and formally defines the structure of transactional programs with Xmodules. This section then restates the rules for Xmodules from Definition 6.1 formally in the extended framework, which guarantees certain properties used by the OAT model. Finally, this section describes the main component of the OAT model, an operational semantics for the OAT runtime system, which dynamically constructs and traverses a "computation tree" as it executes instructions generated by the program. The operational semantics described in this section is not intended to describe an actual implementation, although these semantics can be used to guide an implementation.

Transactional computations

In the framework of Agrawal et at. [5], the execution of a program is modeled using a "computation tree" C that summarizes the information about both the control structure of a program and the

[8] Since all non-Xmodule objects are implicitly assigned higher indices than their Xmodule siblings, these non-Xmodule objects cannot introduce cyclic dependencies between Xmodules.



(a) **(b)**

Figure 6-5: A sample **(a)** computation tree $\mathcal{C}$ and **(b)** its corresponding dag $G(\mathcal{C})$.

nesting structure of transactions, and an "observer function" Φ which characterizes the behavior of memory operations. A program execution is assumed to generate a ***trace*** $(\mathcal{C}, \Phi)$.

A computation tree $\mathcal{C}$ is defined as an ordered tree with two types of nodes: ***memory-operation nodes*** $\texttt{memOps}(\mathcal{C})$ as leaves and ***control nodes*** $\texttt{spNodes}(\mathcal{C})$ as internal nodes. A memory operation v either reads from or writes to a memory location. Control nodes are either S (series) or P (parallel) nodes, where the children of an ***S node*** must be executed serially, from left to right, and the children of ***P node*** can be executed in parallel. Some S nodes are labeled as transactions; define $\texttt{xactions}(\mathcal{C})$ as the set of these nodes.

Instead of specifying the value that an operation reads or writes to a memory location ℓ, the framework abstracts away the values by using an ***observer function*** Φ. For a memory operation v that accesses a memory location ℓ, the node $\Phi(v)$ is defined to be the operation that wrote the value of ℓ that v sees.

The framework defines several structural notations on the computation tree $\mathcal{C}$. Denote the ***root*** of $\mathcal{C}$ as $\texttt{root}(\mathcal{C})$. For any tree node X, let $\texttt{ances}(X)$ denote the set of all X's ancestors (including X itself) in $\mathcal{C}$, and let $\texttt{pAnces}(X)$ denote the set of *proper* ancestors of X (excluding X) by $\texttt{pAnces}(X)$. For any tree node X, define the ***transactional parent*** of X, denoted by $\texttt{xparent}(X)$, as $\texttt{parent}(X)$ if $\texttt{parent}(X) \in \texttt{xactions}(\mathcal{C})$, or $\texttt{xparent}(\texttt{parent}(X))$ if $\texttt{parent}(X) \notin \texttt{xactions}(\mathcal{C})$. Define the ***transactional ancestors*** of X as $\texttt{xAnces}(X) = \texttt{ances}(X) \cap \texttt{xactions}(\mathcal{C})$. Denote the ***least common ancestor*** of two nodes $X_1, X_2 \in C$ by $\texttt{LCA}(X_1, X_2)$. Define $\texttt{xLCA}(X_1, X_2)$ as $Z = \texttt{LCA}(X_1, X_2)$ if $Z \in \texttt{xactions}(\mathcal{C})$, and as $\texttt{xparent}(Z)$ otherwise.

A computation can also be represented as a ***computation dag*** (directed acyclic graph). Given a tree $\mathcal{C}$, the dag $G(\mathcal{C}) = (V(\mathcal{C}), E(\mathcal{C}))$ corresponding to the tree is constructed recursively. Every internal node X in the tree appears as two vertices in the dag. Between these two vertices, the children of X are connected in series if X is an S node, and are connected in parallel if X is a P node. Figure 6-5 show a computation tree and its corresponding computation dag.

Classical theories on serializability refer to a particular execution order for a program as a ***history*** [121]. In this framework, a history corresponds to a topological sort $\mathcal{S}$ of the computation dag $G(\mathcal{C})$, and the framework defines the transactional memory models 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 $\mathcal{S}$.

Xmodules and computation tree

Now we shall see how to extend the framework to model ownership-aware transactions. Formally, a trace generated by a program is organized into a set $\mathcal{N}$ of Xmodules. Each Xmodule $M \in \mathcal{N}$ has some number of methods and a set of memory locations associated with it. Thus, the set of all memory locations $\mathcal{L}$ is partitioned into sets of memory owned by Xmodules. Let $\texttt{modMemory}(M) \subseteq \mathcal{L}$ denote the set of memory locations owned by M. For a location $\ell \in \texttt{modMemory}(M)$, $\texttt{owner}(\ell) = M$. When a method of Xmodule M is called by a method from a different Xmodule, a safe-nested transaction X is generated.[9] We shall use the notation $\texttt{xMod}(X) = M$ to associate the instance X with the Xmodule M and define the instances associated with M as

$$\texttt{modXactions}(M) = \{X \in \texttt{xactions}(C) \ : \ \texttt{xMod}(X) = M\}.$$

As mentioned in Section 6.1, Xmodules of a program are arranged as a 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 M in $\mathcal{D}$ as $\texttt{modParent}(M)$, the ancestors of M as $\texttt{modAnces}(M)$, and the descendants of M as $\texttt{modDesc}(M)$. The root of the computation tree is a transaction associated with the `world` Xmodule, i.e., $\texttt{xMod}(\texttt{root}(C)) = \texttt{world}$.

The module tree $\mathcal{D}$ is used to restrict the sharing of data between Xmodules and to limit the visibility of Xmodule methods according to the rules given in Definition 6.5.

Definition 6.5 (Formal Restatement of Definition 6.1) *A program with a module tree $\mathcal{D}$ should generate only traces (C, Φ) which satisfy the following rules:*

1. **Rule 1***: For any memory operation v which accesses a memory location ℓ, let* $X = \texttt{xparent}(v)$*. Then* $\texttt{owner}(\ell) \in \texttt{modAnces}(\texttt{xMod}(X))$.
2. **Rule 2***: Let* $X, Y \in \texttt{xactions}(C)$ *be transaction instances such that* $\texttt{xMod}(X) = M$ *and* $\texttt{xMod}(Y) = N$*. Then* $X = \texttt{xparent}(Y)$ *only if*$\texttt{modParent}(N) \in \texttt{modAnces}(M)$*, and*$\texttt{xid}(M) < \texttt{xid}(N)$.

As we will see later in this section, these rules guarantee certain properties of the computation tree which are essential to the ownership-aware commit mechanism.

The OAT model overview

An execution using the OAT system is modeled as a nondeterministic state machine with two components: a ***program*** and a ***runtime system***. The runtime system dynamically constructs and traverses a computation tree C as it executes instructions generated by the program. Conceptually, the OAT model maintains a set of ***ready*** nodes, denoted by $\texttt{ready}(C) \subseteq \texttt{nodes}(C)$, and at every ***time step***, the OAT model nondeterministically chooses one of these ready nodes $X \in \texttt{ready}(C)$ 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`. Equivalently for shorthand, one can that X issues an instruction.

The OAT model describes a sequential semantics — at every time step, 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. The rest of this section presents a detailed description of the OAT model, such as the state information it maintains, how it constructs and traverses the computation tree as instructions are issued, and how it handles memory operations, conflict detections, transaction commits, and transaction aborts.

[9] As explained in Section 6.1, callbacks are handled differently.

State information and notation

As the OAT model executes instructions, it dynamically constructs the computation tree $\mathcal{C}$. For each of the sets corresponding to a computation tree defined earlier in the section, the OAT model defines corresponding time-dependent versions of these sets by indexing them with an additional time argument. For instance, let the set $\texttt{nodes}^{(t)}(\mathcal{C})$ denote the set of nodes in the computation tree after t time steps have passed. These generalized time-dependent sets are monotonically increasing; that is, once an element is added to the set, it is never removed at a later time t. As a shorthand, I may omit the time argument when it is clear that we are discussing a particular fixed time t.

At any time t, each internal node $X \in \texttt{spNodes}^{(t)}(\mathcal{C})$ has a ***status*** field $\texttt{status}[X]$. These status fields change with time. If $X \in \texttt{xactions}^{(t)}(\mathcal{C})$, i.e., X is a transaction, then $\texttt{status}[X]$ can be one of COMMITTED, ABORTED, PENDING, or PENDING_ABORT (in the process of being aborted). Otherwise, $X \in \texttt{spNodes}^{(t)}(\mathcal{C}) - \texttt{xactions}^{(t)}(\mathcal{C})$ is either a P-node or a nontransactional S-node, which can either be WORKING or SYNCHED. Several abstract sets for the tree are defined based on this status field, which partition the $\texttt{spNodes}^{(t)}(\mathcal{C})$, the set of internal nodes of the computation tree:

$$\begin{aligned}
\texttt{pending}^{(t)}(\mathcal{C}) &= \left\{X \in \texttt{xactions}^{(t)}(\mathcal{C}) \;:\; \texttt{status}[X] = \texttt{PENDING}\right\}\\
\texttt{pendingAbort}^{(t)}(\mathcal{C}) &= \left\{X \in \texttt{xactions}^{(t)}(\mathcal{C}) \;:\; \texttt{status}[X] = \texttt{PENDING\_ABORT}\right\}\\
\texttt{committed}^{(t)}(\mathcal{C}) &= \left\{X \in \texttt{xactions}^{(t)}(\mathcal{C}) \;:\; \texttt{status}[X] = \texttt{COMMITTED}\right\}\\
\texttt{aborted}^{(t)}(\mathcal{C}) &= \left\{X \in \texttt{xactions}^{(t)}(\mathcal{C}) \;:\; \texttt{status}[X] = \texttt{ABORTED}\right\}\\
\texttt{working}^{(t)}(\mathcal{C}) &= \left\{X \in \texttt{spNodes}^{(t)}(\mathcal{C}) - \texttt{xactions}^{(t)}(\mathcal{C}) \;:\; \texttt{status}[X] = \texttt{WORKING}\right\}\\
\texttt{synched}^{(t)}(\mathcal{C}) &= \left\{X \in \texttt{spNodes}^{(t)}(\mathcal{C}) - \texttt{xactions}^{(t)}(\mathcal{C}) \;:\; \texttt{status}[X] = \texttt{SYNCHED}\right\}
\end{aligned}$$

A transaction is said to be ***active*** if it has status PENDING or PENDING_ABORT. That is, the set of active transactions is defined as $\texttt{activeXactions}^{(t)}(\mathcal{C}) = \texttt{pending}^{(t)}(\mathcal{C}) \cup \texttt{pendingAbort}^{(t)}(\mathcal{C})$. Similarly, the set of active nodes is defined as $\texttt{activeNodes}^{(t)}(\mathcal{C}) = \texttt{activeXactions}^{(t)}(\mathcal{C}) \cup \texttt{working}^{(t)}(\mathcal{C})$

The OAT model maintains a set of ***ready*** S-nodes, denoted as $\texttt{ready}^{(t)}(\mathcal{C})$. We will see later in this section how the nodes are inserted and removed from $\texttt{ready}^{(t)}(\mathcal{C})$ when we discuss how the OAT model construct the computation tree. For now, simply note that $\texttt{ready}^{(t)}(\mathcal{C})$, and the sets defined above which are subsets of $\texttt{activeNodes}^{(t)}(\mathcal{C})$ (i.e., $\texttt{pending}^{(t)}(\mathcal{C})$, $\texttt{pendingAbort}^{(t)}(\mathcal{C})$, and $\texttt{working}^{(t)}(\mathcal{C})$) are not monotonic, because completing nodes remove elements from these sets.

For the purposes of detecting conflicts, at any time t, for any active transaction X, i.e., $X \in \texttt{activeXactions}^{(t)}(\mathcal{C})$, the OAT model maintains a ***read set*** $\texttt{R}^{(t)}(X)$ and a ***write set*** $\texttt{W}^{(t)}(X)$ for X. The read set $\texttt{R}^{(t)}(X)$ is a set of pairs (u, ℓ), where $u \in \texttt{memOps}^{(t)}(\mathcal{C})$ is a ***memory operation*** that reads from memory location $\ell \in \mathcal{L}$. The write set $\texttt{W}^{(t)}(X)$ is defined similarly. We say that the node u satisfies the ***read predicate*** $\texttt{R}(u, \ell)$ if u reads from location ℓ. Similarly, u satisfies the ***write predicate*** $\texttt{W}(u, \ell)$ if u writes to location ℓ. The model represents the main memory as the read and write sets of $\texttt{root}(\mathcal{C})$.

The OAT model assumes that at time $t = 0$, $\texttt{R}^{(0)}(\texttt{root}(\mathcal{C}))$ and $\texttt{W}^{(0)}(\texttt{root}(\mathcal{C}))$ initially contain a pair $(\perp, \ell)$ for all locations $\ell \in \mathcal{L}$.

In addition to the basic read and write sets, the OAT model also defines ***module read set*** and

module write set for all transactions $X \in \texttt{activeXactions}^{(t)}(\mathcal{C})$. Module read set is defined as

$$\texttt{modR}(t,X) = \left\{(u,\ell) \in \texttt{R}^{(t)}(X) \;:\; \texttt{owner}(\ell) = \texttt{xMod}(X)\right\}.$$

In other words, $\texttt{modR}(t,X)$ is the subset of $\texttt{R}^{(t)}(X)$ that accesses memory owned by X's Xmodule $\texttt{xMod}(X)$. Similarly, the ***module write set*** is defined as

$$\texttt{modW}(t,X) = \left\{(u,\ell) \in \texttt{W}^{(t)}(X) \;:\; \texttt{owner}(\ell) = \texttt{xMod}(X)\right\}.$$

The OAT model maintains two invariants on $\texttt{R}^{(t)}(X)$ and $\texttt{W}^{(t)}(X)$. First, $\texttt{W}^{(t)}(X) \subseteq \texttt{R}^{(t)}(X)$ for every transaction $X \in \texttt{xactions}^{(t)}(\mathcal{C})$, i.e., a write also counts as a read. Second, $\texttt{R}^{(t)}(X)$ and $\texttt{W}^{(t)}(X)$ each contain at most one pair (u,ℓ) for any location ℓ. Thus, a shorthand $\ell \in \texttt{R}^{(t)}(X)$ is used to mean that there exists a node u such that $(u,\ell) \in \texttt{R}^{(t)}(X)$, and similarly for $\texttt{W}^{(t)}(X)$. For simplicity, the presentation also overloads the union operator: at some time t, an operation $\texttt{R}(X) = \texttt{R}(X) \cup \{(v,\ell)\}$ means to construct the set $\texttt{R}^{(t+1)}(X)$ by

$$\texttt{R}^{(t+1)}(X) = \{(v,\ell)\} \cup \left(\texttt{R}^{(t)}(X) - \left\{(u,\ell) \in \texttt{R}^{(t)}(X)\right\}\right).$$

In other words, add (v,ℓ) to $\texttt{R}(X)$, replacing any $(u,\ell) \in \texttt{R}^{(t)}(X)$ 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, a detailed description of this construction is included.

Initially, at time $t = 0$, the OAT model begins with only the root node in the tree, i.e., $\texttt{nodes}^{(0)}(\mathcal{C}) = \texttt{xactions}^{(0)}(\mathcal{C}) = \{\texttt{root}(\mathcal{C})\}$, with this root node marked as ready, i.e., $\texttt{ready}^{(0)}(\mathcal{C}) = \{\texttt{root}(\mathcal{C})\}$. 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 $\texttt{ready}(\mathcal{C})$ and adds S_1 and S_2 to $\texttt{ready}(\mathcal{C})$. If X issues an xbegin, then the runtime creates a new transaction $Y \in \texttt{xactions}(\mathcal{C})$ as a child of X, with $\texttt{status}[Y] = \texttt{PENDING}$, removes X from $\texttt{ready}(\mathcal{C})$, and adds Y to $\texttt{ready}(\mathcal{C})$.

In the OAT model, a nontransactional S-node $Z \in \texttt{ready}^{(t)}(\mathcal{C}) - \texttt{xactions}^{(t)}(\mathcal{C})$ (which must have status WORKING) completes by issuing a join instruction. The join instruction first changes $\texttt{status}[Z]$ to SYNCHED. In the tree, since $\texttt{parent}(Z)$ is always a P-node, Z has exactly one sibling. If Z is the first child of $\texttt{parent}(Z)$ to be SYNCHED, the OAT model removes Z from $\texttt{ready}(\mathcal{C})$. Otherwise, Z is the last child of $\texttt{parent}(Z)$ to be SYNCHED, and the runtime removes Z and $\texttt{parent}(Z)$ from $\texttt{ready}(\mathcal{C})$, changes the status of both Z and $\texttt{parent}(Z)$ to SYNCHED, and adds $\texttt{parent}(\texttt{parent}(Z))$ to $\texttt{ready}(\mathcal{C})$.

A transaction $X \in \texttt{ready}^{(t)}(\mathcal{C})$ can complete by issuing either an xend or xabort instruction. If $\texttt{status}[X] = \texttt{PENDING}$, then X can issue an xend to change $\texttt{status}[X]$ to COMMITTED. Otherwise, $\texttt{status}[X] = \texttt{PENDING\_ABORT}$, and X can issue an xabort to change its status to ABORTED. For both xend and xabort, the runtime removes X from $\texttt{ready}(\mathcal{C})$ and adds $\texttt{parent}(X)$ back into $\texttt{ready}(\mathcal{C})$. The xend instruction also performs an ownership-aware commit and changes read sets and write sets, which is described later when we discuss the ownership-aware commits in the OAT model.

Finally, a ready node Z can issue a read or a write instruction. If the instruction does not generate a conflict, the runtime adds a memory operation node v to $\texttt{memOps}^{(t)}(C)$, with v as a child of Z. If the instruction would create a conflict, the runtime may change the status of one PENDING transaction X to PENDING_ABORT to make progress in resolving the conflict. For shorthand, the status change of a transaction X from PENDING to PENDING_ABORT is referred to as a sigabort of X.

This construction of the tree guarantees a few properties. First, the sequence of instructions S generated by the OAT model is a valid topological sort of the computation dag $G(C)$. 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 $v \in \texttt{memOps}(C)$, the OAT model first checks whether creating v would cause a conflict, according to Definition 6.6.

Definition 6.6 *Suppose at time t, the OAT model issues a read or write instruction that potentially creates a memory operation node v. The memory operation v is said to generate a* ***memory conflict*** *if there exists a location $\ell \in L$ and an active transaction $X_u \in \texttt{activeXactions}^{(t)}(C)$ such that*

1. *$X_u \notin \texttt{xAnces}(v)$, and*
2. *either $R(v,\ell) \wedge ((u,\ell) \in \texttt{W}^{(t)}(X_u))$, or $W(v,\ell) \wedge ((u,\ell) \in \texttt{R}^{(t)}(X_u))$.*

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. The mechanism for aborts is described later in this section. Otherwise, a memory operation v that does not generate a conflict observes the value ℓ from $\texttt{R}(X)$, where X is the closest ancestor of v with ℓ in its read set (i.e., $(u,\ell) \in \texttt{R}(X)$ and that either $\Phi(v) = u$ if u is a write or $\Phi(v) = \Phi(u)$ if u is a read). In addition, v updates the read and/or write sets of its enclosing transactions, $Y = \texttt{xparent}(v)$. If v is a read, (v,ℓ) is added to $\texttt{R}(Y)$. If v is a write, (v,ℓ) is added to both $\texttt{R}(Y)$ and $\texttt{W}(Y)$.

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 $\texttt{xMod}(Y)$ 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.

Or more formally, the OAT model's commit mechanism can be described in terms of module read sets and write sets. Suppose at time t, $Y \in \texttt{xactions}^{(t)}(C)$ with $\texttt{status}[Y] = \texttt{PENDING}$ issues an xend. This xend changes read sets and write sets as follows:

$$\begin{aligned}
\mathtt{R}(\mathtt{root}(C)) &= \mathtt{R}(\mathtt{root}(C)) \cup \mathtt{modR}(Y) \\
\mathtt{R}(\mathtt{xparent}(Y)) &= \mathtt{R}(\mathtt{xparent}(Y)) \cup (\mathtt{R}(Y) - \mathtt{modR}(Y)) \\
\mathtt{W}(\mathtt{root}(C)) &= \mathtt{W}(\mathtt{root}(C)) \cup \mathtt{modW}(Y) \\
\mathtt{W}(\mathtt{xparent}(Y)) &= \mathtt{W}(\mathtt{xparent}(Y)) \cup (\mathtt{W}(Y) - \mathtt{modW}(Y))
\end{aligned}$$

Unique committer property

Definition 6.5 guarantees certain properties of the computation tree which are essential to the ownership-aware commit mechanism. Theorem 6.8 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 6.7 *Given a computation tree C, for any $T \in \mathtt{xactions}(C)$, let $S_T = \{\mathtt{xMod}(T') \ : \ T' \in \mathtt{xAnces}(T)\}$. Then* $\mathtt{modAnces}(\mathtt{xMod}(T)) \subseteq S_T$.

PROOF. Lemma 6.7 can be proven by induction on the nesting depth of transactions T in the computation tree. In the base case, the top-level transaction $T = \mathtt{root}(C)$, and $\mathtt{xMod}(\mathtt{root}(C)) = \mathtt{world}$. Thus, the lemma holds trivially.

For the inductive step, assume that $\mathtt{modAnces}(\mathtt{xMod}(T)) \subseteq S_T$ holds for any transaction T at depth d. One can show that the fact holds for any $T^* \in \mathtt{xactions}(C)$ at depth $d+1$. For any such T^*, we know that $T = \mathtt{xparent}(T^*)$ is at depth d. By Rule 2 of Definition 6.5, we have $\mathtt{modParent}(\mathtt{xMod}(T^*)) \in \mathtt{modAnces}(\mathtt{xMod}(T))$. Thus, $\mathtt{modAnces}(\mathtt{xMod}(T^*)) \subseteq \mathtt{modAnces}(\mathtt{xMod}(T)) \cup \{\mathtt{xMod}(T^*)\}$. By construction of the set S_T, we have $S_{T^*} = S_T \cup \{\mathtt{xMod}(T^*)\}$. Therefore, using the inductive hypothesis, $\mathtt{modAnces}(\mathtt{xMod}(T^*)) \subseteq S_{T^*}$. □

Theorem 6.8 *If a memory operation v accesses a memory location ℓ, then there exists a unique transaction $T^* \in \mathtt{xAnces}(v)$, such that*

1. $\mathtt{owner}(\ell) = \mathtt{xMod}(T^*)$, *and*
2. *For all transactions $X \in \mathtt{pAnces}(T^*) \cap \mathtt{xactions}(C)$, X can not directly access ℓ.*

This transaction T^* is the ***committer*** *of memory operation v, denoted* $\mathtt{committer}(v)$.

PROOF. This result follows from the properties of the module tree and computation tree stated in Definition 6.5.

Let $T = \mathtt{xparent}(v)$. First, by Definition 6.5, Rule 1, we know that $\mathtt{owner}(\ell) \in \mathtt{modAnces}(\mathtt{xMod}(T))$. By Lemma 6.7, we know that $\mathtt{modAnces}(\mathtt{xMod}(T)) \subseteq S_T$. Thus, there exists some transaction $T^* \in \mathtt{xAnces}(T)$ such that $\mathtt{owner}(\ell) = \mathtt{xMod}(T^*)$. We can use Rule 2 to show that the T^* is unique. Let X_i be the chain of ancestor transactions of T, i.e., let $X_0 = T$, and let $X_i = \mathtt{xparent}(X_{i-1})$, up until $X_k = \mathtt{root}(C)$. By Rule 2, we know that $\mathtt{xid}(\mathtt{xMod}(X_i)) < \mathtt{xid}(\mathtt{xMod}(X_{i-1}))$, meaning, the xids strictly decrease walking up the tree from T. Thus, there can only be one ancestor transaction T^* of T with $\mathtt{xid}(\mathtt{xMod}(T^*)) = \mathtt{xid}(\mathtt{owner}(\ell))$.

To check the second condition, consider any $X \in \mathtt{pAnces}(T^*) \cap \mathtt{xactions}(C)$. By Rule 1, X can access ℓ directly only if $\mathtt{owner}(\ell) \in \mathtt{modAnces}(\mathtt{xMod}(X))$ implying that $\mathtt{xid}(\mathtt{owner}(\ell)) \leq \mathtt{xid}(\mathtt{xMod}(X))$. But we know that $\mathtt{owner}(\ell) = \mathtt{xMod}(T^*)$ and $\mathtt{xid}(\mathtt{xMod}(T^*)) > \mathtt{xid}(\mathtt{xMod}(X))$, so X can never access ℓ directly. □

Intuitively, $T^* =$ `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 6.8 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 aborts

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 6.9 *The set of operations issued by X or descendants of X after* `status`$[X]$ *changes to PENDING_ABORT are called X's **abort actions**, 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 X_u and X_u's status is PENDING, then the OAT model can nondeterministically choose to abort either `xparent`(v), or X_u. In the latter case, v waits for X_u to finish aborting (i.e., change its status to ABORTED) before continuing. If X_u's status is PENDING_ABORT, then v just waits for X_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 X_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 described later in Section 6.5.

6.4 Serializability by Modules

This section shows that the OAT model guarantees ***serializability by modules***, a definition inspired by the database notion of multilevel serializability (e.g., as described in [136]). Agrawal et al. [5] provide a definition of serializability in their transaction computation framework, which is what the OAT model is based on. Their definition of serializability is too restrictive for ownership-aware transactions, however, since ownership-aware transactions, being a hybrid between closed and open nesting, allow certain kinds of program interleaving that would not be allowed under the definition of serializability. Thus instead, this section considers a less restrictive correctness condition, serializability by modules, which incorporates the notions of Xmodules and ownership-aware commits, and proves that the OAT model guarantees serializability by modules. Lastly, this section discusses the relationship between the definition of serializabiity by modules and the notion of abstract serializability for the open-nesting methodology.

Transactional computations and serializability

In the framework due to Agrawal et. al [5], serializability for a transactional computation with computation tree $\mathcal{C}$ was defined in terms of topological sorts $\mathcal{S}$ of the computation dag $G(\mathcal{C})$. Informally, a trace (C,Φ) is serializable if there exists a topological sort order S of $G(C)$ such that S is "sequentially consistent with respect to Φ", and all transactions appear contiguous in the order S. This section provides a more precise and formal definition of this concept and generalizes it to formally define serializability by modules.

Some notation is needed to formally describe serializability (and serializability by modules). Since the OAT model extends the framework due to Agrawal et al. [5], some definitions overlap and some are modified to fit the OAT model. Furthermore, same as the framework of Agrawal et al. all definitions in this section are *a posteriori*, i.e., they are defined on the computation tree after the program has finished executing.

All memory operations enclosed inside a transaction T (including those belonging to its nested transactions), i.e., $\texttt{memOps}(T)$, can be partitioned into three static "content" sets: $\texttt{cContent}(T)$, $\texttt{oContent}(T)$ and $\texttt{aContent}(T)$. For any $u \in \texttt{memOps}(T)$, the content sets are defined based on the final status of transactions in C that one visits when walking up the tree from u to T.

Definition 6.10 *For any transaction T and memory operation u, define the* ***static content sets*** $\texttt{cContent}(T)$, $\texttt{oContent}(T)$, *and* $\texttt{aContent}(T)$ *according the* $\texttt{ContentType}(u,T)$ *procedure:*

```
ContentType($u,T$)        // For any $u \in \texttt{memOps}(T)$
$X = \texttt{xparent}(u)$
while ($X \neq T$)
    if ($X$ is ABORTED)        return $u \in \texttt{aContent}(T)$
    if ($X = \texttt{committer}(u)$) return $u \in \texttt{oContent}(T)$
    $X = \texttt{xparent}(X)$
return $u \in \texttt{cContent}(T)$
```

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.

For computations with transactions, one 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 6.11 *For $u \in \texttt{memOps}(C), v \in V(C)$, let $X = \texttt{xLCA}(u,v)$. Then, u is* ***hidden from*** *v if $u \in \texttt{aContent}(X)$, denoted as uHv.*

The definition of serializability by modules requires that computations satisfy some notion of sequential consistency, generalized for the setting of TM.

Definition 6.12 *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 $X_S(v)$, is the unique $u \in \texttt{memOps}(C) \cup \{\bot\}$ that satisfies four conditions:*

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

Definition 6.13 *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 v which accesses location ℓ, is the last write u to location ℓ in the order $\mathcal{S}$, except that it skips over writes w which are hidden from (i.e., aborted with respect to) v. Intuitively, Definition 6.13 requires that there exists an order $\mathcal{S}$ explaining all the memory operations of the computation.

Finally, using this framework, ***serializability*** is defined as follows:

Definition 6.14 *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 $\forall v \in V(\mathcal{C})$, $\texttt{xbegin}(T) \leq_{\mathcal{S}} v \leq_{\mathcal{S}} \texttt{xend}(T) \implies v \in V(T)$).*

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

While this definition of serializability is the "correct definition" for flat or closed-nested transactions, it is too strong, however, for ownership-aware transactions. A TM system that enforces this definition of serializability cannot ignore lower-level memory accesses when detecting conflicts for higher-level transactions.

Instead, we consider a definition of serializability by modules which checks for correctness of one Xmodule at a time. For serializability by modules, given a trace $(\mathcal{C},\Phi)$, for each Xmodule M, transform the tree $\mathcal{C}$ into a new tree $\texttt{mTree}(\mathcal{C},M)$, referred to as the ***projection of*** $\mathcal{C}$ ***for Xmodule*** M. The projected tree $\texttt{mTree}(\mathcal{C},M)$ is constructed in such a way as to ignore memory operations of Xmodules which are lower-level than M, and also to ignore all operations which are hidden from transactions of M. For each Xmodule M, check that the transactions of M in the trace $(\texttt{mTree}(\mathcal{C},M),\Phi)$ is serializable. If the check holds for all Xmodules, then trace $(\mathcal{C},\Phi)$ is said to be serializable by modules. Definition 6.15 formalizes the construction of $\texttt{mTree}(\mathcal{C},M)$:

Definition 6.15 *For any computation tree $\mathcal{C}$, define the* ***projection of*** $\mathcal{C}$ ***for*** M*, denoted as* $\texttt{mTree}(\mathcal{C},M)$ *be the result of modifying $\mathcal{C}$ as follows:*

1. *For all memory operations $v \in \texttt{memOps}(\mathcal{C})$ with v accessing ℓ, if $\texttt{owner}(\ell) = N$ for some $\texttt{xid}(N) > \texttt{xid}(M)$, convert v into a nop.*
2. *For all transactions $T \in \texttt{modXactions}(M)$, convert all $v \in \texttt{aContent}(T)$ into nops.*

The intuition behind Step 1 of Definition 6.15 is as follows. To obtain the projected tree $\texttt{mTree}(\mathcal{C},M)$, Step 1 of the construction throws away memory operations belonging to a lower-level Xmodule N, since by Theorem 6.8, transactions of M can never directly access the same memory as those operations anyway. Step 2 of the construction ignores the content of any aborted transactions nested inside transactions of M; those transactions might access the same memory locations as operations which were not turned into nops, but those operations are aborted with respect to transactions of M.

Lemma 6.16 argues that if a trace $(\mathcal{C},\Phi)$ is sequentially consistent, then $(\texttt{mTree}(\mathcal{C},M),\Phi)$ is a valid trace; an operation v that remains in the trace never attempts to observe a value from a $\Phi(v)$ which was turned into a nop due to Definition 6.15. In addition, the transformed trace is also sequentially consistent.

Lemma 6.16 *Let (C,Φ) be any trace and S be any topological sort such that $\Phi = X_S$ (i.e., (C,Φ) is sequentially consistent). Then for any Xmodule M, the following conditions are satisfied:*

1. *If $v \in$ `memOps(mTree(`C,M`))`, then $\Phi(v) \in$ `memOps(mTree(`C,M`))`.*
2. *S is a valid sort of (`mTree(`C,M`)`$,\Phi$), with $\Phi = X_S$.*

In other words, (`mTree(`C,M`)`$,\Phi$) is a valid trace.

PROOF. Let's check Condition 1 first. In the projected tree `mTree(`C,M`)`, pick any node $v \in$ `memOps(mTree(`C,M`))` which remains. Assume for contradiction that $u = \Phi(v)$ was turned into a nop in one of Steps 1 and 2.

If u was turned into a nop in Step 1 of Definition 6.15 during the construction, then it must be that u accessed a memory location ℓ where `xid(owner(`ℓ`))` $>$ `xid(`M`)`. Since v must access the same location ℓ, v must also be converted into a nop.

If u was turned into a nop in Step 2 of Definition 6.15, then $u \in$ `aContent(`T`)` for some `xMod(`T`)` $= M$. Then one can show that either uHv, or v should have also been turned into a nop. Let $X =$ `xLCA(`u,v`)`. Since T and X are both ancestors of u, either T is a proper ancestor of X or X is an ancestor of T.

1. First, suppose T is a proper ancestor of X. Consider the path of transactions $Y_0, Y_1, \ldots Y_k$, where $Y_0 =$ `xparent(`u`)`, `xparent(`Y_i`)` $= Y_{i+1}$, and `xparent(`Y_k`)` $= T$. Since $u \in$ `aContent(`T`)`, for some Y_j for $0 \leq j \leq k$ must have `status`$[Y_j] =$ `ABORTED`. Since T is a proper ancestor of X, $X = Y_x$ for some x satisfying $0 \leq x \leq k$.
   (a) If `status`$[Y_j] =$ `ABORTED` for any j satisfying $0 \leq j < x$, then we know $u \in$ `aContent(`X`)`, and thus uHv. Since (C,Φ) is sequentially consistent and $\Phi(v) = u$, by Definition 6.12, we know $\neg uHv$, leading to a contradiction.
   (b) If Y_j is `ABORTED` for any j satisfying $x \leq j \leq k$, then `status`$[Y_j] =$ `ABORTED` implies that $v \in$ `aContent(`X`)`, and thus, v 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 $u \in$ `aContent(`T`)`, it must be that $u \in$ `aContent(`X`)`. Therefore, this case is analogous to Case 1a above.

To check Condition 2, if Φ is the transactional last writer according to S for (C,Φ), it is still the transactional last writer for (`mTree(`C,M`)`$,\Phi$) because the memory operations which are not turned into nops remain in the same relative order. Thus, Condition 2 is also satisfied. □

Note that Lemma 6.16 *depends on* the restrictions on Xmodules described in Definition 6.5. Without this structure of modules and ownership, the construction of Definition 6.15 is not guaranteed to generate a valid trace.

Finally, serializability by modules is defined as follows.

Definition 6.17 *A trace (C,Φ) is **serializable by modules** if*

1. *There exists a topological sort S such that $\Phi = X_S$, and*
2. *for all Xmodules M in $\mathcal{D}$, there exists a topological sort S_M of $C_M =$ `mTree(`C,M`)` such that:*
   (a) *S_M is a topological sort of C_M such that $\Phi = X_{S_M}$, and*
   (b) *$\forall T \in$ `modXactions(`M`)` and $\forall v \in V(C_M)$, if `xbegin(`T`)` $\leq_{S_M} v \leq_{S_M}$ `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 M, there exists a sequentially consistent order $\mathcal{S}_M$ for the trace $(\texttt{mTree}(\mathcal{C},M),\Phi)$ such that all transactions of M are contiguous in $\mathcal{S}_M$. Even though $\mathcal{S}_M$ may not be the same as $\mathcal{S}$, a computation that satisfies serializability by module has a sensible semantics, because both $\mathcal{S}_M$ and $\mathcal{S}$ are sequentially consistent with respect to Φ.

The OAT model guarantees serializability by modules

The OAT model described in Section 6.3 generates traces $(\mathcal{C},\Phi)$ that are serializable by modules, i.e., that satisfy Definition 6.17. The proof of this fact consists of two parts. The first part shows that the OAT model guarantees that a program execution is prefix-race free. The second part shows that any trace which is prefix-race free is also serializable by modules.

Before we dive into the proofs, we shall first examine how the model defines prefix-race freedom. The following definitions are taken from the framework of Agrawal et al. [5], but adapted for the OAT model with an ownership-aware commit mechanism. Notably, the OAT model uses slightly different notions of hidden (Definition 6.11) and how the content sets of transactions are defined (Definition 6.10).

Definition 6.18 *For any execution order $\mathcal{S}$, for any transaction $T \in \texttt{xactions}(\mathcal{C})$, consider any $v \notin \texttt{memOps}(T)$ such that $\texttt{xbegin}(T) <_{\mathcal{S}} v <_{\mathcal{S}} \texttt{xend}(T)$. There exists a* ***prefix race*** *between T and v if*

1. $\exists w \in \texttt{cContent}(T)$ *such that* $w <_{\mathcal{S}} v$,
2. $\neg(vHw)$, *and*
3. $(\texttt{R}(w,\ell) \wedge \texttt{W}(v,\ell)) \vee (\texttt{W}(w,\ell) \wedge \texttt{R}(v,\ell)) \vee (\texttt{W}(w,\ell) \wedge \texttt{W}(v,\ell))$.

Definition 6.19 *A trace $(\mathcal{C},\Phi)$ is* ***prefix-race free*** *iff exists a topological sort $\mathcal{S}$ of $G(\mathcal{C})$ satisfying two conditions:*

1. $\Phi = X_{\mathcal{S}}$ *($\mathcal{S}$ 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.*

The OAT model preserves certain invariants, and these invariants are used to prove that the OAT model generates only traces $(\mathcal{C},\Phi)$ which are prefix-race free. Theorem 6.20 and Lemma 6.21 state the invariants.

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 6.20 *Suppose the OAT model generates a trace $(\mathcal{C},\Phi)$ and an execution order $\mathcal{S}$. Then, $\Phi = X_{\mathcal{S}}$, 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. The details of this proof is deferred to Appendix A. □

The next lemma, Lemma 6.21, describes an invariant on read sets and write sets that the OAT model maintains. Informally, Lemma 6.21 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 6.21 *Suppose the OAT model generates a trace* (C,Φ) *with an execution order* $\mathcal{S}$. *For any transaction T, consider a memory operation* $u \in \mathtt{cContent}(T)$ *which accesses memory location* ℓ *at step* t_0. *Let* t_f *be step when* $\mathtt{xend}(T)$ *or* $\mathtt{xabort}(T)$ *happens. At any time t such that* $t_0 \leq t < t_f$ *there exists some* $T' \in \mathtt{xDesc}(T) \cap \mathtt{activeXactions}^{(t)}(C)$ *(i.e.,* T' *is an active transactional descendant of T) such that*

1. *If* $R(u,\ell)$, *then* $\ell \in \mathtt{R}^{(t)}(T')$.
2. *If* $W(u,\ell)$, *then* $\ell \in \mathtt{W}^{(t)}(T')$.

PROOF. Let $X_1, X_2, \ldots X_k$ be the chain of transactions from $\mathtt{xparent}(u)$ up to, but not including T, i.e., $X_1 = \mathtt{xparent}(u)$, $X_j = \mathtt{xparent}(X_{j-1})$, and $\mathtt{xparent}(X_k) = T$. Since we assume that $u \in \mathtt{cContent}(T)$ 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 $\mathtt{status}[X_j]$ from PENDING to COMMITTED; otherwise, we would have $u \in \mathtt{aContent}(T)$.

Also, by Theorem 6.8 and Definition 6.10, we know $\mathtt{committer}(u) \in \mathtt{xAnces}(T)$, i.e., none of the X_j's will commit location ℓ in an open-nested fashion to the world; otherwise, we would have $u \in \mathtt{oContent}(T)$.

First, suppose $R(u,\ell)$. At time t_i, when the memory operation u completes, (u,ℓ) is added to $\mathtt{R}(X_1)$. In general, at time t_j, the ownership-aware commit mechanism, as described in Section 6.3, will propagate ℓ from $\mathtt{R}(X_j)$ to $\mathtt{R}(X_{j+1})$. Therefore, for any time t in the interval $[t_{j-1}, t_j)$, we know $\ell \in \mathtt{R}^{(t)}(X_j)$, i.e., for Lemma 6.21, $T' = X_j$. Similarly, for any time t in the interval $[t_k, t_f)$, we have $\ell \in \mathtt{R}^{(t)}(T)$, 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 $\ell \in \mathtt{R}^{(t)}(T')$ and $\ell \in \mathtt{W}^{(t)}(T')$. □

Using Theorem 6.20 and Lemma 6.21, Theorem 6.22 shows that the OAT model generates traces which are prefix-race free.

Theorem 6.22 *Suppose the OAT model generates a trace* (C,Φ) *with an execution order* $\mathcal{S}$. *Then* $\mathcal{S}$ *is a prefix-race-free sort of* (C,Φ).

PROOF. For the first condition of Definition 6.19, we know by Theorem 6.20 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 $v \notin \mathtt{memOps}(T)$. Let w be the memory operation from Definition 6.18, i.e., $w \in \mathtt{cContent}(T)$, $w <_{\mathcal{S}} v <_{\mathcal{S}} \mathtt{xend}(T)$, $\neg(vHw)$, 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 $\mathtt{xend}(T)$ or $\mathtt{xabort}(T)$ 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 6.21, at time t_v, ℓ is in the read set of some active transaction $T' \in \mathtt{xDesc}(T)$. Since $v \notin \mathtt{memOps}(T)$, we know $T \notin \mathtt{xAnces}(v)$. Thus, since T' is a descendant of T, we have $T' \notin \mathtt{xAnces}(v)$. Since $T' \notin \mathtt{xAnces}(v)$, by Definition 6.6, 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. □

The next theorem shows that a trace $(\mathcal{C},\Phi)$ which is prefix-race free is also serializable by modules.

Theorem 6.23 *Any trace $(\mathcal{C},\Phi)$ which is prefix-race free is also serializable by modules.*

PROOF. First, by Definition 6.15 and Lemma 6.16, it is easy to see that a prefix-race-free sort $\mathcal{S}$ of a trace $(\mathcal{C},\Phi)$ is also a prefix-race-free sort of the trace $(\texttt{mTree}(\mathcal{C},M),\Phi)$ for any Xmodule M. Now we shall argue that for any Xmodule M, we can transform $\mathcal{S}$ into $\mathcal{S}_M$ such that all transactions in $\texttt{xactions}(M)$ appear contiguous in $\mathcal{S}_M$.

Consider a prefix-race-free sort $\mathcal{S}$ of $(\texttt{mTree}(\mathcal{C},M),\Phi)$ which has k nodes v which violate the second condition of Definition 6.17. One can construct a new order $\mathcal{S}'$ which is still a prefix-race-free sort of $(\texttt{mTree}(\mathcal{C},M),\Phi)$, but which has only $k-1$ violations.

The following procedure reduces the number of violations:

1. Of all transactions $T \in \texttt{modXactions}(M)$ such that there exists an operation v that causes a violation, i.e., $\texttt{xbegin}(T) \leq_{\mathcal{S}} v \leq_{\mathcal{S}} \texttt{xend}(T)$ 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 $(\texttt{mTree}(\mathcal{C},M),\Phi)$, one needs 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 $\texttt{xbegin}(T) <_{\mathcal{S}} w <_{\mathcal{S}} v$, one can argue that w can not access the same location ℓ, unless both w and v read from ℓ, with the following reasoning. Since the procedure chose v, which is the first memory operation that causes the violation, i.e., $\texttt{xbegin}(T) <_{\mathcal{S}} v <_{\mathcal{S}} \texttt{xend}(T)$ and $v \notin V(T)$, we know that $w \in V(T)$. Otherwise, v wouldn't be the first memory operation that causes the violation. We know by construction of $\texttt{mTree}(\mathcal{C},M)$, that $w \in \texttt{cContent}(T)$ — if $w \in \texttt{oContent}(T)$ or $w \in \texttt{aContent}(T)$, then Step 1 or 2, respectively, in Definition 6.15 would have turned w into a nop. Therefore, by Definition 6.18, 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. That is, either w does not access ℓ, or both w and v read from ℓ, and thus moving v to be before $\texttt{xbegin}(T)$ can not generate any new prefix races. Furthermore, moving v cannot change the transactional last writer for any memory operation w, 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 6.17 one at a time, we can construct a sort $\mathcal{S}_M$ which satisfies serializability by modules by eliminating all violations. □

Finally, we can show the OAT model guarantees serializability by modules by putting the previous results together.

Theorem 6.24 *Any trace $(\mathcal{C},\Phi)$ generated by the OAT model is serializable by modules.*

PROOF. By Theorem 6.22, the OAT model generates only trace $(\mathcal{C},\Phi)$ which are prefix-race free. By Theorem 6.23, any trace $(\mathcal{C},\Phi)$ which is prefix-race free is serializable by modules. □

Abstract serializability

By Theorem 6.24, the OAT model guarantees serializability by modules. As mentioned earlier in the chapter introduction, the ownership-aware commit mechanism is a part of a methodology which includes abstract locks and compensating actions. The last part of this section argues that OAT model provides enough flexibility to accommodate abstract locks and compensating actions. In addition, if a program is "properly locked and compensated," then serializability by modules guarantees "abstract serializability" used in multilevel database systems [136].

The definition of abstract serializability in [136] assumes that the program is divided into levels, and that a transaction at level i can only call a transaction at level $i+1$.[10] In addition, transactions at a particular level have predefined commutativity rules, i.e., some transactions of the same Xmodule can commute with each other and some 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 [117]: if two transactions can not commute, then they grab the same abstract lock in a conflicting manner. In the application described in Section 6.1, for instance, transactions calling `insert` and `remove` on the BST using the same key do not commute and should grab the same write lock. Although abstract locks are not explicitly modeled in the OAT model, transactions acquiring the same abstract lock can be modeled as transactions writing to a common memory location ℓ.[11] Locks associated with an Xmodule M are owned by $\texttt{modParent}(M)$. A module M is said to be ***properly locked*** if the following is true for all transactions X_1, X_2 with $\texttt{xMod}(X_1) = \texttt{xMod}(X_2) = M$: if X_1 and X_2 do not commute, then they access some $\ell \in \texttt{modMemory}(\texttt{modParent}(M))$ in a conflicting manner.

If all transactions are properly locked, then serializability by modules implies abstract serializability as defined above in the special case when the module tree is a chain (i.e., each non-leaf module has exactly one child). Let $\mathcal{S}_i$ be the sort $\mathcal{S}$ in Definition 6.17 for Xmodule M with $\texttt{xid}(M) = i$. This $\mathcal{S}_i$ corresponds to $\mathcal{Z}_i$ in the definition of abstract serializability.

In the general case for ownership-aware TM, however, by Rule 2 of Definition 6.1, a transaction at level i might call transactions from multiple levels $x > i$, not just $x = i+1$. Thus, the definition of abstract serializability must be changed 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

[10] The discussion here assumes that the level number increases as going from a higher level to a lower-level to be consistent with the numbering of `xid`. In the literature (e.g. [136]), levels typically go in the opposite direction.

[11] More complicated locks can be modeled by generalizing the definition of conflict.

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.

6.5 Deadlock Freedom

This section argues that the OAT model described in Section 6.3 can never enter a "semantic deadlock" if suitable restrictions are imposed on the memory accessed by a transaction's abort actions. In particular, an abort action generated by transaction T from $\texttt{xMod}(T)$ should read (write) from a memory location ℓ belonging to $\texttt{modAnces}(\texttt{xMod}(T))$ only if ℓ is already in $\texttt{R}(T)$ ($\texttt{W}(T)$). Under these conditions, this section shows 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 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 X and Y which have status PENDING_ABORT. If X's abort action generates a memory operation u which conflicts with Y, u will wait for Y to finish aborting (i.e., when the status of Y becomes ABORTED). Similarly, Y's abort action can generate an operation v which conflicts with X and waits for X to finish aborting. Thus, X and Y can both wait on each other, and neither transaction will ever finish aborting.

Defining semantic deadlock

Intuitively, we want to say that a TM system exhibits a semantic deadlock if it might enter a state from which it is impossible to "finish" a computation because of transaction conflicts. This section defines semantic deadlock precisely and distinguishes it from these other reasons for noncompletion, such as livelock or infinite loop.

Recall that our abstract model has two entities: the program, and a generic operational model $\mathcal{R}$ 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 $\mathcal{R}$ cannot complete the program no matter what it does. To eliminate programs which have infinite loops, we only consider ***bounded programs***.

Definition 6.25 *A program is **bounded** for an operational model $\mathcal{R}$ if any computation tree that $\mathcal{R}$ generates for that program is of a finite depth, and there exists a finite number K such that at any time t, every node $Z \in \texttt{nodes}^{(t)}(\mathcal{C})$ has at most K children with status PENDING or COMMITTED.*

Even if the program is bounded, it might still run forever if it ***livelocks***. One can use the notion of a ***schedule*** to distinguish livelocks from semantic deadlocks.

Definition 6.26 *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 $\mathcal{R}$ makes two types of nondeterministic choices. First, at any time t, $\mathcal{R}$ 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, $\mathcal{R}$ 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 $\mathcal{R}$ 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 6.27 *Consider an an operational model $\mathcal{R}$ executing a bounded computation. We say that $\mathcal{R}$ does not exhibit a* ***semantic deadlock*** *if for all finite sequences of t_0 instructions that $\mathcal{R}$ can issue that generates some intermediate computation tree C_0, there exists a finite schedule Γ on $[t_0, t_1]$ such that $\mathcal{R}$ brings the computation tree to a rest state C_1, i.e., $\texttt{ready}(C_1) = \{\texttt{root}(C_1)\}$.*

This definition is sufficient, since once the computation tree is at the rest state, and only the root node is ready, $\mathcal{R}$ can execute each transaction serially and complete the computation.

Restrictions to avoid semantic deadlock

The general OAT model described in Section 1.3 exhibits semantic deadlock because it may enter a state where two parallel aborting transactions X and Y 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 $\texttt{xMod}(T)$'s level or higher, however, one can show the OAT model is free of semantic deadlock. More formally, for all transactions T, Definition 6.28 restricts the memory footprint of $\texttt{abortactions}(T)$.

Definition 6.28 *An execution (represented by a computation tree C) has* ***abort actions with limited footprint*** *if the following condition is true for all transactions $T \in \texttt{aborted}(C)$. At time t, if a memory operation $v \in \texttt{abortactions}(T)$ accesses location ℓ and $\texttt{owner}(\ell) \in \texttt{modAnces}(\texttt{xMod}(T))$, then*

1. *if v is a read, then $\ell \in \texttt{R}(T)$, and*
2. *if v is a write then $\ell \in \texttt{W}(T)$.*

Definition 6.28 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 cannot read from (write to) any location ℓ which is owned by any Xmodules which are ancestors of $\texttt{xMod}(T)$ (including $\texttt{xMod}(T)$ itself), unless ℓ is already in the read (or write set) of T.

The properties of Xmodules from Theorem 6.8 in combination with the ownership-aware commit mechanism imply that transaction read sets and write sets exhibit nice properties. In particular, Corollary 6.29 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 $\mathcal{D}$. Lemma 6.30, using Corollary 6.29, shows that a computation whose abort actions have limited footprint, a memory operation v from a transaction T's abort action can only conflict with another transaction T' generated by a lower-level Xmodule than $\texttt{xMod}(T)$. Using these properties, Theorem 6.31 shows that the OAT model is free from semantic deadlock assuming that aborted actions have limited footprint.

Corollary 6.29 *For any transaction T if $\ell \in \texttt{R}(T)$, then $\texttt{xMod}(T) \in \texttt{modDesc}(\texttt{owner}(\ell))$.*

PROOF. This corollary follows from Definition 6.1, Theorem 6.8, and induction on how a location ℓ can propagate into read sets and write sets 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 M can only generate conflicts with a "lower-level" Xmodule.

Lemma 6.30 *Suppose the OAT model generates an execution where abort actions have limited footprint. For any transaction T, consider a potential memory operation $v \in$ `abortactions`(T). If v conflicts with transaction T', then* $\texttt{xid}(\texttt{xMod}(T')) > \texttt{xid}(\texttt{xMod}(T))$.

PROOF. Suppose $v \in \texttt{abortactions}(T)$ accesses a memory location ℓ with $\texttt{owner}(\ell) = M$. Since $\texttt{abortactions}(T) \subseteq \texttt{memOps}(T)$, by the properties of Xmodules given in Definition 6.5, we know that either $M \in \texttt{modAnces}(\texttt{xMod}(T))$, or $\texttt{xid}(M) > \texttt{xid}(\texttt{xMod}(T))$. If $M \in \texttt{modAnces}(\texttt{xMod}(T))$, then by Definition 6.28, T already had ℓ in its read or write set. Therefore, 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, it must be that $\texttt{xid}(M) > \texttt{xid}(\texttt{xMod}(T))$. If v conflicts with some other transaction T', then T' has ℓ in its read or write set. Therefore, from Corollary 6.29, xMod(T') is a descendant of M. Thus, we have $\texttt{xid}(\texttt{xMod}(T')) > \texttt{xid}(M) > \texttt{xid}(\texttt{xMod}(T))$. □

Theorem 6.31 *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, all active transactions T have $\texttt{status}[T] =$ `PENDING_ABORT`. 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, $k-1, k-2, \ldots 1, 0$. 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 $\texttt{xid}(\texttt{xMod}(T)) > i$, i.e., no instances T from Xmodules with `xid` larger than i. During phase i, we finish aborting all active transaction instances T with $\texttt{xid}(\texttt{xMod}(T)) = i$. By Lemma 6.30, any abort action for a T, where $\texttt{xid}(\texttt{xMod}(T)) = i$, can only conflict with a transaction instance T' from a lower-level Xmodule, where $\texttt{xid}(\texttt{xMod}(T')) > i$. 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 consist 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 $\texttt{xMod}(X)$ 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 cannot acquire new locks that were not already acquired by the transaction it is compensating for.

6.6 Related Work

This section describes other work in the literature on open-nested transactions. In particular, this section focuses on two related approaches for improving open-nested transactions, and distinguish them from our work.

Ni et al. [117] 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 nor TM system restrict exactly what memory can be passed into a method of an `open_atomic` class, and the TM system performs a vanilla open-nested commit for a nested transaction, not a safe-nested commit. Thus, it is unclear what exact guarantees are provided with respect to serializability and/or deadlock freedom.

Herlihy and Koskinen [62] describe a technique of transactional boosting which allows transactions to call methods from a nontransactional module M. Roughly, as long as M is linearizable and its methods have well-defined inverses, the authors show that the execution appears to be "abstractly serializable." Boosting does not, however, address the cases when the lower-level module M writes to memory owned by the enclosing higher-level module, or when programs have more than two levels of modules.

6.7 Conclusions

This chapter describes the OAT system, which provides a disciplined methodology for open nesting and bridges the gap between memory-level mechanisms for open nesting and the high-level methodology. Using OAT, the programmer is provided with a concrete set of guidelines as to how Xmodules share data and interact with each other. As long as the program conforms to the guidelines, the OAT system guarantees abstract serializability, which results in a sensible program behavior.

One distinct feature about the OAT system is that, unlike any other transactional memory system proposed, the programmer does not specify transactions explicitly using `atomic` blocks. Rather, she programs with transactional modules, specifying levels of abstractions among program components, and transactions are generated implicitly. With this transactional module interface, the programmer focuses on structuring the code and data into modular components, and the OAT system maintains the memory abstraction that data belonging to a module is updated atomically and thus presents a consistent view to other modules.

Even though this transactional module interface seems promising, the linguistics of the OAT system is an under-investigated topic. As the design stands, the linguistic interface is rather clumsy, since the OAT system employs ownership types for the programmer to specify levels of abstractions and data sharing, and the syntax can get cumbersome quickly as the software grows larger. Another topic of investigation is the expressive power of the linguistics. There are all interesting future directions to pursue.

Chapter 7

Location-Based Memory Fences

This chapter explores the notion of a ***location-based memory fence*** which, when used correctly, provides the same guarantees as an ordinary memory fence and incurs overhead only when synchronization is necessary. Unlike other memory abstractions studies in previous chapters, which are supported by an underlying runtime system, the location-based memory fences can be more efficiently supported by hardware. This chapter proposes a hardware mechanism for location-based memory fences, proves its correctness, and evaluates its potential performance benefit.

On many modern multicore architectures, threads[1] typically communicate and synchronize via shared memory. Classic synchronization algorithms such as Dekker [39], Dijkstra [38], Lamport (Bakery) [85], and Peterson [122] use simple load-store operations on shared variables to achieve mutual exclusion among threads. All these algorithms employ an idiom, referred to as the ***Dekker duality*** [34], in which every thread writes to a shared variable to indicate its intent to enter the critical section and then reads the other's variable to coordinate access to the critical section.

Crucially, the correctness of such an idiom requires that the memory model exhibit ***sequential consistency*** (SC) [86], where all processors observe the same sequence of memory accesses, and within this sequence, the accesses made by each processor appear in its program order. While the SC memory model is the most intuitive to the programmer, existing architectures typically implement weaker memory models that relax the memory ordering to achieve higher performance. The reordering affects the correctness of the software execution in some cases such as the Dekker duality, in which it is crucial that the execution follow the program order, and the processors observe the relevant accesses in the same relative order.

Consider the following code segment shown in Figure 7-1, which is a simplified version of

[1] Throughout this chapter, I assume that threads are surrogates of processors and use the terms threads and processors interchangeably. In particular, I use threads in the context of describing an algorithm and processors in the context of describing hardware features.

```
                    Initially x = y = 0;

        Thread 1                              Thread 2
T1.1  x = 1;                          T2.1  y = 1;
T1.2  if(y == 0) {                    T2.2  if(x == 0) {
T1.3    /* critical section */        T2.3    /* critical section */
T1.4  }                               T2.4  }
T1.5  x = 0;                          T2.5  y = 0;
```

Figure 7-1: A simplified version of the Dekker protocol (omitting the mechanism to allow the threads to take turns), assuming sequential consistency.

the Dekker protocol [39][2], using the idiom to synchronize access to the critical section among two threads. With "Total-Store-Order" and "Processor-Ordering" memory models, which are the memory models considered in this chapter, the read in line T1.2 may get reordered with the write in line T1.1 (and similarly for Thread 2), such that Thread 2 "observes" the read of Thread 1 (line T1.2) before it observes the write of Thread 1 (line T1.1). Thus, Thread 1 and Thread 2 observe different ordering of the reads and writes, resulting in an incorrect execution and causing the two threads to enter the critical section concurrently.

To ensure a correct execution in such cases, architectures that implement weak memory models provide ***serializing instructions*** and ***memory fences*** which allow one to enforce a specific memory ordering when necessary. Thus, a correct implementation of the Dekker protocol for such systems would require a pair of memory fences between the write and the read (between lines T1.1 and T1.2, and lines T2.1 and T2.2 in Figure 7-1), ensuring that the write becomes globally visible to all processors before the read is executed.

Traditional memory fences are program-based — they are part of the code the processor is executing, and they cannot be avoided even when the program is executed serially, or when the synchronization is unnecessary because no other threads are reading the updated memory location. Furthermore, when a memory fence is executed, the processor stalls until all outstanding writes before the fence in the instruction stream become globally visible. Thus, memory fences are costly, taking many more cycles to complete than regular reads and writes. I ran a simple microbenchmark on AMD Opteron with 4 quad-core 2 GHz CPUs, and the result shows that a thread running alone and executing the Dekker protocol with a memory fence, accessing only a few memory locations in the critical section, runs $4-7$ times slower than when it is executing the same code without a memory fence.

This work proposes a ***location-based memory fence***, which causes the executing thread T_1 to "serialize" only when another thread T_2 attempts to access the memory location associated with the memory fence. Location-based memory fences aim to reduce the latency in program execution incurred by memory fences. Unlike a program-based memory fence, a location-based memory fence is *conditional* and *remotely enforced* by T_2 onto T_1; whether T_1 serializes or not depends on whether there exists a T_2 that attempts to access the memory location associated with the memory fence. In essence, location-based memory fences allow T_1 to avoid the latency of memory fences and instead have T_2 borne the overhead of communication to trigger T_1 to serialize. Performance benefit is obtained if the latency avoided by T_1 is greater than the communication overhead borne by T_2.

The concept of location-based memory fences is particularly well suited for applications that employ the Dekker duality. It turns out that this idiom is commonly used to optimize applications that exhibit ***asymmetric synchronization patterns***, where one thread, the ***primary thread***, enters a particular critical section much more frequently than other threads running in the same process, referred to as the ***secondary threads***. Such applications typically employ an augmented version of the Dekker protocol: the secondary threads first compete for the right to synchronize with the primary thread (by grabbing a lock); once obtaining the right, the winning secondary thread synchronizes with the primary thread using the Dekker protocol. The augmented Dekker protocol intends to speed up the execution path of the primary thread, even at the expense of the secondary threads. In such applications, it is also desirable to optimize away the overhead of fences on the primary

[2] This simplified version is vulnerable to livelock, where both threads simultaneously try to enter the critical section — each thread sets its own flag, reads the other thread's flag, retreats, and retries. Without some way of breaking the tie, the two threads can repeatedly conflict with each other and retry perpetually. The full version is augmented with a mechanism to allow the threads to take turns and thus guarantees progress. For the sake of clarity, the simplified version is presented here.

thread's execution path when the application executes serially or when there is no contention.

Many examples of such applications exist. For example, Java Monitors are implemented with biased locking [36,76,119], which uses an augmented version of the Dekker protocol to coordinate between the bias-holding thread (primary) and a revoker thread (secondary). The Java Virtual Machine (JVM) employs the Dekker duality to coordinate between mutator threads (primary) executing outside of the JVM (via the Java Native Interface) and the garbage collector (secondary) [36]. In a runtime scheduler that employs a work-stealing algorithm [8,17,20,21,49,55,80], the "victim" (primary) and a given "thief" (secondary) coordinate a steal using an augmented Dekker-like protocol. Finally, in network packet processing applications, each processing thread (primary) maintains its own data structures for its group of source addresses, but occasionally, a thread (secondary) might need to update data structures maintained by a different thread [134].

Such applications motivate the study of location-based memory fences. This chapter proposes a hardware mechanism to implement location-based memory fences, which aims to be lightweight and requires only modest modifications to existing hardware: two additional registers per processor and a new load instruction, which implements a functionality that many modern architectures already support. With this hardware design for location-based memory fences, a thread running alone and executing the Dekker protocol will observe only negligible overhead when using location-based memory fences compared to executing the same code without fences at all.

To evaluate the feasibility of location-based memory fences, I have implemented a software prototype to simulate its effect and applied it in two applications that exhibit asymmetric synchronization patterns. While the software implementation incurs much higher communication overhead than the proposed hardware mechanism would, experimental results show that applications still benefit from the software implementation and would scale better if the communication overhead were smaller. These results inspires confidence that the proposed hardware design for location-based memory fences is a viable and appealing alternative to traditional program-based memory fences.

The rest of the chapter is organized as follows. Section 7.1 gives an abbreviated background on why reordering occurs in architectures that support a weaker memory model. Section 7.2 presents the proposed hardware mechanism for location-based memory fences. Section 7.3 formally defines the specification of location-based memory fences and proves that the proposed hardware mechanism implements the specification. Section 7.4 evaluates the feasibility of location-based memory fences using a software prototype implementation with two applications. Section 7.5 gives a brief overview on related work. Finally, Section 7.6 draws concluding remarks.

7.1 Store Buffers and Memory Accesses Reordering

This section summarizes features of modern architecture design which are necessary for the proposed hardware mechanism for location-based memory fences. In particular, throughout the rest of the chapter, we shall assume that the target architecture implements either the ***Total Store Order (TSO)*** model (implemented by SPARC-V9 [135]) or the ***Process Ordering (PO)*** model (implemented by Intel 64, IA-32 [71], and AMD64 architectures [3]), and its cache controllers employ the MESI cache coherence protocol [71] (or other similar variants such as MSI [61] and MOESI [3]). This section also describes the use of store buffers and how ***memory reordering*** can occur, i.e., how the observable order in which memory locations are accessed can differ from program order. Memory reordering can be introduced either by the compiler or the underlying hardware. Compiler fences that prevent the compiler from reordering have relatively small overhead, whereas the memory fences that prevent reordering at the hardware level are much more costly. This section focuses on reordering at the hardware level.

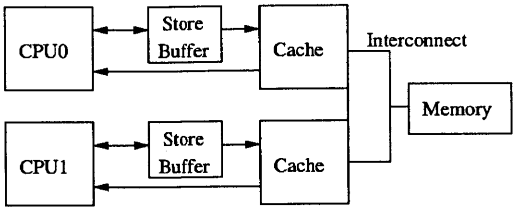


Figure 7-2: A simplified illustration of the relationship between the CPUs, the store buffers, and the memory hierarchy. Each CPU is connected with its own private cache. In addition, a store buffer is placed between the CPU and the cache, so that a write issued by the CPU is first stored in the store buffer and flushed out to the cache at later time. A read may be served by the cache, or by the store buffer if the store buffer contains a write to the same target address as the read.

Definition 7.1 (TSO and PO ordering principles) *Architectures implementing TSO and PO enforce the following ordering principles*[3] *for regular reads and writes issued by a given (single) processor:*

1. *Reads are not reordered with other reads;*
2. *Writes are not reordered with older reads;*
3. *Writes are not reordered with other writes; and*
4. *Reads may be reordered with older writes if they have different target locations (but they are not reordered if they have the same target location).*

Furthermore, in a multiprocessor system, when one considers the interleaving of memory accesses issued by multiple processors, the TSO and PO models enforce the following principles:

5. *Writes by a given processor are seen in the same order by all processors; and*
6. *Any two stores from two different processors, say P_1 and P_2, are seen in a consistent order by processors other than P_1 and P_2.*

Modern architectures typically support out-of-order execution, but "commit" executed instructions in order, thereby enforcing Principles 1–3. We shall come back to visit this pointer later and precisely define what it means for a memory access instruction to be committed. First, we shall focus our attention on Principle 4, which violates the Dekker duality — it allows the read in line T1.2 of Figure 7-1 to appear to Thread 2 as if it has occurred before line T1.1, even though it appears as executed in order for Thread 1.

The reason behind Principle 4 is to allow a typical optimization that modern architectures implement — writes performed by an executing processor are queued up in a private first-in-first-out (FIFO) queue, referred to as the ***store buffer***, instead of being written out to the memory hierarchy. Figure 7-2 provides a simplified illustration of the relationship between the processors (CPUs), the store buffers, and the memory hierarchy. Though not explicitly shown in Figure 7-2, the memory hierarchy in modern architectures typically consists of several levels of private and shared caches and the main memory. The further away the memory hierarchy is from the processor, the higher the latency it incurs. The use of a store buffer improves performance, because writing to a store buffer avoids the latency incurred by writing out to the cache. A write in the store buffer is only visible to the executing processor but not to other processors, however. Thus, from other processors' perspective, it may appear as if a read has taken place before an older write, which differs from the ordering perceived by the executing processor (i.e., its program order).

Most systems employ a cache coherency protocol between the processors. which governs accesses to memory locations and enforces a consistent view of the data among all the caches. The

[3] This is not a complete list but rather a relevant subset for the purposes of our discussion. I refer interested readers to [3, 71, 135] for full details.

cache coherency guarantees that a write becomes *globally visible* once it leaves the store buffer and is written to the cache. The proposed hardware mechanism for location-based memory fences requires that the target architecture employ the ***MESI cache coherence protocol*** [71] (and can e adapted to other variants such as MSI [61] and MOESI [3]). A ***cache controller*** manages a cache using the MESI protocol ensuring that each cache line is labeled with one of the following four states:

1. ***Modified***: the cache line has been modified and no other caches have this cache line;
2. ***Exclusive***: this cache has exclusive access to this cache line and its content matches that in main memory;
3. ***Shared***: the cache line may be shared by other caches; or
4. ***Invalid***: the cache line is invalid, which is equivalent to saying that this cache does not have this particular cache line.

A cache is said to ***hold*** a particular cache line if the cache has the cache line in Modified or Exclusive state. When the oldest write is flushed out of the store buffer, the cache controller must first obtain the corresponding cache line in Exclusive state (if it does not hold the cache line in Exclusive or Modified state already) in order to complete the write. On the other hand, a cache that holds a line may receive a request to ***downgrade*** the cache line into Shared or Invalid state, depending on whether the requesting cache wishes to read or write to the line. A cache that receives a downgrade request must first write the line back to the main memory (if it's modified) before it downgrades the line.

Now we shall define more precisely what it means for a memory access instruction to be committed. A read instruction is considered to be ***committed*** once the data is available (in a state other than Invalid) in the processor's private cache. A read may be speculatively executed out of order, but it must be committed in order. That is, the processor may perform a speculative read and fetch the cache line early, but if the cache line gets invalidated between the speculative read and when the read should commit in program order, the processor must reissue the read and fetch the cache line again. Once a read is committed successfully, the read value can be used in subsequent instructions.

A write instruction involves two phases: "committed" and "completed." A write is considered to be ***committed*** once it is written to the store buffer, although its effect is not yet visible to other processors. A write is considered to be ***completed*** when it is flushed from the store buffer and written to the processor's cache, which entails obtaining the cache line for the flushed location in Exclusive state and updating the cache line with the written value. Once a write completes, its effect becomes globally visible, since the cache coherence protocol ensures that all processors have a consistent view.

Since reads and writes are committed in the order that they arrive in the instruction stream, and the store buffer flushes out entries in FIFO order, it is easy to see how Principles 1–3 and Principle 5 of Definition 7.1 are enforced. The only reordering that can occur between a pair of memory accesses is a write followed by a read with a different target address. Since the read can be committed (i.e., obtaining the cache line in Shared state) while the older write is still in the store buffer, the resulting behavior "observable" by other processors is that the read appears to have taken place before the older write.

The executing processor does not "observe" this reordering, however. An executing processor always sees its own write because the hardware employs ***store-buffer forwarding***, by which a read with a target address that appears in the store buffer is serviced by the store buffer instead of by the cache. Incidentally, the store-buffer forwarding also enforces the ordering principle that a read is not reordered with an older write if they have the same target address (Definition 7.1,Principle 4).

Furthermore, due to store-buffer forwarding, when two writes to the same location from two processors, say P_1 and P_2, interleave, the write ordering observed by P_1 may differ from the write ordering observed by P_2, because each processor always sees its own write as soon as it commits, but not the write performed by the other processor until that write is completed and reaches the cache. On the other hand, all other processors besides P_1 and P_2 observe a consistent ordering of the two writes (i.e., in the order that the writes complete), as stated in Definition 7.1, Principle 6.

A traditional memory fence, `mfence`, is used to ***serialize*** a processor instruction stream, ensuring that the memory accesses before the `mfence` arrive at the cache before memory accesses after the `mfence`. That is, memory accesses become globally visible in the same relative order to the `mfence` as they appear in the executing processor's instruction stream. Operationally, when the executing processor encounters an `mfence`, the `mfence` simply forces the processor to stall until its store buffer is drained, flushing all entries out to the cache in FIFO order.

Even though an `mfence` ensures that instructions of the executing processor arrive at the cache in the same order as in the executing processor's instruction stream, one must note that using a single `mfence` by itself does not necessarily prevent another processor P_2 from observing the memory accesses in a different order than processor P_1. In particular, P_2 can only observe an ordering of P_1's memory accesses A_1 and A_2 by performing memory accesses B_1 and B_2 which access the same memory locations as A_2 and A_1 respectively, thereby inferring an ordering from the results of performing B_1 and B_2 (in that order). If B_1 is a write and B_2 is a read, then B_2 can reach the cache before B_1, which causes P_2 to infer that A_2 occurred before A_1, based on its assumption that B_1 occurred before B_2. Thus, correct use of a memory fence typically involves a pair of `mfence` instructions, ensuring that the two processors involved agree on the ordering of relevant memory accesses performed by both.

Besides `mfence`, other events in the system may trigger a processor to flush its store buffer, such as a context switch, an interrupt, or other serializing instructions. The store buffer also naturally flushes the oldest entry to memory whenever the system bus is available. The invariant is that, the entries are always flushed in FIFO order.

7.2 Location-Based Memory Fences

This section describes location-based memory fences, or `l-mfence` in details, including its informal specification, usage, and a proposed hardware implementation, referred as the ***LE/ST mechanism***. The formal specification, as well as a correctness proof, is presented in Section 7.3. The proposed hardware mechanism that implements the `l-mfence` assumes an underlying architecture as described in Section 7.1.

Informal specification and usage

An `l-mfence` takes in two inputs: a location x, referred as the ***guarded location*** and a value v to store in x (see Figure 7-3(a)). Informally, an `l-mfence` executes a memory fence "on demand" — the `l-mfence` serializes the instruction stream of the executing processor P only when another processor attempts to read the guarded location.

Programming using an `l-mfence` is very similar to programming with an `mfence`— threads synchronizing via `l-mfence` need to coordinate with each other and be careful as to where to place the `l-mfence` and which memory location to guard / read after. Just like `mfence`, correct usages of `l-mfence` consist of a pair of memory fences. When used correctly, the serialization of P's instruction stream enforces a relative order between the store S associated with the execution of the

```
            Primary Thread                          Secondary Thread
K1  l-mfence(&x,1);                         J1  y = 1;
K2                                          J2  mfence();
K3  if(y == 0) {                            J3  if(x == 0) {
K4    /* critical section */                J4    /* critical section */
K5  }                                       J5  }
K6  x = 0;                                  J6  y = 0;
```

(a)

Instruction translation for l-mfence(&x,1) (line K1 in Thread 1)

```
K1.1  mov LEBit <- 1          //set LEBit
K1.2  mov LEAddr <- &x        //LEAddr gets addr of x
K1.3  le  &x                  //load x in E state
K1.4  st  [&x] <- 1           //store x = 1
K1.5  bnq LEbit, 0, done      //jump to done if LEBit != 0
K1.6  mfence                  //else execute mfence
K1.7  done:
K1.8  //the rest of the program (line K3)
```

(b)

Figure 7-3: (a) The asymmetric Dekker protocol using location-based memory fences. The code for the primary thread is shown in lines K1–K6, and the code the secondary thread is shown in lines J1–J6. **(b)** The instructions generated for the `l-mfence` shown in line K1 in **(a)**.

`l-mfence` and the other memory accesses performed by P, and this order is observed consistently across all processors. That is, if P executed S before (after) an access A, no other processor would infer that S "happened" after (before) A.

The effects of an `l-mfence` are very similar to the effects of a regular `mfence`: First, when either an `mfence` or an `l-mfence` is used in a program, an implicit compiler fence should be inserted in that place to prevent reordering of memory accesses by the compiler. Second, neither an `mfence` nor an `l-mfence` themselves prevent other processors from observing a reordering of the memory accesses of the executing processor, and must be used in pairs. Finally, serialization enforced by an `mfence` or an `l-mfence` does not enforce any relative order between two accesses that both happened before (or after) instruction. This serialization ensures that all processors (including P) consistently observe that two accesses A_1 and A_2 happened before (or after) S, but the processors may not have a consistent view of the relative order between A_1 and A_2. The relative order between these accesses is still defined by the TSO / PO memory model.

Figure 7-3(a) presents the usage of an `l-mfence` in the Dekker protocol. To guarantee mutual exclusion it is crucial that *both* processors insert memory fences between the write and the read, to prevent the other processor from observing reordering. For `l-mfence`, the pairing can be with either another `l-mfence` or an ordinary `mfence`.

The correct usage of `l-mfence` has one distinct requirement that is not needed by the use of `mfence`, however. The use of `l-mfence` guarantees that the program execution is serialized correctly only if the program execution does not contain concurrent writes to the guarded location while the `l-mfence` is "in effect." That is, if a processor P executes an `l-mfence` with guarded location x, all other processors running concurrently may read from the guarded location x, but they are prohibited from writing to x or executing an `l-mfence` where the guarded location is x. An `l-mfence` no longer guarantees a correct serialization of P's instruction stream and may be downgraded to an ordinary store (writing v to location x) if a concurrent write is detected while the `l-mfence` is in effect. The reason for this requirement is explained later in this section.

It is important to note that this requirement does not forbid a program using `l-mfence` from having two different threads writing to the location guarded by an `l-mfence`. The requirement is simply that the writes should not be concurrent. In addition, two different threads may execute `l-mfence` with the same guarded location x throughout a program execution, as long as the two `l-mfence` instructions are not in effect concurrently. Even though this requirement is not necessary for an ordinary `mfence`, concurrent writes in a program without proper synchronization typically constitutes a bug, since it results in a nondeterministic execution. Henceforth, whenever we discuss the condition of ***no concurrent writes*** in the context of `l-mfence`'s semantic guarantee, it specifically means that no other processors should be writing to *the guarded location* of an `l-mfence` while the `l-mfence` is *in effect.*

A proposed hardware implementation — the LE/ST mechanism

The proposed implementation of `l-mfence` employs a new hardware mechanism, called ***load-exclusive / store***, or ***LE/ST***. The name of the LE/ST mechanism is reminiscent of the hardware mechanism of load-linked / store-conditional, or LL/SC, originally proposed by Jensen, Hagensen, and Broughton [72]. As we shall see, however, while the concept of linking a load to a store is similar, the LE/ST mechanism operates differently, and its purpose is to provide a fence between memory accesses, not an atomic operation.

Conceptually, the LE/ST mechanism allows the processor to set up a "link" to keep track of the status of the store associated with the `l-mfence` (i.e., whether the store to the guarded location is committed or completed as defined in Section 7.1). The link is set as long as the store is committed but not yet complete. While the link is set, the processor coordinates with the cache controller (for its private cache) to monitor attempts to access the guarded location.

When the link is set, another processor's attempt to read the guarded location causes the processor to clear the link and triggers actions necessary to serialize the instruction stream. On the other hand, if the LE/ST mechanism detects a concurrent write while the link is in effect, downgrading the `l-mfence` to an ordinary store is necessary to ensure that the overall system makes forward progress. Whenever the store completes naturally (before another processor attempts to access the guarded location), the processor clears the link and thus stops guarding the location. We shall first describe how the LE/ST mechanism operates, and then explain why it is necessary for the LE/ST mechanism to downgrade the `l-mfence` if a concurrent write is detected.

LE/ST requires one new instruction and two additional hardware registers. The new instruction, `le`, takes one operand, the location of the variable to load, and obtains Exclusive state on that location. Therefore, once `le` is ***committed***, the processor has the location in its cache in at least Exclusive state, and no other processors have a valid copy of the location in their cache. Since `le` is very similar to a regular load, except the requirement for having at least Exclusive state on the location, it can be easily implemented by modern architectures using the MESI coherency protocol. The two additional hardware registers are `LEBit` and `LEAddr`, both readable and writable by the processor, and readable by the cache controller.

Figure 7-3(b) presents an assembly-like translation of the `l-mfence` performed by the executing processor, where the value 1 is being stored in location x.[4] Initially, `LEBit` and `LEAddr` are cleared. As part of the `lmfence(&x, 1)`, the processor initiates the link to the guarded location, which involves three instructions. The first two instructions set the `LEBit` with 1 and `LEAddr` with the address of x (lines K1.1 and K1.2 in Figure 7-3(b)). Next, the `le` instruction in line K1.3 loads x into the cache in Exclusive state, so that no other processor holds a copy of x in its cache. Once the

[4] The code shown is not strictly assembly. First, it is not using a particular instruction set. Second, for the sake of clarity, I chose to use the store instruction (line K1.4) instead of using the regular move instructions to specify instructions that write to memory (i.e., non registers).

cache line of x is obtained in Exclusive state, the link is fully set. The st instruction in line K1.4 stores the value 1 to x, committing it into the store buffer. If for any reason the link is broken, implied by the zero value in LEBit (line K1.5), the processor executes an mfence (line K1.6). The mfence causes the processor to serialize its execution — it flushes the store buffer, and thus completes the store of the guarded location, making it globally observable by other processors. If the link is not broken when the st in line K1.4 commits, the processor may continue without flushing the store buffer.

Let's now examine how the cache controller interacts with the processor to guard the location stored in LEAddr. Essentially, the LE/ST mechanism piggybacks on the cache coherency protocol to detect another processor's attempt to access the guarded location. Whenever both LEBit and LEAddr are set, the cache controller listens to cache coherency traffic, and notifies the processor if any request requires the cache controller to downgrade the state of the cache line corresponding to the guarded location. There are three possible events that cause the cache line to be downgraded from Exclusive state:

A. ***Eviction*** — the cache line needs to be evicted;
B. ***Concurrent read*** — the cache line needs to be downgraded to Shared state; and
C. ***Concurrent write*** — the cache line needs to be downgraded to Invalid state.

When the cache controller encounters an eviction or a concurrent read — in these cases the cache controller notifies the processor and waits for the processor's response before it takes any actions regarding the guarded location, since these events require the serialization of the instruction stream. When the processor receives the notification from the cache controller, it clears the LEBit and LEAddr and flushes the store buffer. The processor responds to the cache controller only when the most up-to-date value of the guarded location is flushed from the store buffer to the cache. When the cache controller receives the response it replies back to the requesting processor. Since the cache controller only resumes the action regarding the guarded location after it receives a response from the processor, it is guaranteed that it will send the most up-to-date value to the read request (or to memory in the case of eviction). By clearing the LEBit, the processor remembers that the link to the guarded location is broken. In the event that the link is broken before st (line K1.4) was committed, the code for l-mfence takes the branch that executes an mfence, causing the store buffer to flush (line K1.5) after the store commits. If none of the scenarios above occurs, the link remains set for as long as the store is not yet complete and the processor still owns the cache line.

When the cache controller encounters a concurrent write — in this case the LE/ST mechanism does not guarantee the serialization and regards the l-mfence as a regular store. The cache controller notifies the processor, and the processor simply clears the link and responds immediately to the cache controller, without flushing the store buffer. The cache controller can then respond to the requesting processor. The l-mfence semantic is not guaranteed in the event of concurrent write because the LE/ST mechanism may create additional dependencies between processors when several of them attempt to flush their store buffers, and these dependencies may cause the system to deadlock. To avoid the possibility of a deadlock and allow the system as a whole to make forward progress, the LE/ST mechanism gives up the serialization guarantee in the presence of concurrent writes and regards the store associated with the l-mfence as a regular write. How additional dependencies are created by the LE/ST mechanism and why regarding the l-mfence as a store avoids a deadlock are discussed in detail after the following concluding remarks on the hardware implementation.

The design of the LE/ST hardware mechanism is intended to be light-weight and efficient, which uses only existing mechanisms and adds minimal hardware. Since the design assumes only one pair of LEBit and LEAddr is allocated per processor, if a processor encounters a second l-mfence

```
          Processor 1                           Processor 2
T1.1  x = 1;                         T2.1  y = 1;
T1.2  lmfence(&y, 1);                T2.2  lmfence(&x, 1);
```

Figure 7-4: An example of multiple writers to the same location, where some of the writes are not protected by `l-mfence`. This situation can cause deadlock without the additional mechanism to avoid it.

while the link from the first `l-mfence` is still in effect, the processor must clear the link and flush the store buffer before it can proceed with the second `l-mfence`, *unless* the second `l-mfence` has the same guarded location as the first one. That means that a processor may possibly handle two consecutive `l-mfence` instructions with the same guarded location without flushing the store buffer in between. The semantics of `l-mfence` is still guaranteed (assuming no concurrent writes), even if another processor attempts to read the guarded location between the two `l-mfence` instructions. However, in the event where a downgrade request arrives at the processor between setting up the `LEBit` (line K1.1) and committing `st` (line K1.4) for the second `l-mfence`, the processor will flush the store buffer twice — the first flush is performed when the processor is notified, making the store associated with the first `l-mfence` visible, and the second flush is performed after the `st` commits, via taking the branch (lines K1.5 and K1.6) since the link has been cleared, making the store associated with the second `l-mfence` visible.

Examining the LE/ST mechanism in the context of the Dekker protocol, since `le` ensures that the primary processor has the cache line for x in Exclusive state before the `st` in line K1.4, its cache controller must receive a downgrade request from a secondary processor before the secondary processor can access x. Furthermore, since the cache controller of the primary processor cannot respond to the downgrade request until the primary processor responds to its notification, the secondary processor will see the most up-to-date value of x.

Why `l-mfence` *does not guarantee serialization when concurrent writes exist*

To explain how the system may deadlock in the case where concurrent writes exist, let's examine a simple example in which two processors may deadlock. Figure 7-4 shows code snippets that are executing concurrently on two different processors. Processor P_1 writes to memory location x and executes an `l-mfence` with guarded location y. Similarly, P_2 executes the mirrored code which writes to location y and executes an `l-mfence` with guarded location x. Now lets look at the store buffer of P_1 after it executes an `l-mfence` on location y. Since the write to x is not guarded by an `l-mfence`, location x may or may not be in P_1's cache (and it is not in this example, given that P_2 is executing concurrently), but location y is in P_1's cache in Exclusive state. P_2's store buffer similarly contains y followed by x, with x being in P_2's cache in Exclusive state but not y.

Suppose that P_1 is trying to flush location x from its store buffer to its private cache. In order to do so, P_1's cache must gain Exclusive state on x. By the MESI protocol, P_1's cache controller thus sends a request to P_2, who holds x, to downgrade its state to Invalid. Assuming P_2's `l-mfence` is still in effect when it received the request, in order to guarantee P_2's serialization in such a scenario, P_2's cache controller must notify the processor and not respond to P_1's cache request until P_2 performs an `mfence` successfully. In order for P_2 to execute an `mfence`, P_2's cache controller must now obtain the cache line for y in Exclusive state, which involves sending a request to P_1 to invalidate its cache line on y. Similarly, since P_1's `l-mfence` is still in effect, in order to guarantee P_1's serialization, P_1's cache controller cannot respond to P_2's cache request until P_1 executes an `mfence` successfully, which means it must obtain Exclusive state on x. Thus, the cache controllers of the two processors are locked in circular dependencies — P_1 is waiting on getting x in Exclusive state before it can release the cache line on y, and P_2 is waiting on getting y in Exclusive state before it can release the cache line on x.

This example illustrates the dependency the LE/ST mechanism creates between satisfying an incoming request to invalidate the guarded location and obtaining Exclusive state on some memory location in the store buffer. This dependency means that the cache controller of a processor P_1 with an `l-mfence` in effect can no longer immediately respond to an invalidation request from another cache controller on the guarded location. Instead, the cache controller must wait until P_1 successfully flushes its store buffer (at least up to the point where the guarded location is flushed into the cache), which involves gaining Exclusive state on memory locations in P_1's store buffer. The system deadlocks if another processor P_2 has exactly the opposite dependency, i.e., P_2 has P_1's guarded location in its store buffer and has an `l-mfence` in effect on a location which happens to be in P_1's store buffer. Regarding the `l-mfence` as an ordinary store breaks the circular dependency, because the processor does not attempt to flush its store buffer and the cache controller can immediately respond to the incoming invalidation request.

Even though Figure 7-4 illustrates a simple example involving only two processors with two memory locations, the circular dependencies can potentially occur among several processors with multiple guarded memory locations, where each processor has an `l-mfence` in effect. Since all processors regards the `l-mfence` as a regular store when a concurrent write is detected, the potential circular dependencies are guaranteed to be broken and the system as a whole can make forward progress.

The circular dependencies between satisfying an incoming request and obtaining Exclusive state can only rise when concurrent writes exist in the program. This is because only a write operation is saved in the store buffer, and needs to gain Exclusive state before reaching the cache. A read operation does not go through the store buffer, and therefore will not cause a dependency when the processor is trying to flush the store buffer. To distinguish between read and write attempts, the LE/ST mechanism relies on the cache controller and its implementation of the cache coherency protocol, to only send an invalidation request to another cache if it intends to write to a location, but not if it intends to read — in which case, it sends a "downgrade to shared" request.

The design decision to downgrade the `l-mfence` into an ordinary store and not guarantee serialization if a concurrent write is detected was made to keep the LE/ST mechanism lightweight. In addition, concurrent writes in a program without proper synchronization typically constitute a bug and result in nondeterministic execution. Therefore, keeping the `l-mfence` semantics would cumbersome the implementation and would not benefit the programmer. Even though the existence of concurrent writes may not necessarily lead to circular dependencies, it is certain that circular dependencies involve concurrent writes. Concurrent writes are easily detected by the type of the coherency message received by the cache controller, and the processor actions are simple and effective — the deadlock is avoided. Thus, whenever concurrent writes are detected, the LE/ST mechanism downgrades the `l-mfence` semantics, instead of keeping track of actual dependencies, which would require global coordination among all cache controllers.[5]

Finally, there is one important implication that follows from how the LE/ST mechanism handles concurrent writes. That is, a memory location guarded by an `l-mfence` should be allocated on its own cache line so as to avoid false sharing. Otherwise, a cache controller may receive an invalidation

[5] One could imagine that some form of policy can be employed in the cache controller so that an `l-mfence` is not immediately downgraded whenever a concurrent write is detected. For instance, one could employ some form of time-out policy — a cache controller guarding location x but needs Exclusive on y to flush the store buffer only downgrades the `l-mfence` if some amount of time has elapsed and its request on y has not been fulfilled. One could also employ some "tie-breaking" policy so that in the event of circular dependencies, only one processor will ever downgrade its `l-mfence`. One possible tie breaker is to say that a cache controller guarding location x and needs Exclusive on y only downgrades the `l-mfence` if the address of x is smaller than y.

request on the cache line not because another processor wishes to write to the guarded location, but rather because another processor wishes to write to some memory location that happens to be allocated next to the guarded location.

7.3 Formal Specification and Correctness of `l-mfence`

This section formally defines the specification of `l-mfence` and proves that the LE/ST mechanism described in Section 4.3 implements the specification. This section also shows that, in the event that there are concurrent writes during an execution, the LE/ST mechanism does not introduce deadlock. Then, based on the specification of `l-mfence`, one can show that the asymmetric Dekker Protocol using `l-mfence` (as shown in Figure 7-3(a)) achieves mutual exclusion.

Formal specification of `l-mfence`

To formally define the specification of an `l-mfence`, some notation and definitions are required. Throughout this section, we shall use the short hand notation $S = W_P(x)$ to mean that S is a store performed by processor P, writing a value to memory location x. Similarly, the short hand notation $L = R_P(x)$ means that L is a load performed by processor P, reading from memory location x. In cases where it is not important to distinguish which processor performed the operation, the notation $W(x)$ or $R(x)$ is used, omitting which processor performed the memory operation.

To formally define the specification of `l-mfence`, Definition 7.2 first defines the "serialization order" for a given memory location.

Definition 7.2 (Serialization order) *Given a memory location x, the ordering of accesses to x performed by all processors is as follows.*

1. *A load $L = R(x)$ is* ***serialized immediately after*** *a store $S = W(x)$ if and only if L reads the value written by S.*
2. *A store $S = W_P(x)$ is* ***serialized immediately after*** *a store $S' = W(x)$ if at the time* ***completion*** *of S, had P executed a load $L = R_P(x)$, L would have read the value written by S'.*
3. *A load $L = R(x)$ is* ***serialized immediately before*** *a store $S = W(x)$ if there exists a store $S' = W(x)$ such that L is serialized immediately after S', and S is also serialized immediately after S'.*

Given two memory accesses A_1 and A_2 to a memory location x, we say that A_1 is ***serialized before*** *A_2, denoted as $A_1 <_S A_2$, if and only if A_1 is serialized immediately before A_2, or if A_1 is serialized immediately before some other memory access that is serialized before A_2. Vice versa, A_2 is* ***serialized after*** *A_1. The order of all accesses to x performed by all processors is referred to as the* ***serialization order*** *of x.*

While the relation of serialized immediately before / after is not transitive, the serialization order defined in Definition 7.2 is transitive, i.e., if $A_1 <_S A_2$ and $A_2 <_S A_3$, then $A_1 <_S A_3$. Furthermore, the serialization order on a given memory location is globally consistent across all processors, since the serialization because it is defined by the time of completion, not the time of commit. To complete a store to location x, the executing processor P must gain Exclusive state on x, and thus all processors must agree on a single serialization order for the location x.

The ***program order*** of a processor P is defined by the ordering of memory accesses executed in P's instruction stream. Formally, the program order is determined by the time instruction are committed.

Definition 7.3 (Program order) *The **program order** of a processor P is defined by the ordering of memory accesses committed in P's instruction stream. Let A_1 and A_2 be $R_P(x)$ and / or $W_P(y)$ (where x may or may not be the same as y). We say that $A_1 <_P A_2$ if A_1 executed before A_2 in P's program order.*

Given serialization order on all memory locations and the program order of all processors, Definition 7.4 defines the ***inferred order*** of memory accesses for a given processor P, denoted as $\prec_P$:

Definition 7.4 (Inferred order) *Let A_1 and A_2 be memory operations performed by processor P, and let B and C be memory operations performed by other processors $\neq P$. The **inferred order** for P is defined as follows.*

1. *If $A_1 <_P A_2$ then $A_1 \prec_P A_2$.*
2. *If $B <_S A_1$ and $A_1 <_P A_2$, then $B \prec_P A_1 \prec_P A_2$. Similarly, if $A_2 <_S B$ and $A_1 <_P A_2$, then $A_1 \prec_P A_2 \prec_P B$.*
3. *If $A \prec_P B$ and $B \prec_P C$, then $A \prec_P B \prec_P C$.*

For each processor P, the inferred order combines P's program order with the serialization orders of memory locations that P accessed. By definition, two memory accesses are ordered in program order ($<_P$) if and only if both memory accesses are performed by P. In addition, two memory accesses are ordered in serialization order ($<_S$) only if both memory accesses have the same target memory location.[6] Finally, the inferred order relation is transitive. The inferred order does not provide a total order on all accesses performed by all processors. Rather, it provides a partial order for each processor P, that agrees with P's program order and the serialization orders of all the memory locations accessed by P.

Definition 7.5 (Consistent inferred orders) *The inferred orders of processors P_1 and P_2 are **consistent** with respect to specific memory accesses A_1 and A_2 if all the following conditions are satisfied:*

1. *A_1 and A_2 are ordered by both inferred orders $\prec_{P_1}$ and $\prec_{P_2}$,*
2. *A_1 and A_2 were performed by the same processor, and*
3. *If $A_1 \prec_{P_1} A_2$ then $A_1 \prec_{P_2} A_2$.*

The last condition guarantees that if A_1 precedes A_2 in one inferred order, it must precedes A_2 in the other order, and vice verse, so that the relative ordering of A_1 and A_2 in both orders agree.

Let's go back to the TSO and PO models that are the assumed architecture for the `l-mfence` implementation. It is due to the TSO and PO reordering that the inferred orders of different processors may be inconsistent. The ordering principles of TSO and PO defined in Definition 7.1 (Section 7.1) lay out the discrepancies between the inferred orders that two different processors may deduce. Memory fences were created to provide consistency in the inferred orders, by enforcing consistency between particular memory accesses across the inferred orders of all processors.

The difference in the inferred orders can be demonstrated using an example. Assuming `mfence` or `l-mfence` is not used, imagine the following scenario. A processor P_1 committed $W_{P_1}(x)$ and then committed $R_{P_1}(y)$, and another processor P_2 committed $W_{P_2}(y)$ and then committed $R_{P_2}(x)$. By Definition 7.3, we have:

$$W_{P_1}(x) \quad <_{P_1} \quad R_{P_1}(y)\ , \tag{7.1}$$

$$W_{P_2}(y) \quad <_{P_2} \quad R_{P_2}(x)\ . \tag{7.2}$$

[6] In this case, "only if" but not "if and only if" is used, because two reads to the same location may not be ordered.

However, at the end of P_1 and P_2 execution, it is possible to reach the following serialization order given the TSO and PO re-orderings and Definition 7.2:

$$R_{P_2}(x) \quad <_S \quad W_{P_1}(x)\,, \tag{7.3}$$

$$R_{P_1}(y) \quad <_S \quad W_{P_2}(y)\,. \tag{7.4}$$

Then, by Definition 7.4, we have:

$$R_{P_2}(x) \prec_{P_1} W_{P_1}(x) \prec_{P_1} R_{P_1}(y) \prec_{P_1} W_{P_2}(y)\,, \tag{7.5}$$

$$R_{P_1}(y) \prec_{P_2} W_{P_2}(y) \prec_{P_2} R_{P_2}(x) \prec_{P_2} W_{P_1}(x)\,. \tag{7.6}$$

P_1's inferred order (7.5) is obtained via orderings (7.3), (7.1), and (7.4). Similarly, P_2's inferred order (7.6) is obtained via orderings (7.4), (7.2), and (7.3). Therefore, even though from P_1's perspective, $W_{P_1}(x) \prec_{P_1} R_{P_1}(y)$, P_2 observed the opposite order. Similarly, even though from P_2's perspective, $W_{P_2}(y) \prec_{P_2} R_{P_2}(x)$, P_1 observed the opposite order. These differences in their inferred orders are consistent with Principle 4 of Definition 7.1. In addition, P_1 observed that $W_{P_1}(x) \prec_{P_1} W_{P_2}(y)$, whereas P_2 observed the opposite order, which is consistent with Principle 6 of Definition 7.1.

As mentioned at the end of Section 7.1, correct usage of a memory fence typically involves a pair of `mfence` instructions. Using the same example, one can also show that P_1 and P_2 can observe different ordering of memory accesses if only one `mfence` is used. Assume that P_1 executed a `mfence` between $W_{P_1}(x)$ and $R_{P_1}(y)$, but P_2 did not use `mfence`. At the end of P_1 and P_2 executions, it is still possible to end up with serialization orderings (7.3) and (7.4), because even though an `mfence` executed by P_1 ensures that $R_{P_1}(y)$ did not commit until $W_{P_1}(x)$ completed, $W_{P_1}(x)$ could have still completed *after* $R_{P_2}(x)$ committed, resulting in ordering 7.3, and $R_{P_1}(y)$ could have still committed *before* $W_{P_2}(y)$ completed, resulting in ordering 7.4. Given the same program orders 7.1 and 7.2 and the same serialization orders (7.3) and (7.4), $P1$ and P_2 inferred orders are still different, even though P_1 used an `mfence`. Had P_2 *also* executed an `mfence` between $W_{P_2}(y)$ and $R_{P_2}(x)$, this scenario could not have happened, and both processors would have consistent inferred orders with respect to their read and write operations.

Now Definition 7.6 defines the specification of `l-mfence` formally.

Definition 7.6 (`l-mfence` specification) *Let C be a program execution that does not contain concurrent writes, S be a store associated with an `l-mfence` executed by processor P_1, and A be a memory access also executed by P_1. Let P_2 be another processor whose inferred order enforces an ordering between A and S. Let B_1 and B_2 be the two memory operations executed by P_2 accessing the same locations as A and S that lead to the ordering of A and S in P_2's inferred order. The `l-mfence` enforces that the inferred orders of P_1 and P_2 are consistent with respect to A and S if they are also consistent with respect to B_1 and B_2.*

The condition that the inferred orders of P_1 and P_2 are consistent with respect to B_1 and B_2 performed by P_2 implicitly states that if the relevant memory accesses are a write followed by a read, there ought to be an `mfence` (or `l-mfence`) between them to prevent reordering in P_1's inferred order. This condition follows from the correct usage of `mfence` (and `l-mfence`), that involves a pair of fences. If the condition is met, then an `l-mfence` with a store $S = W_{P_1}(x)$ enforces an inferred order between S and another access A performed by P_1, that is consistent with the inferred order of P_2.

Correctness proof of the LE/ST mechanism

First let's see some definitions and lemmas that will help us show that the LE/ST mechanism (which includes the code sequence shown in Figure 7-3(b)) implements the specification of `l-mfence`.

Definition 7.7 *Given a particular instance of `l-mfence` with guarded location x implemented with the LE/ST mechanism, a link for the `l-mfence` is* **set** *if `LEBit` contains* 1, *`LEAddr` contain x, and the executing processor's private cache holds the cache line for x (i.e., in Exclusive or Modified state). If any of these conditions is not met, the link is* **clear**.

Lemma 7.8 *Given a particular instance of `l-mfence` with guarded location x, if `LEBit` contains* 1 *when the associated store commits (line K1.4), the link must be set.*

PROOF. By executing the instructions in lines K1.1–K1.3, the executing processor sets up the link. Since `LEBit` is set as the **first** instruction of the `l-mfence` execution, if the link was broken at any point before the commit of `st` in line K1.4, the LE/ST mechanism clears `LEBit` as part of the protocol to break the link. Once the link is broken, `LEBit` is never set again until the next instance of `l-mfence`. □

Lemma 7.9 *The LE/ST mechanism maintains the ordering principles defined by the TSO / PO memory model described in Section 7.1.*

PROOF. The ordering principles are maintained by the fact that instructions are committed in order, and a processor's store buffer is flushed in FIFO order. The LE/ST mechanism employs regular loads[7], stores, and memory fences, which do not interfere with the commit ordering of instructions and the FIFO ordering of the store buffer. Thus, the TSO / PO ordering principles are maintained. □

Lemma 7.10 *Let $S = W_{P_1}(x)$ be a store associated with an `l-mfence` performed by processor P_1. Let $L = R_{P_2}(x)$ be a read operation performed by processor P_2 and committed after P_1 gained Exclusive state on x (line K1.3 in Figure 7-3(a)). The LE/ST mechanism ensures that, before P_1 commits the next instruction following this `l-mfence`, either the store S in line K1.4 is already complete, or L is serialized after S, i.e., $S <_S L$.*

PROOF. Since the lemma assumes that P_2 executes the read after P_1 gained Exclusive state on x, it must be that the cache controller of P_2 sent a request to downgrade x to Shared to P_1. Let's look at the link situation when S commits, and examine the actions of P_1 when it receives P_2's request. There are two cases to consider: either the link is clear at the time when S commits, or the link is still set.

1. **Link is clear when S commits**. The link can be clear only if P_2's request was detected after the Exclusive state was gained (line K1.3) but before S had a chance to commit. By Lemma 7.8, we know that if the link is clear, the `LEBit` must be 0. Therefore, by the implementation of the LE/ST mechanism (Figure 7-3(b)), the condition for the branch (line K1.5) is false, and thus P_1 must execute an `mfence` in line K1.6 right after it commits S, causing S to complete before the next instruction (line K3 in Figure 7-3(a)) commits. Note that in this case L is serialized before S.
2. **Link is set when S commits**. If the link is set, then by Definition 7.7, we know that P_1 still has x in Exclusive / Modify state when S commits. By the LE/ST mechanism, this means that when the cache controller receives P_2's downgrade request, P_1's cache controller must notify the processor when such a request arrives, and upon notification, P_1 clears the link, flushes its store buffer to complete S, and replies to the cache controller. After that, P_1's cache controller responds to the downgrade request. Thus, L must be serialized after S.

[7] As explained in Section 7.1, the `le` instruction is very similar to a regular load and can be implemented using the existing architecture and cache coherency protocol.

□

Theorem 7.11 is the main theorem that shows that the LE/ST mechanism implements the `l-mfence` specification.

Theorem 7.11 *The LE/ST mechanism implements `l-mfence` as specified in Definition 7.6. That is, let C be a program execution that does not contain concurrent writes, S be a store associated with an `l-mfence` executed by processor P_1, and A be a memory access also executed by P_1. Let P_2 be another processor whose inferred order enforces an ordering between A and S. Let B_1 and B_2 be the two memory operations executed by P_2 accessing the same locations as A and S that lead to the ordering of A and S in P_2's inferred order. An `l-mfence` implemented using the LE/ST mechanism guarantees that the inferred orders of P_1 and P_2 are consistent with respect to A and S if they are also consistent with respect to B_1 and B_2.*

PROOF. The proof is split into two cases: one that proves that the inferred orders of all processors are consistent with respect to accesses that happened before an `l-mfence` in the program order of the executing processor, and the other proves the same about accesses after the `l-mfence`.

Case 1: $A \prec_{P_1} S$**.** Since $A \prec_{P_1} S$ and both A and S were executed by P_1, it must be that $A <_{P_1} S$. If A is a store, then by the TSO and PO ordering Principle 5 in Definition 7.1 and by Lemma 7.9, it is impossible for another processor P_2 to infer that $S \prec_{P_1} A$.

If A is a load, in order for P_2 to infer that $S \prec_{P_2} A$, it must be that S completed before A committed. Given our assumption that $A <_{P_1} S$, by the TSO Principle 2 in Definition 7.1 and by Lemma 7.9, this cannot be the case.

Case 2: $S \prec_{P_1} A$**.** Since $S \prec_{P_1} A$ and both S and A were executed by P_1, it must be that $S <_{P_1} A$. If A is a store then by the TSO and PO ordering Principle 5 in Definition 7.1 and by Lemma 7.9, it is impossible for another processor P_2 to infer that $A \prec_{P_1} S$. Thus, we only need to consider the case where A is a load, which can be reordered with older stores by the TSO and PO ordering principles.

Without loss of generality, let $S = W_{P_1}(x)$ and $A = R_{P_1}(y)$, where x is the location guarded by the `l-mfence`. The case where $x = y$ is trivial — assuming $x = y$, since both A and S are executed by P_1, A must have observed the value written by S due to store-buffer forwarding, and no reordering could have occurred, since in this case $S <_S A$. Hence, another processor P_2 must also infer that $S \prec_{P_2} A$ and the inferred orders are consistent with respect to A and S. Thus, we should consider the case $x \neq y$.

There are three possible pairing of B_1 and B_2 executed by P_2 that allows P_2 to infer and ordering between A and S, and we consider them one by one.

1. $B_1 = R_{P_2}(y)$ and $B_2 = R_{P_2}(x)$. Given that the inferred orders of P_1 and P_2 are consistent with respect to B_1 and B_2, there are two cases to consider: either $S <_S B_2$ or $B_2 <_S S$, depending on the value read by B_2. If $S <_S B_2$, there is no placement of A and B_1 that could force P_2 to infer $A \prec_{P_2} S$. Thus, the inferred orders of P_1 and P_2 are consistent with respect to A and S.
   If $B_2 <_S S$, then it must be that S committed after B_2 committed. Moreover, it must be that B_2 committed before the `l-mfence` executed by P_1 on x gained the Exclusive state on x. This follows from Lemma 7.10, which says that if B_2 committed after the Exclusive state was gained by P_1, then either S is completed by the time A committed, or B_2 is serialized after S. Since we assume that $B_2 <_S S$, it must be that S is completed by the time A committed. Thus, P_2 cannot possibly infer that $A \prec_{P_2} S$. Thus, the inferred orders of P_1 and P_2 are also consistent with respect to A and S in this case.

2. $B_1 = W_{P_2}(y)$ and $B_2 = W_{P_2}(x)$. The case that $B_2 <_{P_2} B_1$ is trivial, since in this case, nothing can force P_2 to infer that $A \prec_{P_2} S$, no matter what the serialization order between B_1 and A, and B_2 and S are. Thus, let's consider the case $B_1 <_{P_2} B_2$.
   Let's assume for the sake of contradiction that P_2 infers an ordering inconsistent from P_1 with respect to A and S. That is, $A \prec_{P_2} S$, which can only be true if $A <_S B_1$ and $B_2 <_S S$. Since both B_1 and B_2 are stores, by Principle 5 in Definition 7.1 and by Lemma 7.9, the inferred orders of P_1 and P_2 must be consistent with respect to B_1 and B_2. Furthermore, since B_2 is a store to location x, guarded by S, based on the assumption that C contains no concurrent write, B_2 cannot occur while the link for P_1 is in effect. If B_2 reached the cache before the link was set, then B_2 must have completed before S committed, which is before A committed. This leads to a contradiction to our assumption — since B_1 completed before B_2 completed, which is before S committed, which is before A committed, it cannot be possible to have $A <_S B_1$. Thus, P_1 and P_2 must infer a consistent ordering with respect to A and S. On the other hand, if B_2 reached cache when the link was no longer set, that means either S has completed at this point, or S has not committed but would complete before A commits since the link was broken. If S has completed, then it must be that $S <_S B_2$, which leads to a contradiction to our assumption, $B_2 <_S S$. If S has not committed but would complete before A commits, this again leads to a contradiction that $A <_S B_1$, since B_1 must be completed by the time B_2 reached the cache.
3. $B_1 = W_{P_2}(y)$, $B_2 = R_{P_2}(x)$. The only interesting case here is when $B_1 <_{P_2} B_2$. This is because, if $B_2 <_{P_2} B_1$, then P_2 can always infer that $S \prec_{P_2} A$, no matter what the serialization orders between B_1 and A, and B_2 and S are. Thus, we focus on the case where $B_1 <_{P_2} B_2$. Again, let's assume for the sake of contradiction that the inferred orders of P_1 and P_2 are not consistent with respect to A and S. That is, P_2 infer that $A \prec_{P_2} S$, which can only be true if $A \prec_{P_2} B_1 \prec_{P_2} B_2 \prec_{P_2} S$. Furthermore, since the lemma guarantees that the inferred orders of P_1 and P_2 are consistent with respect to A and S *only if* they are also consistent with respect to B_1 and B_2, we must also assume that $B_1 \prec_{P_1} B_2$.
   To achieve this assumption, all the following constraints must hold:

   (a) A commits before B_1 completes,
   (b) B_2 commits before S completes,
   (c) B_1 completes before B_2 commits.

   Note that constraint (c), if not occurring naturally, can be enforced by either inserting an `mfence` or an `l-mfence` between B_1 and B_2. An `mfence` guarantees that the next instruction after the `mfence` commits only after all instructions before the `mfence` have completed, which meets the constraint.
   On the other hand, if an `l-mfence` is used to serialize B_1 and B_2, by Lemma 7.10, either $B_1 <_S A$, or B_1 completes before B_2 commits. Since $B_1 <_S A$ breaks the first constraint, which leads to a contradiction, it must be that B_1 completes before B_2 commits.
   Taking all the constraints together, A must commit before S completes for P_1 and P_2 to infer inconsistent orders with respect to S and A.
   Let's examine the `l-mfence` link status of P_1 when it commits A.

   - The link is clear when A commits. By Lemma 7.10, S must be completed before the next instruction A commits, which means that S is already completed when A commits. This leads to a contradiction to that A must commit before S completes, and thus P_2 cannot infer that $A \prec_{P_2} S$.
   - The link is set when A commits. This means that S has been committed but not yet completed, and that P_1 holds the guarded location x in Exclusive or Modify state. Let Z

be the next access to x performed by any processor. If $Z = R(x)$, by Lemma 7.10, it must be that $S <_S Z$. If $Z = B_2$ then constraint (b) is violated, which leads to a contradiction. If $Z \neq B_2$, since Z is the next access to x after S, and since constraints (a) and (c) dictate that A commits before B_2 commits, it must be that $S <_S Z <_S B_2$. This again violates constraint (b) and leads to a contradiction. If $Z = W(x)$ is the next access to x, since the lemma assumes that C does not contain concurrent writes, it must be that S completed before Z completed, and therefore the link was not set when Z completed. Thus, $S <_S Z$. Following the same reasoning as the case where $Z = R(x)$ and $Z \neq B_2$, this again leads to a contradiction. Thus, P_2 cannot infer $A \prec_{P_2} S$.

In all cases, we have shown that the inferred orders of P_1 and P_2 must be consistent with respect to A and S, assuming that they are also consistent with respect to B_1 and B_2 executed by P_2 accessing the same locations as A and S. Thus, the LE/ST mechanism correctly implements the specification of `l-mfence` as specified in Definition 7.6. □

Theorem 7.11 shows that the LE/ST mechanism correctly implements the specification of `l-mfence`, which provides guarantees only for computations that do not contain concurrent writes. This is necessary to avoid deadlock due to the additional dependencies that the LE/ST mechanism creates. Next, we show next that the LE/ST mechanism does not introduce system deadlock.

Theorem 7.12 *The LE/ST mechanism does not introduce system deadlock.*

PROOF. The LE/ST mechanism is implemented using mostly instructions ready available in the architecture, where each instruction can make progress by itself. The only situation in which the LE/ST mechanism introduces a new dependency is when the link is set for an `l-mfence` executed by a processor P_1, the store associated with the `l-mfence` has been committed into P_1's store buffer, and a different processor P_2 requests P_1 to invalidate its guarded location. In this case, P_1 cannot satisfy the incoming invalidation request from P_2 until all its outgoing requests to get Exclusive states on locations in the store buffer before the guarded location are satisfied. This is because P_1 must flush the locations in its store buffer in FIFO order up to and including the guarded location before it can invalidate the guarded location.

A concurrent write is easily detected by the LE/ST mechanism when the cache controller receives an invalidation request for the guarded location while the link is set. When invalidation request to the guarded location is detected, the LE/ST mechanism notifies the processor, which in turn just clears the link and let the cache controller reply to the invalidation request immediately. Since the cache controller no longer need to wait for other locations in the store buffer to be completed before it responds to the invalidation request, the system does not deadlock. □

Given that the LE/ST mechanism implements the specification as described in Definition 7.6, it is not difficult to see that the asymmetric Dekker protocol shown in Figure 7-3(a) guarantees mutual exclusion. Since the primary thread uses an `l-mfence` between the store to x and the read from y, and the secondary thread uses an `mfence` between the store to y and read from x, according to the specification, we know that the primary thread and the secondary thread agree upon the orderings of $W_{P_1}(x) \prec_{P_1,P_2} R_{P_2}(y)$ and $W_{P_2}(y) \prec_{P_1,P_2} R_{P_2}(x)$. As long as they agree on the orderings of the relevant memory accesses, mutual exclusion is guaranteed.

The asymmetric Dekker protocol is designed to optimize away the overhead incurred onto the primary thread at the expense of additional overhead on the secondary thread, which is advantageous for applications that exhibit asymmetric synchronization patterns. Hence, an `mfence` is used in the secondary thread instead of an `l-mfence` to avoid incurring additional overhead on the primary thread. If the secondary thread was using an `l-mfence`, the primary thread may need to wait for the secondary thread to flush its store buffer when it attempts to read y in line K3. Nevertheless, the secondary thread has the option of executing the mirrored code (using `l-mfence(&y,1)` in line J2), and the protocol still provides mutual exclusion in such case.

7.4 An Empirical Evaluation of Location-Based Memory Fences

This section presents an empirical evaluation of a software-based implementation of location-based memory fences, which have two purposes. First, the evaluation demonstrates that performance benefits can be gained using location-based memory fences instead of program-based memory fences. Second, the evaluation allows us to analyze the expected performance of the proposed hardware mechanism, based on performance results of the software implementation.

The software prototype of `l-mfence` used in this section is implemented using software signals. This implementation is applied to two applications that exhibit asymmetric synchronization patterns, and their performance is evaluated during serial and parallel execution. All experiments were conducted on an AMD Opteron system with 4 quad-core 2 GHz CPU's having a total of 8 GBytes of memory. Each core on a chip has a 64-KByte private L1-data-cache and a 512-KByte private L2-cache, and all cores on a chip share a 2-MByte L3-cache.

When executed serially, the benchmarks perform better using the software implementation of `l-mfence` instructions than their counterparts using ordinary `mfence` instructions. The reason for these results is that the software prototype incurs effectively no overhead on the executing thread when it runs serially. When executed in parallel, even though the communication overhead of the software prototype is high, some benchmarks still see performance benefit from using the software implementation of `l-mfence` instructions. While the software implementation is feasible, the LE/ST mechanism should significantly enhance the performance of the benchmarks in parallel executions (without affecting the results in the serial executions), and enable a larger class of programs to benefit from `l-mfence`.

This section briefly summarizes the software prototype, compares the overhead between the software prototype and the LE/ST mechanism, describes the experimental results based on the software prototype, and discusses how the outcomes would differ with the LE/ST mechanism.

Software prototype of `l-mfence`

The software prototype of the location-based memory fences is implemented using signals, similar to the approach proposed in [34]. The software prototype must correctly capture two main effects. First, the primary thread must not reorder the write and the read at the compiler level. This can be achieved simply by inserting a compiler fence at the appropriate location. Second, before the secondary thread attempts to read the variable written by the primary thread, it must cause the primary thread to serialize, and only proceed with the read *after* the primary thread has performed the serialization. This is achieved via signals — a software signal generates an interrupt on the processor receiving the signal, and the processor flushes its store buffer before calling the signal handling routine. Thus, the secondary thread sends a signal to the primary thread and waits for an acknowledgment by spinning on a shared variable. Upon receiving the signal (which implicitly flushes the store buffer), the primary thread executes a user-defined signal handler, which sets the shared variable as an acknowledgment, thereby allowing the secondary thread to resume execution.

Overhead comparisons between the software prototype and the LE/ST mechanism

Let's compare the overhead between the software prototype and the LE/ST mechanism in two cases: when the primary thread executes alone, and when other secondary threads exist in the same context.

When the primary thread executes alone, the software prototype incurs negligible overhead from the compiler fence, while the LE/ST mechanism would incur small additional overhead from setting the link, performing the load-exclusive, and taking the branch. Nevertheless, this additional overhead should be negligible as well, since the target cache line of the load stays in the primary processor's cache, and the branch is a predictable branch for the most part.

During parallel execution, the software implementation using signals would incur much higher communication overhead compared to the LE/ST mechanism. In the software implementation, the communication overhead includes the secondary thread sending the signal and waiting for the primary thread to flush its store buffer and handle the signal. Furthermore, this software implementation also slows down the primary thread whenever communication occurs, because the primary thread must handle the signal (which entails crossing between kernel and user modes four times to execute a user-defined signal[8]) while the secondary thread waits. The estimated cost of a single round trip communication is on the order of $10,000$ cycles on the system in which the experiments were run. On the other hand, the round trip time in the LE/ST mechanism involves waiting for the cache controllers of the two processors to send and handle messages (akin to a L1 cache miss / L2 cache hit), and for the primary processor to flush its store buffer. I ran a synthetic benchmark to simulate this round trip time, which costs about 150 cycles on the system where the experiments were conducted. Moreover, the performance impact on the primary processor is negligible: it just needs to flush the store buffer and regain the cache line the next time around; the processor performance is not affected by the cache controller listening to cache traffic and handling messages.

Performance benefit can be gained using `l-mfence` if the latency avoided by the primary thread is greater than the communication overhead borne by the secondary thread. Putting the overhead comparison into the context of benchmark execution, the software implementation requires significantly more asymmetry in the benchmarks in order to obtain performance gain than the LE/ST mechanism.

Applications overview

Two applications are used to evaluate the location-based memory fences using the software prototype — the asymmetric Cilk-5 runtime system and an asymmetric multiple-reader single-writer lock.

For the first application, the open-source Cilk-5 runtime system [49][9] is modified to incorporate `l-mfence` into the Dekker-like protocol employed by its work stealing scheduler, referred to as the ***ACilk-5 runtime system***. In a work-stealing scheduler, when a thief (the secondary thread) needs to find more work to do, it engages in an augmented Dekker-like protocol with a given victim (the primary thread) in order to steal work from the victim's deque. Assuming the benchmarks contains ample parallelism, a victim would access its own deque much more frequently than a thief, because steals occur infrequently.

The second application uses an ***asymmetric multiple-reader single-writer lock***, where the lock is biased towards the readers, henceforth referred to as the ***ARW lock***. From time to time, a reader (the primary thread) turns into a writer (the secondary thread), and attempts to acquire the ARW

[8] One could modify the operating system to cut the signal handling overhead down by half (crossing two times instead of four), but that would still be on the order of thousands of cycles.

[9] The open-source Cilk-5 system is available at http://supertech.csail.mit.edu/cilk/cilk-5.4.6.tar.gz.

Benchmark	*Input*	*Description*
`cholesky`	4000/40000	Cholesky factorization
`cilksort`	10^8	Parallel merge sort
`fft`	2^{26}	Fast Fourier transform
`fib`	42	Recursive Fibonacci
`fibx`	280	Alternate between fib(n-1) and fib(n-40)
`heat`	2048×500	Jacobi heat diffusion
`knapsack`	32	Recursive knapsack
`lu`	4096	LU-decomposition
`matmul`	2048	Matrix multiply
`nqueens`	14	Count ways to place N queens
`qsort`	10^8	Parallel quick sort
`rectmul`	4096	Rectangular matrix multiply
`strassen`	4096	Strassen matrix multiply

Figure 7-5: The 13 benchmark applications.

lock in the write mode by engaging in an augmented Dekker protocol with each of the registered readers.

Evaluation using ACilk-5

13 benchmarks are used to evaluate the effect of location-based memory fences, comparing how ACilk-5 performs against Cilk-5 running these benchmarks. Figure ?? provides a brief description of each benchmark.

Figure 7-6(a) compares the performance of the benchmarks running on ACilk-5 and Cilk-5 when executed serially. Figure 7-6(b) shows a similar performance comparison when executed on 16 cores. For each measurement, the mean of 10 runs is used (with standard deviation of less than 3%). A value below 1 means that the benchmark runs faster on ACilk-5 than on Cilk-5.

Not surprisingly, when executed serially, benchmarks on ACilk-5 run faster, because the victim executes on the fast path with virtually no overhead from memory fences. The improvement that ACilk-5 exhibits over Cilk-5 when running a given benchmark is directly related to the ratio between the overall work in a given benchmark and the number of fences avoided in the benchmark (which corresponds to the the number and the granularity of parallel tasks that the benchmark generates). The fewer the number of memory accesses performed under a given fence, the more saving gained from avoiding the fence. All these benchmarks except for `fib`, `fibx`, and `knapsack` have their base case coarsened (so as to generate enough parallel tasks and avoid parallel overhead when there is enough parallelism), so the ratio of work per fence is high. On the other hand, `fib` is specifically designed to measure the spawn (for generating parallel tasks) overhead, and the number suggests that the spawn overhead is cut by half if the fence is avoided. I believe the numbers will be comparable if `l-mfence` were implemented using the LE/ST mechanism.

Figure 7-6(b) shows the same performance comparison when executed on 16 cores. When executed in parallel, the software implementation of `l-mfence` incurs an additional communication overhead for every steal attempt (which impacts both the victim and the thief). Despite the communication overhead, many benchmarks still exhibit saving or stay even (meaning that savings and overhead even out). The three exceptions are `cholesky`, `heat`, and `lu`. There are two factors at play here. First, while the work-first principle [49] states that one should put the scheduling overhead onto the steal (thief's) path instead of onto the work (victim's) path, one must be able to amortize the overhead against successful steals in order to obtain good scalability. In the case of

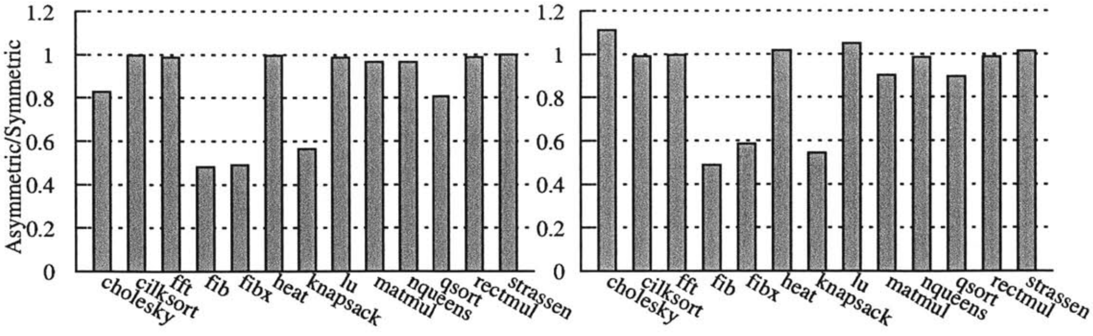


Figure 7-6: (a) The relative serial execution time of the ACilk-5 runtime system compared to the original Cilk-5 runtime system for 13 Cilk benchmarks. **(b)** The relative execution time of the ACilk-5 runtime system compared to the original Cilk-5 runtime system for 13 Cilk benchmarks on 16 cores. A value below 1 means that the application runs faster on ACilk-5 than on Cilk-5; a value above 1 means the other way around. Each value is calculated by normalizing the execution time of the benchmarks on ACilk-5 with that on Cilk-5.

`cholesky` and `lu`, much of the communication overhead did not translate into successful steals — only 53.6% of signals sent in `cholesky` turn into successful steals, and only 72.8% for `lu` (while other benchmarks have over 90%). As a result, the benchmarks do not scale as well. Second, while over 90% of the signals sent in `heat` translate to successful steals, the number of fences avoided per signal sent is much smaller compared to other benchmarks, so the communication overhead incurred by `l-mfence` outweighs the benefit. Given that the LE/ST mechanism has much smaller communication overhead and impacts only the thief, I believe both problems would be avoided.

Evaluation using ARW lock

The next application of location-based memory fences is the ARW lock, where we compare the read throughput between the ARW lock and its symmetric counterpart: the same design but using an `mfence` for the primary thread in the Dekker protocol instead of an `l-mfence`, henceforth referred as the ***SRW lock***. The application works as follows. Each thread performs read operations most of the time, and only occasionally performs a write. In the tests, the threads read from and write to an array with 4 elements. The read-to-write ratio is an input parameter to the microbenchmark: assuming the ratio is $N:1$, and there are P threads executing, then for every N/P reads, a thread performs a write. With each configuration, the microbenchmark is run for 10 seconds to measure the overall read throughput.

Figure 7-7(a) shows the throughput comparison between the ARW lock and the SRW lock. In the software implementation of `l-mfence`, since a request for serialization translates to a signal, the writer ends up signaling a list of readers and waiting for their responses one by one, which becomes a serializing bottleneck. This is particularly inefficient when the thread counts is high, and the read-to-write ratio is low (less asynchronous), since the communication overhead outweighs the benefit from avoiding fences.

I believe that the lack of scalability is again due to the high communication overhead in the software implementation. To confirm this, I devised an ARW lock that implements a *waiting heuristic*: when a writer wants to write, instead of sending signals to the readers immediately, it first indicates intent to write and spin-waits to see if any reader responds, acknowledging the writer's intent to write. Only after spin-waiting for awhile, the writer sends signals to readers who have not acknowledged. The ARW lock with this heuristic is referred as the ***ARW+ lock***.

Figure 7-7(b) shows the throughput comparison between the ARW+ lock and the SRW lock.

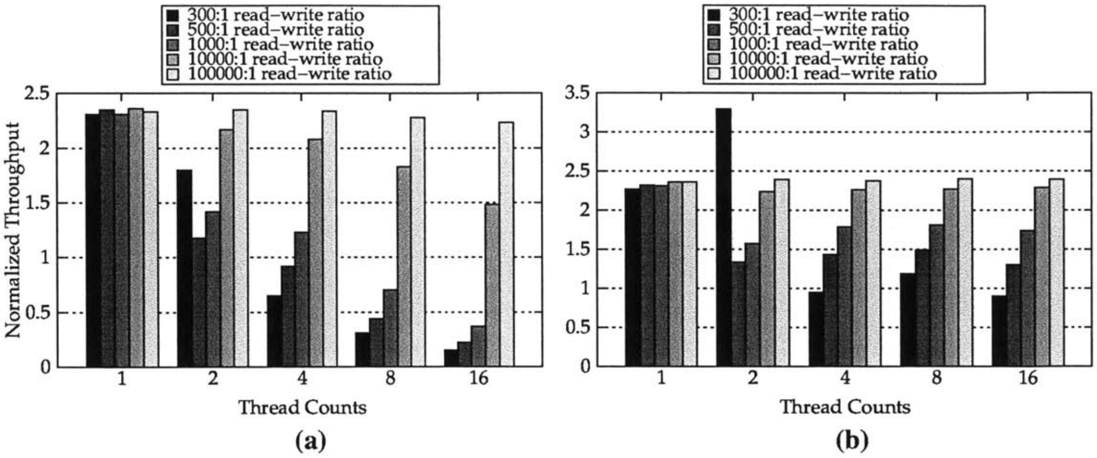


Figure 7-7: **(a)** The relative read throughput of execution using the ARW lock compared to that using the SRW lock. **(b)** The relative throughput of execution using the ARW+ lock (i.e., the ARW lock with the waiting heuristics) compared to that using the SRW lock. A value above 1 means that the ARW lock / ARW+ lock performs better; a value below 1 means that the SRW lock performs better. Each value is calculated by normalizing the read throughput from the execution using the ARW lock by that using the SRW lock.

A value above 1 means that the ARW+ lock performs better. There are two main trends to notice. Indeed, the ARW+ lock scales much better and consistently has higher throughput compared to the SRW lock, except for the 300 : 1 read / write ratio (which is close to 1). One notable outlier in Figure 7-7(b) is the data point for 300 : 1 ratio with two threads, which has much higher throughput compared to other thread counts. This is due to the fact that when there are only two threads, the writer end up receiving the acknowledgment most of the time and does not need to send signals.

While the waiting heuristic seems to work well in the microbenchmarks, if the reader does not access the lock frequently, the heuristic would not help as much, because a thread would only check for pending intent during lock acquire and release. With that in mind, the results inspire confidence that the ARW lock should perform and scale well when the `l-mfence` is implemented with the LE/ST mechanism.

7.5 Related Work

This work is closely related to studies performed on biased locks and asymmetric synchronization, so this section focuses on these studies. Several researchers studied this area, mainly in the context of improving performance for Java locks.

[134] describes a fast biased lock algorithm, which allows the primary thread to avoid executing memory fences, until a secondary thread attempts to enter the critical section. In this case, the secondary thread must wait for the primary thread to grant access in order to continue execution. While this request and grant protocol is performed via shared variables and is therefore fairly efficient, this implementation can potentially deadlock if the biased lock is nested within another lock (or any resource that can block). Imagine the following scenario: suppose that a primary thread and secondary thread try to acquire a lock *A* and then an biased lock *B* (biased towards the primary thread). If the secondary threads acquires *A* first, the system deadlocks, because the secondary thread must wait for the primary thread to set the grant bit while the primary thread is blocked on acquiring lock *A*, which is held by the secondary thread.

The studies in [36] and [119] describe similar biased lock implementations, where the owner of the lock is on the fast path for accessing the lock, and other threads need to revoke it and compete for ownership, and the lock ownership may transfer. Both algorithms use the "collocation" trick, where the status field and the lock field are allocated on the same word. They first write to one field and then the whole word is read. The correctness of the algorithm depends on the fact that hardware typically does not reorder a read before an older write when the addresses overlap. This collocation trick, while interesting, is not guaranteed to be safe, and on systems where this trick works correctly, it always forces a memory fence to be issued regardless of whether there is contention [33].

Serialization using signal and notify was proposed in [34], along with other more heavy-weight serialization mechanisms. Their work focus on software means to cause serialization in another thread, while decreasing synchronization overhead on the primary thread in applications that exhibit asymmetric synchronization patterns.

Finally, Lin et al. [99] propose a hardware mechanism for conditional memory fences, whose aim is also to reduce the overhead of memory fences when synchronization is unnecessary. In [99], however, the assumption is that the compiler would automatically insert memory fences in order to enforce sequential consistency everywhere, and there may be multiple outstanding memory fences for a given thread at a given moment. Thus, their hardware mechanism is much more heavyweight compared to the LE/ST mechanism for implementing `l-mfence`. The LE/ST mechanism, on the other hand, aims to be lightweight and does not focus on enforcing sequential consistency everywhere automatically.

7.6 Conclusion

This chapter investigates in location-based memory fences, which aim to reduce the overhead incurred by memory fences in parallel algorithms. Location-based memory fences are particularly well-suited for algorithms that exhibit asymmetric synchronization patterns. This chapter describes a hardware mechanism to support location-based memory fences, proves its correctness and evaluates the feasibility of the fences using a software prototype. The evaluation with the software prototype inspires confidence that the suggested LE/ST mechanism for supporting location-based memory fences in hardware is worth considering.

Finally, location-based memory fences lend itself to a different way of viewing programs compared to the traditional program-based memory fences. It would be interesting to investigate what other algorithms can benefit from location-based memory fences, as well as other mechanisms that exploit the location-based model.

Chapter 8

Conclusion

This dissertation has explored five different memory abstractions:

1. TLMM-based cactus stacks that interoperate with linear stacks (Chapter 3),
2. memory-mapped reducers (Chapter 4),
3. reducer arrays (Chapter 5),
4. ownership-aware transactions (Chapter 6), and
5. location-based memory fences (Chapter 7).

These memory abstractions ease the task of parallel programming, either directly, by mitigating the complexity of synchronization, and/or indirectly, by enabling one to design a concurrency platform which utilizes resources more efficiently than one could do without the memory abstraction.

I would like to revisit the definition of memory abstractions and provide some perspective on the work explored in this dissertation. At the beginning of this dissertation, I defined memory abstraction to be an abstraction layer between the program execution and the memory that provides a different view of a memory location depending on the execution context in which the memory access is made. This definition does not specify where a memory abstraction should be implemented. There can be many layers along the software stack between the raw memory provided by the hardware system and the program execution. A memory abstraction can be implemented within a specific layer or with support across multiple layers and have the topmost layer providing an interface for the program execution to interact with the memory abstraction.

In a sense, a memory abstraction can be viewed as a contract defined between a program execution and the system layer on which the program is executing. The contract defines how the program execution may interact with the memory and what kind of guarantees the underlying system provides. Here, the system layer can be anything within the software stack — the underlying hardware architecture, the operating system, a virtual machine, or a concurrency platform. Taking this view of a memory abstraction, one begins to see that memory abstractions constitute some integral parts of the system that we use on a daily basis, such as the virtual memory mechanism provided by the operating system or the automatic memory management in a managed runtime environment.

Virtual memory [44,78][1] is a memory abstraction provided by an operating system for programs running directly on top of the operating system. Virtual memory abstracts away the underlying raw physical memory so that the addresses as seen by the program are nicely decoupled from the addresses of the physical memory provided by the underlying hardware. This decoupling provided by the virtual memory significantly simplifies the task of programming. It frees the programmer

[1] Articles from Peter J. Denning [31,32] provide a nice overview and historical context for the development of virtual memory.

from worrying about the problem of ***overlaying*** — replacing a block of code or data with another when the program or data accessed by the program is larger than the main memory supported by the hardware. Because the problem of overlaying, or address space allocation, is handled automatically by the operating system, modular programming becomes possible, where components of programs can be compiled separately and reused. The virtual memory mechanism also provides an additional layer of safety. An operating system employing the virtual memory mechanism can seamlessly time-share among multiple executing processes, precluding them from interfering with each other and providing the illusion that each process is executing in isolation. A process can specify regions of address space with different protection modes, and the virtual-memory mechanism ensures that the access protection is not violated. For instance, a user program which accidentally accesses a region of address space that should only be accessed in kernel mode triggers a fault.

Automatic memory management provided by a managed runtime environment, such as Java Virtual Machine [100] and Common Language Runtime [107], is yet another example of a memory abstraction. This memory abstraction is enabled by the use of a garbage collector [104], which manages the allocation and deallocation of memory for programs executing in such a managed runtime environment. The automatic memory management abstracts away the notion of explicit memory addresses, which simplifies the task of programming and provides a layer of memory safety. It simplifies the task of programming, because the programmer is freed from manually managing memory usage. The programmer no longer needs to worry about allocating the right amount of memory for a piece of data or remembering to free a piece of allocated memory when the memory is no longer being used. This memory abstraction also provides a layer of memory safety. In such a managed runtime, a program execution assigns names to objects, and a name provides a handle to its associated object. Since this model eliminates the possibility of a program execution performing arbitrary memory accesses, a program execution cannot access memory out of bounds without generating an exception or accidentally corrupt a piece of data. Dangling pointers, resulted from freeing some memory while the memory is still in use, can no longer exist, because memory deallocation is handled automatically by the runtime system.

With the proliferation of multicore architectures, the computing field must move from writing sequential software to parallel software in order to take advantage of the computation power provided by modern hardware. Writing parallel programs, however, gives rise to a new set of challenges in how programs interact with memory, such as how to properly synchronize concurrent accesses to shared memory. I believe that investigating memory abstractions is a fruitful path. The previous two examples of memory abstractions designed for sequential programming are widely adopted and have proven to be successful. They hide the complexity of dealing with raw memory as supported by the underlying hardware, thereby significantly simplifying the task of programming, and they provide an additional layer of safety. These are precisely the same goals that we would like to achieve today for parallel programming.

This dissertation explores three memory abstractions designed to mitigate the complexity of synchronization, namely memory-mapped reducers, reducer arrays, and ownership-aware transactions. Reducer hyperobjects [48] are shown to be a useful linguistic mechanism for avoiding determinacy race [42, 116] in a dynamically multithreaded computation. This dissertation proposes an alternative design and implementation of reducers (Chapter 4) and reducer arrays (Chapter 5) that perform much more efficiently than existing implementations. The ownership-aware transactions (OAT) enable the use of the open-nesting methodology [113], which is more efficient than closed nesting, while providing a sensible semantics that the programmer can use to reason about the program behaviors. The hope is that, by exploring different kinds of memory abstractions, we can obtain a deeper understanding of these new sets of challenges concerning how parallel programs interact with the memory, which then allows us to design sensible synchronization mechanisms that simplify parallel programming and achieve safe and efficient concurrent accesses to shared memory as well.

As we gather more experiences in designing memory abstractions, I believe that we should also move down the software stack and investigate what other memory abstractions the lower system layers may provide to enable support for memory abstractions in the higher layers. This dissertation proposes operating system support for thread-local memory mapping (TLMM), which in itself can be viewed as a memory abstraction provided by the operating system that allows a partially shared and partially private virtual address space. The support for TLMM provides a convincing case study, since it has been shown to be useful for implementing memory abstractions offered by a concurrency platform, such as TLMM-based cactus stacks (Chapter 3), memory-mapped reducers (Chapter 4), and reducer arrays (Chapter 5). Besides these memory abstractions, TLMM can benefit other memory abstractions proposed by other researchers [2, 11, 101, 123] as well.

The memory abstractions explored in this dissertation by no means provides a final answer to the challenges in parallel programming — not a complete one anyway. In fact, there is still much room for exploration, improvement, and addressing challenges. In the case of memory-mapped reducers and reducer arrays, the way that the reducer mechanism operates imposes a fundamental limitation on how many reducers (or how large size of a reducer array) a particular computation can employ before the reduce overhead becomes a scalability bottleneck. The reduce overhead is incurred by the need to reduce all the additional views created during parallel execution, which is difficult to avoid if one wishes to maintain the serial ordering in which the updates are performed on the reducer. In some cases, however, if the updates are commutative as well as associative, one may be able to design a more efficient mechanism for commutative reducers. In the case of ownership-aware transactions, the use of ownership types, albeit necessary to enforce the proper data sharing that the OAT system depends on, results a cumbersome linguistic interface. The expressiveness of OAT's linguistic interface is another area that is not fully investigated. Nevertheless, I hope the that the study on memory abstractions documented in this dissertation represents a small step towards understanding how memory abstractions may aid parallel programming in the future.

Appendix A

The OAT Model and Sequential Consistency

This appendix contains the details of the proof of Theorem 6.20: if the OAT model generates a trace $(\mathcal{C},\Phi)$ and a topological sort order $\mathcal{S}$, then $\mathcal{S}$ satisfies Definition 6.13, i.e., $\mathcal{S}$ is sequentially consistent with respect to Φ.

The first part of the appendix proves that the OAT model preserves several invariants on memory operations and content sets of transactions. The second part of the appendix uses these invariants to prove Theorem 6.20.

The OAT model invariants

In order to state the OAT model invariants, we shall first examine the notion of "dynamic content sets" for transactions, which is a generalization of the static content sets from Definition 6.10.

Definition A.1 *At any time t, for any transaction $T \in \mathtt{xactions}^{(t)}(\mathcal{C})$ and a memory operation $u \in \mathtt{memOps}^{(t)}(\mathcal{C})$, define the* ***dynamic content sets*** $\mathtt{cContent}^{(t)}(T)$, $\mathtt{oContent}^{(t)}(T)$, $\mathtt{aContent}^{(t)}(T)$, *and* $\mathtt{vContent}^{(t)}(T)$ *according the* $\mathtt{ContentType}(t,u,T)$ *procedure:*

```
ContentType($t,u,T$)        // For any $u \in \mathtt{memOps}^{(t)}(T)$
$X = \mathtt{xparent}(u)$
while ($X \neq T$)
    if $X \in \mathtt{activeXactions}^{(t)}(\mathcal{C})$,     return $u \in \mathtt{vContent}^{(t)}(T)$
    if $X \in \mathtt{aborted}^{(t)}(\mathcal{C})$,     return $u \in \mathtt{aContent}^{(t)}(T)$
    if ($X = \mathtt{committer}(u)$)   return $u \in \mathtt{oContent}^{(t)}(T)$
    $X = \mathtt{xparent}(X)$
return $u \in \mathtt{cContent}^{(t)}(T)$
```

The difference between the dynamic content sets defined in Definition A.1 and the static content sets (defined in Definition 6.10) is that for dynamic content sets, if a PENDING or PENDING_ABORT transaction is encountered when walking up the tree from a memory operation u to a transaction T, u is placed in the *active content* of T, i.e., $u \in \mathtt{vContent}^{(t)}(T)$. The static content sets, on the other hand, are defined on the computation tree after the program has finished executing, and no active transactions should be encountered. If a transaction T completes at time t_{endT}, it is not hard to see that the dynamic classification $\mathtt{ContentType}(t,u,T)$ gives the same answer as the static classification $\mathtt{ContentType}(u,T)$ for all times $t \geq t_{endT}$. Furthermore, once a memory operation u is classified into one of the following the content sets $\mathtt{cContent}^{(t)}(T)$, $\mathtt{oContent}^{(t)}(T)$, or $\mathtt{aContent}^{(t)}(T)$

with respect to a transaction T at time t, u stays in that content set with respect to T for all times $t^* \geq t$. Lemma A.2 states this observation formally.

Lemma A.2 *Any any time t, for any transaction $T \in \texttt{xactions}^{(t)}(C)$, and a memory operation $u \in \texttt{memOps}^{(t)}(C)$, the following invariants are satisfied:*

1. *If $u \in \texttt{cContent}^{(t)}(T)$, then $u \in \texttt{cContent}(T)$.*
2. *If $u \in \texttt{oContent}^{(t)}(T)$, then $u \in \texttt{oContent}(T)$.*
3. *If $u \in \texttt{aContent}^{(t)}(T)$, then $u \in \texttt{aContent}(T)$.*

PROOF. Let ${S_T}^{(t)}(u) = \texttt{xactions}^{(t)}(C) \cap \texttt{ances}(u) \cap \texttt{pDesc}(T)$. That is, define ${S_T}^{(t)}(u)$ to be the set of transactions along the path from u to T at time t, excluding T. We shall consider each of the three cases one by one.

1. $u \in \texttt{cContent}^{(t)}(T)$: Since $u \in \texttt{cContent}^{(t)}(T)$, the set ${S_T}^{(t)}(u)$ is precisely the set of transactions examined by the procedure $\texttt{ContentType}(t,u,T)$ before it returns. Moreover, we know that there is no active transactions at time t in ${S_T}^{(t)}(u)$, i.e., ${S_T}^{(t)}(u) \cap \texttt{activeXactions}^{(t)}(C) = \emptyset$, or u would be in $\texttt{vContent}^{(t)}(T)$ instead. Therefore, ${S_T}^{(t)}(u) = {S_T}^{(t^*)}(u)$ for all times $t^* \geq t$. Since the $\texttt{ContentType}(u,T)$ procedure examines the set ${S_T}^{(t^*)}(u)$, with t^* being the time execution ends, and the status of $\texttt{ABORTED}$ and $\texttt{COMMITTED}$ transactions does not change, it must be that $u \in \texttt{cContent}(T)$.
2. $u \in \texttt{oContent}^{(t)}(T)$: Since $u \in \texttt{oContent}^{(t)}(T)$, it must be that $\texttt{committer}(u) \in {S_T}^{(t)}(u)$. Let $X = \texttt{committer}(u)$ and define ${S_X}^{(t)}(u) = \texttt{xactions}^{(t)}(C) \cap \texttt{ances}(u) \cap \texttt{desc}(X)$ (which includes X), i.e., ${S_X}^{(t)}(u)$ is precisely the set $\texttt{ContentType}(t,u,T)$ examines before it returns (it returns as soon as it finds X). We know that there is no active transactions at time t in ${S_X}^{(t)}(u)$, or u would be in $\texttt{vContent}^{(t)}(X)$ instead. Thus, the same argument from Case 1 applies, and it must be that $u \in \texttt{oContent}(T)$.
3. $u \in \texttt{aContent}^{(t)}(T)$: This case is similar to Case 2 if we define X to be the "first" aborted transactions encountered when walking along the path from u to T. That is, define:

$$\begin{aligned} leaf(S) &= \{Z \in S : \texttt{pDesc}(Z) \cap S = \emptyset\} \\ {S_A}^{(t)} &= \left\{A \in {S_T}^{(t)}(u) \ : \ status[A] = \texttt{ABORTED}\right\} \end{aligned}$$

   Let $X = leaf(S_A)$, and the same argument from Case 2 follows, i.e., since there is no active transactions in ${S_X}^{(t)}(u)$, it must be that $u \in \texttt{aContent}(T)$.

□

Lemma A.3 characterizes when a transaction should have a location in its write set.

Lemma A.3 *At any time step t, consider any transaction $T \in \texttt{activeXactions}^{(t)}(C)$ and any memory location ℓ. Let ${S_\ell}^{(t)} = \left\{u \in \texttt{memOps}^{(t)}(C) \ : \ W(u,\ell)\right\}$. Exactly one of the following cases holds:*

1. *It is the case that $\ell \notin \texttt{W}^{(t)}(T)$, and $\texttt{cContent}^{(t)}(T) \cap {S_\ell}^{(t)} = \emptyset$.*
2. *There exists an $(u,\ell) \in \texttt{W}^{(t)}(T)$ which happens at time t_u, and two conditions are satisfied:*
   (a) $\left(\texttt{cContent}^{(t)}(T) \cup \texttt{oContent}^{(t)}(T)\right) \cap {S_\ell}^{(t)}$.
   (b) *For any operation $v \in \left({S_\ell}^{(t)} - \{u\}\right)$ which happens at time t_v, where $t_u < t_v \leq t$, $v \in$* $\texttt{aContent}^{(t)}(T) \cup \texttt{vContent}^{(t)}(T)$.

3. $T = \texttt{root}(C)$*,* $(\perp, \ell) \in \texttt{W}^{(t)}(T)$*, and two conditions are satisfied:*

   (a) $\texttt{cContent}^{(t)}(T) \cap S_\ell{}^{(t)} = \emptyset$*.*

   (b) For all $v \in S_\ell{}^{(t)}$*,* $v \in \texttt{aContent}^{(t)}(T) \cup \texttt{vContent}^{(t)}(T)$*.*

PROOF. This theorem can be proved by induction on time, showing that every instruction executed in the OAT model preserves the invariant.

In the base case, at time step $t = 0$, the OAT model starts with a computation tree C that has a single transaction $\texttt{root}(C)$ with $(\perp, \ell) \in \texttt{W}(\texttt{root}(C))$ for all $\ell \in L$. On this step, we only have a single transaction which falls into Case 3, and the invariant is reserved.

For the inductive step, consider each instruction that a program in the OAT model can issue, as described in Section 6.3: `fork`, `join`, `xbegin`, `xend`, `xabort`, `read`, and `write`. The instructions `fork` and `join` do not create or finish any transactions, nor do they change any transaction write sets. Thus, they do not affect the invariant in Lemma A.3. Similarly, a successful `read` does not affect the invariant because it only adds a new pair (u, ℓ) into a read set of a transaction, but does not change any write sets.

Consider a successful `write` on setup t that creates a memory operation u satisfying $\texttt{W}(u, \ell)$. Let $X = \texttt{xparent}(u)$. Then the `write` adds (u, ℓ) to $\texttt{W}(X)$. For all transactions $T \in \texttt{activeXactions}^{(t)}(C)$, let's examine how u affect the invariant for T.

1. Suppose that $T = X$. Since `write` adds (u, ℓ) to $\texttt{W}^{(t)}(X)$, we shall check that Case 2 holds for X on step t. To check the first condition, we know that $u \in \texttt{cContent}^{(t)}(X)$ because $X = \texttt{xparent}(u)$, and so the first condition holds. The second condition holds trivially, because u happens on the current time step t, and there are no other operations v such that $t_v > t_u$.
2. For any transaction $T \neq X$ with $\ell \notin \texttt{W}^{(t)}(T)$, we know by the inductive hypothesis and Case 1 that $\texttt{cContent}^{(t-1)}(T) \cap S_\ell{}^{(t-1)} = \emptyset$. After the step, we still have $\ell \notin \texttt{W}^{(t)}(T)$ and $\texttt{cContent}^{(t)}(T) \cap S_\ell{}^{(t)} = \emptyset$, since u only changes the closed content set of $\texttt{cContent}^{(t)}(X)$.
3. For any transaction $T \neq X$ with $(w, \ell) \in \texttt{W}^{(t)}(T)$, we know that $T \in \texttt{xAnces}(u)$, which also implies that $T \in \texttt{xAnces}(X)$. Otherwise, u would have caused a memory conflict with T according to Definition 6.6.
   There are two subcases to consider: either $w \neq \perp$ or $w = \perp$.
   - If $w \neq \perp$, by inductive hypothesis and Case 2a, $w \in (\texttt{cContent}(T) \cup \texttt{oContent}(T))$ before and after step t. Also, since X is issuing the `write` instruction, we know that $X \in \texttt{activeXactions}^{(t)}(C)$, and thus u is added to $\texttt{vContent}^{(t)}(T)$, and Case 2b still holds.
   - If $w = \perp$, which implies that $T = \texttt{root}(C)$, we have a similar subcase, except that T falls into Case 3 of Lemma A.3 instead of Case 2. Case 3a is preserved because $T \neq X$ and the `write` instruction does not change $\texttt{cContent}^{(t)}(T)$. Case 3b is preserved as well because u is added to $\texttt{vContent}^{(t)}(T)$.

Thus, a successful `write` instruction preserves the invariant of Lemma A.3.

Consider an xbegin that creates a transaction Z. Since Z begins with $\texttt{R}(Z) = \texttt{W}(Z) = \emptyset$, Z falls into Case 1, which is trivially satisfied because $\texttt{cContent}^{(t)}(Z) = \emptyset$.

Next, consider an xend that successfully commits a transaction Z. Let $Y = \texttt{xparent}(Z)$. Then, since the xend changes Z's status from PENDING to COMMITTED, we know that

$$\texttt{cContent}^{(t)}(Y) = \texttt{cContent}^{(t-1)}(Y) \cup \texttt{cContent}^{(t-1)}(Z) - \left\{ w \in \texttt{cContent}^{(t-1)}(Z) \,:\, Z = \texttt{committer}(w) \right\} .$$

That is, the commit of Z merges its closed content into the closed content of its parent, except for the memory operations that operate on memory locations owned by xMod(Z) (since those are committed in an open-nested fashion to root(C)).

The write sets and content sets for all other transactions besides Y, Z, and root(C) are unchanged by the xend, and we no longer need to consider Z's write set and content sets since it is no longer active (i.e., $Z \notin$ activeXactions$^{(t)}(C)$). Thus, we only need to check weather the xend still preserves the invariant of Lemma A.3 for Y and root(C). For any memory location ℓ, consider the possible cases for how the commit of Z can change W(Y) and W(root(C)).

1. Suppose that $\ell \notin$ W$^{(t-1)}(Z)$. By inductive hypothesis and Case 1, we know that cContent$^{(t-1)}(Z) \cap S_\ell{}^{(t-1)} = \emptyset$. We also know that the set cContent$(Y) \cap S_\ell$ is the same before and after step t. The same argument applies to the root(C). Thus, for Y and root(C), xend preserves Case 1, Case 2a, or Case 3a in this scenario.

   Now we check for Case 2b or Case 3b. The only way that the xend instruction can contradict Case 2b or Case 3b is to remove a memory operation v from aContent(Y) or vContent(Y). This cannot be the case, however. For aContent(Y), we know by Lemma A.2 that, for any memory operation $v \in$ aContent$^{(t-1)}(Y)$, it must be that aContent$^{(t)}(Y)$. For vContent(Y), on the other hand, any memory operation v removed from vContent$^{(t-1)}(Y)$ must be added to cContent$^{(t)}(Y)$, but this cannot be the case because cContent(Y) remains the same. Again, the same argument applies to the root(C). Thus, the xend instruction also preserves Case 2b or Case 3b in this scenario.

2. Suppose that $(u,\ell) \in$ W$^{(t-1)}(Z)$. To check whether the invariant still holds for Y and for root(C), we have two subcases to consider: $Z =$ committed(u) or $Z \neq$ committed(u).

   - Suppose $Z =$ committer(u). It must be the case that $(u,\ell) \notin$ W$^{(t)}(Y)$ before and after the step, since by Theorem 6.8, Z is the unique committer of ℓ, and Y, being a proper ancestor of Z, can never directly access l. This scenario falls under Case 1, and the invariant is preserved for Y.

     For root(C), on the other hand, (u,ℓ) is propagated to W$^{(t)}$(root(C)), so we need to check that Case 2 still holds. We know that Case 2a holds, since $Z =$ committer(u), and so when Z commits on step t, $u \in$ oContent$^{(t)}$(root(C)).

     Now we check that Case 2b holds for root(C). By the inductive hypothesis (Case 2), we know that for all $v \in S_\ell{}^{(t-1)}$ such that $t_v > t_u$, we have $v \in$ aContent$^{(t-1)}(Z) \cup$ oContent$^{(t-1)}(Z)$. When Z commits on step t, however, it must be that vContent$^{(t)}(Z) = \emptyset$, since Z can only commit if all its nested transactions have completed. Thus, any such v must be in aContent$^{(t-1)}(Z)$. Since aContent$^{(t-1)}(Z) \subseteq$ aContent$^{(t-1)}$(root(C)) $=$ aContent$^{(t)}$(root(C)), v satisfies Case 2b for root(C).

   - Suppose $Z \neq$ committer(u). In this case, we just need to check that the invariant still holds for Y, since the write set and content sets for root(C) with respect to ℓ remains the same before and after the step. Since $Z \neq$ committer(u), we know that after step t, $(u,\ell) \in$ W$^{(t)}(Y)$, so we need to check Case 2 for Y.

     First, we can verify that Case 2a holds for Y. By inductive hypothesis, $u \in$ cContent$^{(t-1)}(Z)$. Thus, after xend, we have $u \in$ cContent$^{(t)}(Y)$.

     Next, we can very that Case 2b holds for Y. This subcase is similar to the subcase of root(C) when $Z =$ committer(u), and the same argument applies.

Thus, xend preserves the invariant in Lemma A.3.

Finally, the xabort instruction (which could be triggered by sigabort) preserves the invariant in Lemma A.3. The xabort of a transaction Z causes Z to be removed from $\texttt{activeXactions}^{(t)}(C)$, which eliminates the need to check the invariants for Z. In addition, the only content sets affected by the abort of transaction Z are the content sets of transactions $X \in \texttt{pAnces}(Z) \cap \texttt{activeXactions}(C)$, where xabort of Z only moves an operation v from $\texttt{vContent}^{(t-1)}(X)$ to $\texttt{aContent}^{(t)}(X)$, so the invariant is preserved for any active transactions that are Z's proper ancestors. □

The intuition for Lemma A.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. Finally, for the most part, when a write operation u is committed with respect to T, it is the case that $u \in \texttt{cContent}(T)$ (in Case 2a), unless $T = \texttt{root}(C)$, since if $T \neq \texttt{root}(C)$ and has $(u, \ell) \in \texttt{W}(T)$, it must be that $\texttt{xid}(\texttt{owner}(\ell)) \leq \texttt{xid}(\texttt{xMod}(T))$. Otherwise T would not be able to access ℓ directly by Theorem 6.8. The only case where $(u, \ell) \in \texttt{W}(T)$ and $w \in \texttt{oContent}(T)$ is when $T = \texttt{root}(C)$, since a transaction $Z = \texttt{committer}(u)$ commits (u, ℓ) to $\texttt{W}(\texttt{root}(C))$ as described in Section 6.3.

Case 1 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 3 of Lemma A.3 handles the special case of the root.

Proof of sequential consistency

Finally, Theorem 6.20 uses invariants from Lemma A.2 and Lemma A.3 to prove that, if the OAT model generates a trace (C, Φ) and a topological sort order $\mathcal{S}$, then $\mathcal{S}$ satisfies Definition 6.13, i.e., $\Phi = \mathcal{X}_{\mathcal{S}}$, or $\mathcal{S}$ is sequentially consistent with respect to Φ.

PROOF. [Theorem 6.20]
To show that $\Phi = \mathcal{X}_{\mathcal{S}}$, one must show that for all $v \in \texttt{memOps}(C)$, let $\Phi(v) = u$, and u satisfies the four conditions of the transactional last writer of v according to $\mathcal{S}$, as described in Definition 6.12:

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 condition and second conditions are true by construction, since the OAT model can only set $\Phi(v) = u$ if $u <_{\mathcal{S}} v$, $W(u, \ell)$ and $R(v, \ell) \vee W(v, \ell)$.

Now we check the third condition. Suppose at time t_v, memory operation v happens and the OAT model sets $\Phi(v) = u$. We know that $u \in S_{\ell}{}^{(t_v)}$ as defined in Lemma A.3, since $u <_{\mathcal{S}} v$ and $u = \Phi(v)$ (i.e., u is a write). Also, it must be that $(u, \ell) \in \texttt{W}^{(t_v)}(X)$ for some transaction $X \in \texttt{xAnces}(v)$, or v would have caused a conflict with X (by Definition 6.6). Let $L = \texttt{xLCA}(u, v)$, and we know that $X \in \texttt{xAnces}(L)$, since $u, v \in \texttt{memOps}(X)$ and $L = \texttt{xLCA}(u, v)$. By Lemma A.3 Case 2a, we have $u \in \texttt{cContent}^{(t_v)}(X) \cup \texttt{oContent}^{(t_v)}(X)$. Since $X \in \texttt{xAnces}(L)$, it must be that $u \in \texttt{cContent}^{(t_v)}(L) \cup \texttt{oContent}^{(t_v)}(L)$ as well. Thus, by Lemma A.2, it must be that $u \in \texttt{cContent}(L) \cup \texttt{oContent}(L)$ at the end of the computation, and $\neg(uHv)$, satisfying the third condition.

To check the fourth condition, assume for contradiction that there exists a w such that $W(w, \ell)$, and $u <_{\mathcal{S}} w <_{\mathcal{S}} v$. Since $u \in \texttt{W}^{(t_v)}(X)$, by Lemma A.3 Case 2b, we know $w \in \texttt{aContent}^{(t_v)}(X) \cup \texttt{vContent}^{(t_v)}(X)$ (which also implies $w \in \texttt{memOps}^{(t_v)}(X)$).

Let $Y = \mathtt{xLCA}(w,v)$. Since $w \in \mathtt{memOps}^{(t_v)}(X)$, we know $X \in \mathtt{ances}(Y)$. There are two cases to consider for w:

1. Suppose $w \in \mathtt{aContent}^{(t_v)}(X)$. Since $X \in \mathtt{ances}(Y)$, $w \in \mathtt{cContent}^{(t_v)}(Y) \cap \mathtt{aContent}^{(t_v)}(Y)$. We can show by contradiction that $w \in \mathtt{aContent}^{(t_v)}(Y)$, and so we have wHv.

   (a) Suppose $Y = T$. Then we already have $w \in \mathtt{aContent}^{(t_v)}(Y)$ by the original assumption.

   (b) Suppose $T \in \mathtt{pAnces}(Y)$. If we had $w \in \mathtt{cContent}^{(t_v)}(Y)$, then by Lemma A.3, we must have some `write` y such that $(y,\ell) \in \mathtt{W}^{(t_v)}(Y)$. This statement contradicts the fact that OAT model found (u,ℓ) from transaction X, since a closer transaction Y had ℓ in its read set. Thus, it must be that $w \in \mathtt{aContent}^{(t_v)}(Y)$.

2. Suppose $w \in \mathtt{vContent}^{(t_v)}(T)$. Then, we know $w \in \mathtt{cContent}^{(t_v)}(Y) \cup \mathtt{vContent}^{(t_v)}(Y)$. As in the previous case, we can show $w \notin \mathtt{cContent}^{(t_v)}(Y)$ and we have wHv.

   If $w \in \mathtt{vContent}^{(t_v)}(Y)$, then there exists some transaction $Z \in \mathtt{activeXactions}^{(t_v)}(Y) - \{Y\}$ such that $\ell \in \mathtt{W}^{(t_v)}(Z)$ (by Definition A.1). This statement leads to a contradiction, however. We know that $Z \notin \mathtt{xAnces}(v)$ since $Y = \mathtt{xLCA}(w,v)$ and Z is a proper descendant of Y. Thus, if it were the case that $w \in \mathtt{W}^{(t_v)}(Z)$, since $Z \notin \mathtt{xAnces}(v)$, v would have caused a conflict, contradicting the assumption that v is a successful operation.

In both cases, the fourth condition is satisfied. Therefore, we have $\Phi = X_S$. □

Appendix B

Rules for the OAT Type System

This appendix contains the type rules for the OAT type system. The grammar for the type system is presented below:

$$
\begin{array}{rcl}
P & = & \mathit{defn}^{*};\ e \\
\mathit{defn} & = & \textsf{class}\ \mathit{cDecl}\ \textsf{extends}\ \mathit{cDecl}\ \textsf{where}\ \mathit{constr}^{*}\ \{\ \mathit{field}^{*};\ \mathit{init};\ \mathit{meth}^{*}\ \} \\
\mathit{cDecl} & = & \mathit{cn}\langle \mathit{formal}^{+}\rangle\ \mid\ \textsf{Object}\langle \mathit{formal}\rangle\ \mid\ \textsf{Xmodule}\langle \mathit{formal}\rangle \\
\mathit{constr} & = & \mathit{formal} < \mathit{formal}\ \mid\ \mathit{formal} = \mathit{formal}\ \mid\ \mathit{formal} \neq \mathit{formal} \\
\mathit{field} & = & t\ \mathit{fd} \\
\mathit{init} & = & \mathit{cn}\langle \mathit{formal}^{+}\rangle(\mathit{param}^{*})\ \{\ \textsf{super}\langle \mathit{formal}^{+}\rangle(e^{*});\ \textsf{this}.\mathit{fd} = e;^{*}\ \} \\
\mathit{meth} & = & t\ \mathit{mn}\langle \mathit{formal}^{*}\rangle(\mathit{param}^{*})\ \textsf{where}\ \mathit{constr}^{*}\{\ e\ \} \\
\mathit{param} & = & t\ x \\
\mathit{owner} & = & \textsf{world}[i]\ \mid \mathit{formal}\ \mid\ \textsf{this}[i] \\
\mathit{formal} & = & f \\
t & = & \textsf{int}\ \mid\ \textsf{constraint}\ \mid\ \mathit{ct} \\
\mathit{ct} & = & \mathit{cn}\langle \mathit{owner}^{+}\rangle \\
e & = & \textsf{new}\ \mathit{ct}(e^{*})\ \mid\ x\ \mid\ x = e\ \mid\ \textsf{let}\ (\mathit{param} = e)\ \textsf{in}\ \{\ e\ \} \\
 & & \mid\ x.\mathit{fd}\ \mid\ x.\mathit{fd} = e\ \mid\ x.\mathit{mn}\langle \mathit{owner}^{+}\rangle(e^{*}) \\
 & & \\
\mathit{cn} & = & \textit{a class name that is not}\ \textsf{Object}\ \textit{nor}\ \textsf{Xmodule} \\
\mathit{mn} & = & \textit{a method name that is not a constructor} \\
\mathit{fd} & = & \textit{a field name} \\
x, y & = & \textit{a variable name} \\
f, g & = & \textit{an owner formal} \\
i, j & = & \textit{an}\ \textsf{int}\ \textit{literal}
\end{array}
$$

For simplicity, the OAT type system makes the following assumptions. First, each class has only one constructor (specified by the term *init*), and that all fields are initialized properly after the call to the constructor. Second, all field names (whether inherited or declared) are distinct. Third, the call to super is explicit. Fourth, an index is always specified when the ownership tags world and this are used. Fifth, the class names Object and Xmodule are special and assumed to be properly defined by the system. Finally, the explicit use of upcast and downcast are not allowed, as specified in the abstract syntax.

For the constraints on owners (*constr*), the notation $<$ is used as defined in Section 6.2: Assuming f_1 and f_2 are instantiated with o_1 and o_2, $f_1 < f_2$ specifies that either $o_1.name \prec o_2.name$, or $o_1.name = o_2.name$ and $o_1.index < o_2.index$. Similarly, $f_1 = f_2$ specifies that $o_1.name = o_2.name$ and $o_1.index = o_2.index$. On the other hand, $f_1 \neq f_2$ specifies that either $o_1.name \neq o_2.name$, or $o_1.name = o_2.name$ and $o_1.index \neq o_2.index$.

The OAT type system uses some shorthand notation. Henceforth, for brevity, the notation $\lhd$ is used in place of the keyword `extends` (i.e., A extends B is written as $A \lhd B$). The notation $\unlhd$ between class names is the reflexive and transitive closure induced by the $\lhd$ relation. On the other hand, the notation $\ntrianglelefteq$ simply indicates that the $\unlhd$ relation does not hold. Note that the $\lhd$ is not the same as subtyping (denoted as $<:$), because $\lhd$ only considers the static relation defined by the `extends` keyword, and does not account for the ownership tags. Furthermore, $\mathit{field} \in_d cn\langle\ldots\rangle$ is used to mean that class $cn\langle\ldots\rangle$ declares *field* and $\mathit{field} \in_i cn\langle\ldots\rangle$ is used to mean that class $cn\langle\ldots\rangle$ inherits *field*. Finally, $\mathit{field} \in cn\langle\ldots\rangle$ is used to mean that either $\mathit{field} \in_d cn\langle\ldots\rangle$ or $\mathit{field} \in_i cn\langle\ldots\rangle$. These notations are used for *fd* (field name), *meth* (method), and *mn* (method name) similarly.
The following predicates are used in the typing rules:

Predicate	Meaning
ClassOnce(P)	No class is declared twice in P $\forall cn, cn'$ *in* P, $cn \neq cn'$
FieldsOnce(P)	No class contains two fields with the same name $\forall ct\ \forall fd, fd' \in ct$ *in* $P, fd \neq fd'$
MethodsOnce(P)	No class declares two methods with the same name $\forall ct\ \forall mn, mn' \in_d ct$ *in* P, $mn \neq mn'$
WFClasses(P)	No cycles in the class hierarchy; i.e., the $\unlhd$ relation is antisymmetric $\forall cn, cn'$ *in* P, $cn \unlhd cn' \wedge cn' \unlhd cn \implies cn = cn'$

The typing judgment has the form: $P; \Gamma \vdash e : t$, where P is the program being checked to provide information about class definitions; Γ is the typing environment, providing mappings from a variable name to its static type for the free variables in e; finally, t is the static type of e.

The typing environment Γ is defined as $\Gamma ::= \emptyset \mid \Gamma, x : t \mid \Gamma, f : \mathit{owner} \mid \Gamma, \mathit{constr} : \text{constraint}$. That is, the typing environment Γ contains the types of variables, the owner parameters and the constraints among owners. Note that an entry *constr* always has type constraint , which is a type used implicitly by the type system and cannot be used by the user program. For simplicity, the type rules drop the constraint type when listing the *constr* entries in Γ when it is clear from the context. When checking for well-formness of the typing environment, we assume the new entries are checked in the order listed, from left to right. The domain of the typing environment, $Dom(\Gamma)$, intuitively, is defined to be the set of variables, owner parameters, and constraints bound by Γ.

The typing system uses the following judgments:

Judgment	Meaning
$\vdash P : t$	program P yields type t
$P \vdash \mathit{defn}$	*defn* is a well-formed class
$P \vdash cn\langle f_{1..n}\rangle \lhd cn'\langle g_{1..k}\rangle$	class $cn\langle f_{1..n}\rangle$ extends class $cn'\langle g_{1..k}\rangle$
$P \vdash cn \unlhd cn'$	cn' is an ancestor of cn in the graph defined by the extends keyword
$P \vdash \mathit{field} \in_d cn\langle\ldots\rangle$	class $cn\langle\ldots\rangle$ declares *field*
$P \vdash \mathit{field} \in_i cn\langle\ldots\rangle$	class $cn\langle\ldots\rangle$ inherits *field*
$P \vdash \mathit{field} \in cn\langle\ldots\rangle$	class $cn\langle\ldots\rangle$ declares / inherits *field*
$P \vdash \mathit{init} \in cn\langle\ldots\rangle$	class $cn\langle\ldots\rangle$ declares *init*
$P \vdash \mathit{meth} \in_d cn\langle\ldots\rangle$	class $cn\langle\ldots\rangle$ declares *meth*
$P \vdash \mathit{meth} \in_i cn\langle\ldots\rangle$	class $cn\langle\ldots\rangle$ inherits *meth*
$P \vdash \mathit{meth} \in cn\langle\ldots\rangle$	class $cn\langle\ldots\rangle$ declares / inherits *meth*
$P; \Gamma \vdash \mathit{field}$	*field* is a well-formed field
$P; \Gamma \vdash \mathit{meth}$	*meth* is a well-formed method
$P; \Gamma \vdash \mathit{wf}$	typing environment Γ is well-formed
$P; \Gamma \vdash t$	t is a well-formed type
$P; \Gamma \vdash \mathit{constr}$	constraint *constr* is satisfied
$P; \Gamma \vdash_{owner} o$	o is an owner
$P; \Gamma \vdash e : t$	expression e has type t
$P; \Gamma \vdash t <: t'$	t is a subtype of t'

In the type rules, we also use the following auxiliary rules:

The Extends Relation

$$\frac{P \vdash \textsf{class}\ cn\langle f_{1..n}\rangle\ \textsf{extends}\ cn'\langle g_{1..m}\rangle\ \ldots}{P \vdash cn\langle f_{1..n}\rangle \lhd cn'\langle g_{1..m}\rangle}$$

$$\frac{}{P \vdash cn \unlhd cn} \qquad \frac{P \vdash cn\langle f_{1..n}\rangle \lhd cn'\langle g_{1..m}\rangle}{P \vdash cn \unlhd cn'} \qquad \frac{P \vdash cn \unlhd cn' \qquad P \vdash cn' \unlhd cn''}{P \vdash cn \unlhd cn''}$$

Type Lookup

$$\begin{aligned}
&\mathit{type}(\,) \;=\; ()\\
&\mathit{type}(t\,x) \;=\; t\\
&\mathit{type}(t\,\mathit{fd}) \;=\; t\\
&\mathit{type}(t_1\,x_1,\,t_2,\,x_2,\,\ldots) \;=\; t_1,\,t_2,\,\ldots
\end{aligned}$$

Field Lookup

$$\frac{P \vdash \textsf{class}\ cn\langle f_{1..n}\rangle\ \ldots\ \{\ \ldots\ \mathit{field}\ \ldots\ \}}{P \vdash \mathit{field}\ \in_d\ cn\langle f_{1..n}\rangle}$$

$$\frac{\begin{array}{c} P \vdash \mathit{field}\ \in\ cn'\langle g_{1..m}\rangle \\ P \vdash cn\langle f_{1..n}\rangle \lhd cn'\langle o_{1..m}\rangle \end{array}}{P \vdash \mathit{field}\,[o_1/g_1]..[o_m/g_m]\ \in_i\ cn\langle f_{1..n}\rangle}$$

$$\frac{P \vdash \mathit{field}\ \in_d\ cn\langle f_{1..n}\rangle \vee P \vdash \mathit{field}\ \in_i\ cn\langle f_{1..n}\rangle}{P \vdash \mathit{field}\ \in\ cn\langle f_{1..n}\rangle}$$

Init Lookup

$$\frac{P \vdash \textsf{class}\ cn\langle f_{1..n}\rangle\ \ldots\ \{\ \ldots\ \mathit{init}\ \ldots\ \}}{P \vdash \mathit{init}\ \in\ cn\langle f_{1..n}\rangle}$$

Method Lookup

$$\frac{P \vdash \textsf{class}\ cn\langle f_{1..n}\rangle\ \ldots\ \{\ \ldots\ \mathit{meth}\ \ldots\ \}}{P \vdash \mathit{meth}\ \in_d\ cn\langle f_{1..n}\rangle}$$

$$\frac{\begin{array}{c} P \vdash \mathit{meth}\ \in\ cn'\langle g_{1..m}\rangle \\ P \vdash cn\langle f_{1..n}\rangle \lhd cn'\langle o_{1..m}\rangle \end{array}}{P \vdash \mathit{meth}\,[o_1/g_1]..[o_m/g_m]\ \in_i\ cn\langle f_{1..n}\rangle}$$

$$\frac{P \vdash \mathit{meth}\ \in_d\ cn\langle f_{1..n}\rangle \vee P \vdash \mathit{meth}\ \in_i\ cn\langle f_{1..n}\rangle}{P \vdash \mathit{meth}\ \in\ cn\langle f_{1..n}\rangle}$$

Override Ok

$$\frac{\begin{array}{c} P \vdash cn\langle f_{1..n}\rangle \lhd cn'\langle o_{1..m}\rangle \\ P \vdash t\ mn\langle\ldots\rangle(t_i\ \ x_i{}^{i\in 1..k})\ \ \ldots\ \in_d\ cn\langle f_{1..n}\rangle \\ P \vdash t[g_1/o_1]..[g_m/o_m]\ \ mn\langle\ldots\rangle(t_i[g_1/o_1]..[g_m/o_m]\ \ y_i{}^{i\in 1..k})\ \ \ldots\ \notin\ cn'\langle g_{1..m}\rangle \end{array}}{\mathit{OverrideOk}(cn\langle f_{1..n}\rangle,\ cn'\langle o_{1..m}\rangle,\ \mathit{meth})}$$

$$\frac{\begin{array}{c} P \vdash cn\langle f_{1..n}\rangle \lhd cn'\langle o_{1..m}\rangle \\ P \vdash t\ mn\langle f_{n+1..n+j}\rangle(t_i\ \ x_i{}^{i\in 1..k})\ \ \ldots\ \in_d\ cn\langle f_{1..n}\rangle \\ P \vdash t'\ mn\langle g_{n+1..n+j}\rangle(t_i'\ \ y_i{}^{i\in 1..k})\ \ \ldots\ \in\ cn'\langle g_{1..m}\rangle \\ t\ =\ t'\ [o_1/g_1]..[o_m/g_m] \qquad \mathit{type}(t_i\ \ x_i{}^{i\in 1..k}) = \mathit{type}(t_i'\ \ y_i{}^{i\in 1..k})\ [o_1/g_1]..[o_m/g_m] \end{array}}{\mathit{OverrideOk}(cn\langle f_{1..n}\rangle,\ cn'\langle o_{1..m}\rangle,\ \mathit{meth})}$$

The type rules are presented below:

$\boxed{\vdash P : t}$

[PROG]

$$\frac{\begin{array}{c} \mathit{WFClasses}(P) \qquad \mathit{ClassOnce}(P) \qquad \mathit{FieldsOnce}(P) \qquad \mathit{MethodsOnce}(P) \\ P = \mathit{defn}_{1..n};\ e \qquad P \vdash \mathit{defn}_i \qquad P;\, \emptyset \vdash e : t \end{array}}{\vdash P : t}$$

$\boxed{P \vdash \mathit{defn}}$

[CLASS]

$$\frac{\begin{array}{c} P \vdash cn \ntrianglelefteq \textsf{Xmodule} \\ \Gamma = f_{1..n} : \mathit{owner},\ f_1 < f_i : \textsf{constraint},\ \mathit{constr}^*,\ \textsf{this} : cn\langle f_{1..n}\rangle \\ P;\, \Gamma \vdash \mathit{wf} \qquad P;\, \Gamma \vdash cn'\langle f_1,\, o^*\rangle \qquad P;\, \Gamma \vdash \mathit{field}_i \qquad P;\, \Gamma \vdash \mathit{init} \qquad P;\, \Gamma \vdash \mathit{meth}_i \\ \mathit{OverrideOk}(\ cn\langle f_{1..n}\rangle,\ cn'\langle f_1,\, o^*\rangle,\ \mathit{meth}_i\ ) \end{array}}{P \vdash \textsf{class}\ \ cn\langle f_{1..n}\rangle\ \ \textsf{extends}\ \ cn'\langle f_1,\, o^*\rangle\ \ \textsf{where}\ \ \mathit{constr}^*\ \{\ \mathit{field}^*;\ \ \mathit{init};\ \ \mathit{meth}^*\ \}}$$

[XMODULE CLASS]

$$\frac{\begin{array}{c} P \vdash cn \trianglelefteq \textsf{Xmodule} \\ \Gamma = f_{1..n} : \mathit{owner},\ f_1 < f_i : \textsf{constraint},\ \mathit{constr}^*,\ \textsf{this} : cn\langle f_{1..n}\rangle,\ \textsf{this} : \mathit{owner},\ \textsf{this}[i] < f_1 \\ P;\, \Gamma \vdash \mathit{wf} \qquad P;\, \Gamma \vdash cn'\langle f_1,\, o^*\rangle \qquad P;\, \Gamma \vdash \mathit{field}_i \qquad P;\, \Gamma \vdash \mathit{init} \qquad P;\, \Gamma \vdash \mathit{meth}_i \\ \mathit{type}(\ \mathit{field}_i\ ) \neq \textsf{int} \qquad \mathit{OverrideOk}(\ cn\langle f_{1..n}\rangle,\ cn'\langle f_1,\, o^*\rangle,\ \mathit{meth}_i\ ) \end{array}}{P \vdash \textsf{class}\ \ cn\langle f_{1..n}\rangle\ \ \textsf{extends}\ \ cn'\langle f_1,\, o^*\rangle\ \ \textsf{where}\ \ \mathit{constr}^*\ \{\ \mathit{field}^*;\ \ \mathit{init};\ \ \mathit{meth}^*\ \}}$$

$\boxed{P;\, \Gamma \vdash \mathit{init}}$

[INIT]

$$\frac{\begin{array}{c} P \vdash cn\langle f_{1..n}\rangle \lhd cn'\langle f_1, o_{2..m}\rangle \\ \Gamma' = \Gamma,\ \mathit{param}^* \qquad P;\, \Gamma' \vdash \mathit{wf} \qquad P;\, \Gamma' \vdash \textsf{this}.\mathit{fd}_j = e_j \\ P \vdash cn'\langle g_{1..m}\rangle(t_i\ \ x_i{}^{i \in 1..k})\ \{\ \ldots\ \} \in cn'\langle g_{1..m}\rangle \qquad P;\, \Gamma' \vdash e_i : t_i\ [f_1/g_1][o_2/g_2]..[o_m/g_m] \end{array}}{P;\, \Gamma \vdash cn\langle f_{1..n}\rangle(\mathit{param}^*)\ \{\ \textsf{super}\langle f_1,\ o_{2..m}\rangle(e_i{}^{i \in 1..k});\quad \textsf{this}.\mathit{fd} = e;^*\ \}}$$

$\boxed{P;\, \Gamma \vdash \mathit{field}}$ $\boxed{P;\, \Gamma \vdash \mathit{meth}}$

[FIELD] [METHOD]

$$\frac{P;\, \Gamma \vdash t}{P;\, \Gamma \vdash t\ \mathit{fd}} \qquad \frac{\Gamma' = \Gamma,\ f_{1..n} : \mathit{owner},\ \mathit{constr}^*,\ \mathit{param}^* \qquad P;\, \Gamma' \vdash \mathit{wf} \qquad P;\, \Gamma' \vdash e : t}{P;\, \Gamma \vdash t\ mn\langle f_{1..n}\rangle(\mathit{param}^*)\ \textsf{where}\ \ \mathit{constr}^*\ \{e\}}$$

$\boxed{P; \Gamma \vdash wf}$

[ENV $\emptyset$]

$$\frac{}{P; \emptyset \vdash wf}$$

[ENV X]

$$\frac{P; \Gamma \vdash t \qquad x \notin \mathrm{Dom}(\Gamma) \qquad P; \Gamma \vdash wf}{P; \Gamma, x : t \vdash wf}$$

[ENV OWNER]

$$\frac{f \notin \mathrm{Dom}(\Gamma) \qquad P; \Gamma \vdash wf}{P; \Gamma, f : owner \vdash wf}$$

[ENV CONSTR <]

$$\frac{\begin{array}{c} P; \Gamma \vdash wf \qquad P; \Gamma \vdash_{owner} o, o' \qquad \Gamma' = \Gamma, o < o' : \text{constraint} \\ \nexists_{f,g}\, (P; \Gamma' \vdash f < g) \wedge (P; \Gamma' \vdash g < f) \qquad \nexists_{f,g}\, (P; \Gamma' \vdash f < g) \wedge (P; \Gamma' \vdash f = g) \end{array}}{P; \Gamma, o < o' \vdash wf}$$

[ENV CONSTR =]

$$\frac{\begin{array}{c} P; \Gamma \vdash wf \qquad P; \Gamma \vdash_{owner} o, o' \qquad \Gamma' = \Gamma, constr : \text{constraint} \\ \nexists_{f,g}\, (P; \Gamma' \vdash f < g) \wedge (P; \Gamma' \vdash f = g) \\ \nexists_{f,g}\, (P; \Gamma' \vdash g < f) \wedge (P; \Gamma' \vdash f = g) \qquad \nexists_{f,g}\, (P; \Gamma' \vdash f = g) \wedge (P; \Gamma' \vdash f \neq g) \end{array}}{P; \Gamma, constr \vdash wf}$$

[ENV CONSTR $\neq$]

$$\frac{\begin{array}{c} P; \Gamma \vdash wf \qquad P; \Gamma \vdash_{owner} o, o' \qquad \Gamma' = \Gamma, o \neq o' : \text{constraint} \\ \nexists_{f,g}\, (P; \Gamma' \vdash f < g) \wedge (P; \Gamma' \vdash f \neq g) \qquad \nexists_{f,g}\, (P; \Gamma' \vdash f = g) \wedge (P; \Gamma' \vdash f \neq g) \end{array}}{P; \Gamma, constr \vdash wf}$$

$\boxed{P; \Gamma \vdash t}$

[TYPE INT]

$$\frac{}{P; \Gamma \vdash \text{int}}$$

[TYPE CONSTRAINT]

$$\frac{}{P; \Gamma \vdash \text{constraint}}$$

[TYPE OBJECT]

$$\frac{P; \Gamma \vdash_{owner} o}{P; \Gamma \vdash \text{Object}\langle o \rangle}$$

[TYPE XMODULE]

$$\frac{P; \Gamma \vdash_{owner} o}{P; \Gamma \vdash \text{Xmodule}\langle o \rangle}$$

[TYPE CT]

$$\frac{\begin{array}{cc} & P \vdash \text{class } cn\langle f_{1..n} \rangle \ \ldots \text{ where } constr^{*} \ \ldots \\ P; \Gamma \vdash_{owner} o_i & P; \Gamma \vdash o_1 < o_i : \text{constraint} \qquad P; \Gamma \vdash constr\,[o_1/f_1]..[o_n/f_n] \end{array}}{P; \Gamma \vdash cn\langle o_{1..n} \rangle}$$

$\boxed{P;\ \Gamma \vdash constr}$

[CONSTR ENV]

$$\frac{\Gamma = \Gamma',\ constr,\ \Gamma''}{P;\ \Gamma \vdash constr}$$

[< WORLD I]

$$\frac{\begin{array}{c} P;\ \Gamma \vdash_{owner} o \\ P;\ \Gamma \vdash o \neq \mathsf{world} \end{array}}{P;\ \Gamma \vdash o < \mathsf{world}[i]}$$

[< WORLD II]

$$\frac{i < j}{P;\ \Gamma \vdash \mathsf{world}[i] < \mathsf{world}[j]}$$

[< THIS]

$$\frac{\begin{array}{c} i < j \\ P;\ \Gamma \vdash_{owner} \mathsf{this} \end{array}}{P;\ \Gamma \vdash \mathsf{this}[i] < \mathsf{this}[j]}$$

[< TRANS]

$$\frac{\begin{array}{c} P;\ \Gamma \vdash o_1 < o_2 \\ P;\ \Gamma \vdash o_2 < o_3 \end{array}}{P;\ \Gamma \vdash o_1 < o_3}$$

[= WORLD]

$$\frac{i = j}{P;\ \Gamma \vdash \mathsf{world}[i] = \mathsf{world}[j]}$$

[= THIS]

$$\frac{\begin{array}{c} i = j \\ P;\ \Gamma \vdash_{owner} \mathsf{this} \end{array}}{P;\ \Gamma \vdash \mathsf{this}[i] = \mathsf{this}[j]}$$

[= TRANS]

$$\frac{\begin{array}{c} P;\ \Gamma \vdash o_1 = o_2 \\ P;\ \Gamma \vdash o_2 = o_3 \end{array}}{P;\ \Gamma \vdash o_1 = o_3}$$

[= REFL]

$$\frac{\begin{array}{c} P;\ \Gamma \vdash_{owner} o \\ P;\ \Gamma \vdash o \neq \mathsf{world} \qquad P;\ \Gamma \vdash o \neq \mathsf{this} \end{array}}{P;\ \Gamma \vdash o = o}$$

[≠ WORLD]

$$\frac{i \neq j}{P;\ \Gamma \vdash \mathsf{world}[i] \neq \mathsf{world}[j]}$$

[≠ THIS]

$$\frac{\begin{array}{c} i \neq j \\ P;\ \Gamma \vdash_{owner} \mathsf{this} \end{array}}{P;\ \Gamma \vdash \mathsf{this}[i] \neq \mathsf{this}[j]}$$

[≠ WORLD]

$$\frac{P;\ \Gamma \vdash_{owner} \mathsf{this}[i]}{P;\ \Gamma \vdash \mathsf{this}[i] \neq \mathsf{world}}$$

[SUBSTITUTION]

$$\frac{\begin{array}{c} P;\ \Gamma \vdash o_1 = o_2 \\ P;\ \Gamma \vdash constr \end{array}}{P;\ \Gamma \vdash constr\ [o_1/o_2]}$$

[RELATION]

$$\frac{P;\ \Gamma \vdash o_1 < o_2}{P;\ \Gamma \vdash o_1 \neq o_2}$$

$\boxed{P;\ E\ \vdash_{\mathrm{owner}} o}$

[OWNER WORLD]

$$\frac{}{P;\ \Gamma \vdash_{owner} \mathsf{world}[i]}$$

[OWNER FORMAL]

$$\frac{\Gamma = \Gamma',\ f : owner,\ \Gamma''}{P;\ \Gamma \vdash_{owner} f}$$

[OWNER THIS]

$$\frac{\Gamma = \Gamma',\ \mathsf{this} : owner,\ \Gamma''}{P;\ \Gamma \vdash_{owner} \mathsf{this}[i]}$$

$\boxed{P;\ E \vdash e : t}$

[EXP SUB]

$$\frac{\begin{array}{c} P;\ \Gamma \vdash e : t' \\ P;\ \Gamma \vdash t' <: t \end{array}}{P;\ \Gamma \vdash e : t}$$

[EXP NEW]

$$\frac{\begin{array}{c} P \vdash cn\langle f_{1..n}\rangle(t_i\ x_i\ ^{i \in 1..k})\ \{\ \ldots\ \}\ \in\ cn\langle f_{1..n}\rangle \\ P;\ \Gamma \vdash cn\langle o_{1..n}\rangle \qquad P;\ \Gamma \vdash e_i : t_i\ [o_1/f_1]..[o_n/f_n] \end{array}}{P;\ \Gamma \vdash \mathsf{new}\ cn\langle o_{1..n}\rangle(e_i\ ^{i\,\in\,1..k}) : cn\langle o_{1..n}\rangle}$$

[EXP VAR]

$$\frac{\Gamma\ =\ \Gamma',\ x : t,\ \Gamma''}{P;\ \Gamma \vdash x : t}$$

[EXP VAR ASSIGN]

$$\frac{P; \Gamma \vdash x : t \qquad P; \Gamma \vdash e : t}{P; \Gamma \vdash x = e : t}$$

[EXP LET]

$$\frac{P; \Gamma \vdash e' : t' \qquad P; \Gamma, x : t' \vdash wf \qquad P; \Gamma, x : t' \vdash e : t}{P; \Gamma \vdash \text{let}\,(t'\, x = e')\ \text{in}\, \{\, e \,\} : t}$$

[EXP REF]

$$\frac{P; \Gamma \vdash x : cn\langle o_{1..n}\rangle \qquad P \vdash t\ fd \in cn\langle f_{1..n}\rangle}{P; \Gamma \vdash x.fd : t\ [o_1/f_1]..[o_n/f_n]}$$

[EXP REF ASSIGN]

$$\frac{P; \Gamma \vdash x : cn\langle o_{1..n}\rangle \qquad P \vdash t\ fd \in cn\langle f_{1..n}\rangle \qquad P; \Gamma \vdash e : t\ [o_1/f_1]..[o_n/f_n]}{P; \Gamma \vdash x.fd = e : t\ [o_1/f_1]..[o_n/f_n]}$$

[EXP INVOKE]

$$\frac{\begin{array}{c} P \vdash t\ mn\langle f_{(k+1)..n}\rangle(t_i\ y_i\ ^{i \in 1..h})\ \text{where}\ constr^*\ \ldots \in cn\langle f_{1..k}\rangle \\ P; \Gamma \vdash x : cn\langle o_{1..k}\rangle \qquad P; \Gamma \vdash e_i : t_i\,[o_1/f_1]..[o_n/f_n] \qquad P; \Gamma \vdash constr\ [o_{k+1}/f_{k+1}]..[o_n/f_n] \end{array}}{P; \Gamma \vdash x.mn\langle o_{(k+1)..n}\rangle(e_{1..h}) : t\ [o_1/f_1]..[o_n/f_n]}$$

$$\boxed{P; \Gamma \vdash t <: t'}$$

[SUBTYPE]

$$\frac{P; \Gamma \vdash cn\langle o_{1..n}\rangle \qquad P \vdash cn\langle f_{1..n}\rangle \lhd cn'\langle f^+\rangle}{P; \Gamma \vdash cn\langle o_{1..n}\rangle <: cn'\langle f^+\rangle[o_1/f_1]..[o_n/f_n]}$$

[SUBTYPE TRANS]

$$\frac{P; \Gamma \vdash t <: t' \qquad P; \Gamma \vdash t' <: t''}{P; \Gamma \vdash t <: t''}$$

[SUBTYPE REFL]

$$\frac{P; \Gamma \vdash t}{P; \Gamma \vdash t <: t}$$

Bibliography

[1] Intel® Cilk$^{\text{TM}}$ Plus is now available in open-source and for gcc 4.7! `http://www.cilkplus.org`, 2011. The source code for the compiler and its associated runtime is available at `http://gcc.gnu.org/svn/gcc/branches/cilkplus`.

[2] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 185–196, Raleigh, NC, USA, 2009. ACM.

[3] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, June 2010.

[4] K. Agrawal, I.-T. A. Lee, and J. Sukha. Safe open-nested transactions through ownership. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 151–162, Raleigh, NC, USA, 2009. ACM.

[5] K. Agrawal, C. E. Leiserson, and J. Sukha. Memory models for open-nested transactions. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, San Jose, California, USA, Oct. 2006. In conjunction ASPLOS.

[6] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification Version 1.0.* Sun Microsystems, Inc., Mar. 2008.

[7] C. S. Ananian, K. Asanović, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. *IEEE Micro*, 26(1), Jan. 2006. Won the *IEEE Micro* "Top Picks" award for the most industry relevant and significant papers of the year in computer architecture.

[8] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, Puerto Vallarta, Mexico, June 1998.

[9] R. Barik, Z. Budimlić, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşırlar, Y. Yan, Y. Zhao, and V. Sarkar. The Habanero multicore software research project. In *Proceeding of the 24th ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, OOPSLA '09, pages 735–736, Orlando, Florida, USA, 2009. ACM.

[10] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-LX)*, pages 117–128, Cambridge, MA, Nov. 2000.

[11] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe multithreaded programming for c/c++. In *Proceedings of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 81–96, Orlando, Florida, USA, 2009. ACM.

[12] G. E. Blelloch. NESL: A nested data-parallel language (version 3.1). Technical Report CMU-CS-95-170, School of Computer Science, Carnegie Mellon University, Sept. 1995.

[13] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3), Mar. 1996.

[14] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, Santa Barbara, California, July 1995.

[15] R. D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, Sept. 1995. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-677.

[16] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, Padua, Italy, June 1996.

[17] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, California, July 1995.

[18] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25 1996. (An early version appeared in the *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 207–216, Santa Barbara, California, July 1995.).

[19] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, Feb. 1998.

[20] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, Sept. 1999.

[21] R. D. Blumofe and D. Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical Report, University of Texas at Austin, 1999.

[22] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, New Orleans, Louisiana, Jan. 2003.

[23] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, Apr. 1974.

[24] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 187–194, Portsmouth, New Hampshire, Oct. 1981.

[25] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional collection classes. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, pages 56–67, San Jose, California, USA, 2007. ACM.

[26] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, New York, NY, USA, 2005.

[27] Cilk Arts, Inc. *Cilk++ Programmer's Guide*, release 1.0 edition, December 2008.

[28] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 67–78, Bangalore, India, 2010. ACM.

[29] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 336–346, San Jose, California, USA, 2006. ACM.

[30] J. S. Danaher, I.-T. A. Lee, and C. E. Leiserson. Programming with exceptions in JCilk. *Science of Computer Programming*, 63(2):147–171, Dec. 2006.

[31] P. J. Denning. Virtual memory. *Computing Surveys*, 2(3):153–189, Sept. 1970.

[32] P. J. Denning. Before memory was virtual. In *In the Beginning: Personal Recollections of Software Pioneers*, Nov. 1996.

[33] D. Dice. David dice's weblog. `https://blogs.com/dave/entry/biased_locking_in_hotspot#comments`, 2006.

[34] D. Dice, H. Huang, and M. Yang. Asymmetric Dekker synchronization. Technical report, Sun Microsystems Inc., July 2001.

[35] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 157–168, Washington, DC, USA, 2009. ACM.

[36] D. Dice, M. Moir, and W. S. III. Quickly reacquirable locks. Technical report, Sun Microsystems Inc., 2003.

[37] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *In Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[38] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569, Sept. 1965.

[39] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, London, England, 1968. Originally published as Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.

[40] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Trans. Comput.*, 38(3):408–423, Mar. 1989.

[41] R. Feldmann, P. Mysliwietz, and B. Monien. Studying overheads in massively parallel min/max-tree evaluation. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 94–103, Cape May, New Jersey, June 1994.

[42] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 1–11, Newport, Rhode Island, June 1997.

[43] R. Finkel and U. Manber. DIB — A distributed implementation of backtracking. *ACM TOPLAS*, 9(2):235–256, Apr. 1987.

[44] J. Fotheringham. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. *Communications of the ACM*, 4(10):435–436, Oct. 1961.

[45] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 201–213, Monterey, California, Nov. 1994.

[46] M. Frigo. *Portable High-Performance Programs*. PhD thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, Cambridge, Massachusetts, June 1999.

[47] M. Frigo, 2009. Private communication.

[48] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the Twenty-First Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 79–90, Calgary, Canada, Aug. 2009. Won Best Paper award.

[49] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[50] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl*, 13:333–356, 1992.

[51] S. C. Goldstein, K. E. Schauser, and D. Culler. Enabling primitives for compiling parallel languages. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Troy, New York, May 1995.

[52] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, second edition, 2000.

[53] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45:1563–1581, Nov. 1966.

[54] M. Halbherr, Y. Zhou, and C. F. Joerg. MIMD-style parallel programming with continuation-passing threads. In *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software, and Applications*, Capri, Italy, Sept. 1994.

[55] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, Oct. 1985.

[56] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 102–113, M ddotunchen, Germany, June 2004.

[57] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[58] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 14–25, Ottawa, Ontario, Canada, 2006. ACM.

[59] J. M. Hart. *Windows System Programming*. Addison-Wesley, third edition, 2004.

[60] E. A. Hauck and B. A. Dent. Burroughs' B6500/B7500 stack mechanism. *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 245–251, 1968.

[61] J. L. Hennessy and D. A. Patterson. *Computer Architecture: a Quantitative Approach*. Morgan Kaufmann, San Francisco, California, USA, fourth edition, 2007.

[62] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Salt Lake City, Utah, USA, Feb. 2008. ACM.

[63] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

[64] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Conference on Computer Architecture. (Also published as ACM SIGARCH Computer Architecture News, Volume 21, Issue 2, May 1993.)*, pages 289–300, San Diego, California, 1993.

[65] 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.

[66] Intel Corporation. *Intel Cilk++ SDK Programmer's Guide*, Oct. 2009. Document Number: 322581-001US.

[67] Intel Corporation. *Intel® C++ Compiler 12.0 User and Reference Guides*. Intel Corporation, 2010. Document number: 323271-011US.

[68] Intel Corporation. *Intel® Cilk$^{\mathrm{TM}}$ Plus Application Binary Interface Specification*, 2010. Available at `http://software.intel.com/sites/products/cilk-plus/cilk_plus_abi.pdf`.

[69] Intel Corporation. *Intel® Cilk$^{\mathrm{TM}}$ Plus Language Specification*, 2010. Available at `http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf`.

[70] Intel Corporation. *C++ and C interfaces for Cilk reducer hyperobjects*. Intel Corporation, 2011. Intel® C++ Compiler 12.0: `reducer.h` Header File.

[71] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, Jan. 2011.

[72] E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, Livermore, California, Nov. 1987.

[73] C. Joerg and B. C. Kuszmaul. Massively parallel chess. In *Proceedings of the Third DIMACS Parallel Implementation Challenge*, Rutgers University, New Jersey, Oct. 17–19 1994.

[74] C. F. Joerg. *The Cilk System for Parallel Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, Jan. 1996. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-701.

[75] R. M. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM*, 40(3):765–789, July 1993.

[76] K. Kawachiya, A. Koseki, and T. Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 130–141, Seattle, Washington, USA, Nov. 2002.

[77] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., second edition, 1988.

[78] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *IRE Trans. Electronic Computers*, (2):223–235, Apr. 1962.

[79] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, J. Guy L. Steele, and M. E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[80] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-T: A high-performance parallel Lisp. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 81–90, Portland, Oregon, June 1989.

[81] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 209–220, New York, New York, USA, 2006. ACM.

[82] B. C. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1994. Available as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-645.

[83] B. C. Kuszmaul. The StarTech massively parallel chess program. *The Journal of the International Computer Chess Association*, 18(1):3–20, Mar. 1995.

[84] E. Ladan-Mozes, I.-T. A. Lee, and D. Vyukov. Location-based memory fences. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 75–84, San Jose, California, USA, 2011. ACM.

[85] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[86] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

[87] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. *ACM Transactions on Computer Systems*, 10:265–310, Nov. 1992.

[88] J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. *Softw. Pract. Exper.*, 24(2):197–218, 1994.

[89] C. Lasser and S. M. Omohundro. *The Essential *Lisp Manual, Release 1, Revision 3.* Thinking Machines Technical Report 86.15, Cambridge, MA, 1986.

[90] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 36–43. ACM, 2000.

[91] I.-T. A. Lee, S. Boyd-Wickizer, Z. Huang, and C. E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *PACT '10: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 411–420, Vienna, Austria, September 2010. ACM.

[92] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Proceeding of the 24th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 227–242, Orlando, Florida, USA, 2009.

[93] C. E. Leiserson. *Encyclopedia of Distributed Computing*. Joseph Urban and Partha Dasgupta, editors, Kluwer Academic Publishers. to appear.

[94] C. E. Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51(3):244–257, March 2010.

[95] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing*, 33(2):145–158, 1996.

[96] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 303–314, June 2010.

[97] Y. Lev and J.-W. Maessen. Split hardware transactions: True nesting of transactions using best-effort hardware transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 197–206, Salt Lake City, UT, USA, 2008. ACM.

[98] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *The 2nd ACM SIGPLAN Workshop on Transactional Computing (Transact)*, Portland, Oregon, USA, Aug. 2007.

[99] C. Lin, V. Nagarajan, and R. Gupta. Efficient sequential consistency using conditional fences. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 295–306, Vienna, Austria, Sept. 2010. ACM.

[100] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Boston, Massachusetts, second edition, 2000.

[101] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: Efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, Cascais, Portugal, 2011. ACM.

[102] V. J. Marathe, W. N. S. Iii, and M. L. Scott. Adaptive software transactional memory. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC)*, pages 354–368, Cracow, Poland, Sept. 2005.

[103] M. Matz, J. Hubička, A. Jaeger, and M. Mitchell. System V application binary interface AMD64 architecture processor supplement draft version 0.99.

[104] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, Apr. 1960.

[105] D. McCrady. Avoiding contention using combinable objects. Microsoft Developer Network blog post, Sept. 2008.

[106] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, June 2006.

[107] E. Meijer and J. Gough. Technical overview of the common language runtime. http://research.microsoft.com/en-us/um/people/emeijer/Papers/CLR.pdf, 2000.

[108] J. M. Mellor-Crummey and M. L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Prgoramming (PPoPP)*, pages 106–113, Williamsburg, Virginia, United States, 1991. ACM.

[109] MIPS Computer Systems, Inc. *RISCompiler Languages Programmer's Guide*, December 1988.

[110] G. E. Moore. Progress in digital integrated electronics. In *International Electron Devices Meeting Technical Digest*, pages 11–13, Dec. 1975.

[111] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture (HPCA)*, pages 254–265, Austin, Texas, USA, Feb. 2006.

[112] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. The MIT Press, Cambridge, Massachusetts, USA, 1985.

[113] J. E. B. Moss. Open nested transactions: Semantics and support. In *Proceedings of the Workshop on Memory Performance Issues (WMPI)*, Austin, Texas, Feb. 2006.

[114] J. E. B. Moss and A. L. Hosking. Nested transactional memory: Model and architecture sketches. 63(2):186–201, Dec. 2006.

[115] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 89–100, New York, NY, USA, 2007. ACM.

[116] R. H. B. Netzer and B. P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1):74–88, March 1992.

[117] Y. Ni, V. Menon, A. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, Mar. 2007.

[118] R. S. Nikhil. Cid: A parallel, shared-memory C for distributed-memory machines. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, Aug. 1994.

[119] T. Onodera, K. Kawachiya, and A. Koseki. Lock reservation for java reconsidered. In *Proceedings of the 18th European Conference on Object-Oriented Programming*, pages 559–583, Oslo, Norway, June 2004. Springer Berlin / Heidelberg.

[120] OpenMP application program interface, version 3.0. OpenMP specification, May 2008.

[121] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[122] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, June 1981.

[123] H. K. Pyla and S. Varadarajan. Avoiding deadlock avoidance. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 75–86, Vienna, Austria, 2010. ACM.

[124] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, Madison, Wisconsin, USA, June 2005.

[125] K. H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998.

[126] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.* O'Reilly Media, Inc., 2007.

[127] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 187–197, New York, NY, USA, 2006. ACM.

[128] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, Ottowa, Ontario, Canada, Aug. 1995.

[129] D. Stein and D. Shah. Implementing lightweight threads. In *USENIX '92*, pages 1–9, 1992.

[130] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Boston, MA, third edition, 2000.

[131] J. Sukha. Brief announcement: A lower bound for depth-restricted work stealing. In *The Twenty-first ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, Canada, Aug. 2009.

[132] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk 5.4.6 Reference Manual*, 2006.

[133] M. T. Vandevoorde and E. S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, Aug. 1988.

[134] N. Vasudevan, K. S. Namjoshi, and S. A. Edwards. Simple and fast biased locks. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 65–74, Vienna, Austria, Sept. 2010. ACM.

[135] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual, Version 9.* PTR Prentice Hall, 1994.

[136] G. Weikum. A theoretical foundation of multi-level concurrency control. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*, pages 31–43, Cambridge, Massachusetts, United States, 1986. ACM.

[137] W. Wulf and M. Shaw. Global variable considered harmful. *SIGPLAN Notices*, 8(2):28–34, 1973.