Safe Concurrent Programming and Execution

Hari K. Pyla

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Srinidhi Varadarajan, Chair
Calvin Ribbens, Co-Chair
Vikram Adve
Doug Lea
Naren Ramakrishnan
Eli Tilevich

February 26, 2013
Blacksburg, Virginia

Keywords: Concurrent Programming, Concurrency Bugs, Program Analysis, Runtime Systems, Deadlock Detection and Recovery, Speculative Parallelism, and Coarse-grain Speculation.

Safe Concurrent Programming and Execution

Hari K. Pyla

(ABSTRACT)

The increasing prevalence of multi and many core processors has brought the issues of concurrency and parallelism to the forefront of everyday computing. Even for applications amenable to traditional parallelization techniques, the subtleties of concurrent programming are known to introduce concurrency bugs. Due to the potential of concurrency bugs, programmers find it hard to write correct concurrent code. To take full advantage of parallel shared memory platforms, application programmers need safe and efficient mechanisms that can support a wide range of parallel applications.

In addition, a large body of applications are inherently hard-to-parallelize; their data and control dependencies impose execution order constraints that preclude the use of traditional parallelization techniques. Sensitive to their input data, a substantial number of applications fail to scale well, leaving cores idle. To improve the performance of such applications, application programmers need effective mechanisms that can fully leverage multi and many core architectures.

These challenges stand in the way of realizing the true potential of emerging many core platforms. The techniques described in this dissertation address these challenges. Specifically, this dissertation contributes techniques to transparently detect and eliminate several concurrency bugs, including deadlocks, asymmetric write-write data races, priority inversion, live-locks, order violations, and bugs that stem from the presence of asynchronous signaling and locks. A second major contribution of this dissertation is a programming framework that exploits coarse-grain speculative parallelism to improve the performance of otherwise hard-to-parallelize applications.

Acknowledgments

My graduate school experience has been an exciting journey and it has provided me with a wonderful opportunity to learn. This experience has made me realize that life could be viewed as a non-deterministic finite state machine and that every state transition presents an opportunity for improvement. During this journey, I've had the pleasure and privilege of meeting with several individuals who have in ways, both big and small, have enriched my graduate school experience and they have positively contributed to the success of my Ph.D.

I would like thank my family for their love, support, and encouragement. I probably would not be writing this dissertation if it weren't for my parents who have provided me with the foundations for a good education. My family has instilled in me that success can be achieved with honesty, hard work, and perseverance. This dissertation is a testament of their trust and many sacrifices. I am extremely fortunate to have a wonderful brother. I would like to thank my brother for everything. He has led me by example — he graduated with a B.S, M.S, M.S, Ph.D., and Post-doc. His constant encouragement and guidance at all stages of my graduate education has helped me successfully complete my Ph.D.

I would like to express my deep gratitude to my advisor Srinidhi Varadarajan, co-advisor Calvin Ribbens, and the members of my Ph.D. committee — Eli Tilevich, Naren Ramakrishnan, Doug Lea, and Vikram Adve.

I would like to thank my advisor, Srinidhi Varadarajan for everything. This dissertation would not have been successfully completed without his support, guidance, and encouragement. I would also like to thank him for his trust and respect. He provided me with a wonderful learning environment

and the freedom to explore areas that were of interest to me. I believe very few advisors provide such a degree of freedom to their graduate students. Srindhi's technical insights, analytical abilities, and uncanny eye for details is inspiring and amazing. His ability to context switch and yet restore focus to the core of the problem is hard to fathom and sometimes intimidating. I feel fortunate to have such a wonderful advisor who has taught me many things and enriched my graduate experience. I would like to thank him for his patience and all the wonderful and intellectual discussions he has had with me over the years.

I would also like to thank my co-advisor, Calvin Ribbens for his support and guidance during my graduate studies. I wouldn't have successfully completed this dissertation without his guidance. I deeply appreciate his feedback during our brainstorming sessions and technical discussions. I admire his patience and I am particularly grateful for his prompt feedback that helped me improve my ability to write technical documents. I feel fortunate to have such an amazing co-advisor and it has been my pleasure working with him during my graduate studies.

I would like to thank Eli Tilevich for his support and encouragement. Eli has been instrumental in my success in grad school. His unique sense of humor has restored a sense of sanity to my graduate studies and made the time spent in grad school fun. I am thankful to Eli for teaching me how to survive in graduate school. He taught me how to communicate ideas efficiently and effectively. I would like to thank him for his insightful comments and invaluable feedback that helped me improve my ability to write technical documents. I am extremely fortunate to have an amazing mentor like him and I would like to thank Eli for enriching my Ph.D. experience.

I would like to thank Naren Ramakrishnan for his support and encouragement. Despite his busy schedule he always managed to provide quick and prompt responses to my emails. His ability to manage time and his work ethic are particularly inspiring. I am thankful to him for his feedback on my dissertation document.

I feel extremely fortunate to have had the opportunity to interact with Doug Lea and Vikram Adve. It has been a wonderful learning experience. I am deeply honored and fortunate to have Doug Lea as my external program committee member. I've had the pleasure of meeting with him

at the OOPSLA/SPLASH conference in 2011. I deeply appreciate his insightful comments during the prelim and the final exam. I am particularly thankful to him that he took time from his busy schedule to not only read my dissertation draft but also provide me with invaluable feedback.

I am also deeply honored and fortunate to have Vikram Adve as my external program committee member. I would like to thank him for taking time from his busy schedule to provide me with invaluable feedback during my prelim and final defense.

During my graduate years I've had the pleasure of meeting with several wonderful people. I would like to thank my friend Rajesh Sudarsan for his encouragement and patience during our technical discussions and providing me with positive feedback.

I would like to thank Vedavyas Duggirala (Vyas) for providing me with his insightful feedback during our white board brainstorming sessions and technical discussions. I appreciate his feedback on my research papers. I would also like to thank Vyas and Shankha Banerjee for helping me get me up to speed with LLVM. I would also like to thank Scott Schneider for his feedback during our technical discussions and his comments on my research papers.

I would like to thank Rajesh Sudarsan, Vedavyas Duggirala, Scott Schneider, Bharat Ramesh, Shankha Banerjee, and Balaji Subramaniam for their company and all the wonderful discussions we've had during our tea breaks in the lab.

I would like to thank my friends Rajaram Bhagavathula and Gayatri Ankem for their support and encouragement. They made my graduate school experience a memorable one. I feel fortunate to have such wonderful friends and colleagues who had a positive impact during my Ph.D. studies.

I would like to thank Aleksandr Khasymski (Alex) for providing me with access to the multicore machines during the initial stages of my dissertation. I would like to thank Ryan Chase and Rob Hunter for helping me with several of the IT issues. I deeply appreciate their help.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last decade, processor design has undergone a paradigm shift. Rather than increasing the clock frequency, processors feature an increasing number of simpler cores with reduced clock frequency and shorter pipelines. No longer able to rely on increasing clock frequencies to boost performance, application programmers have focused their attention on concurrent programming, especially multi-threading. The increasing prevalence of multi and many core processors has brought the issues related to concurrency and parallelism to the forefront of everyday computing.

On one hand, we have applications that are amenable to traditional parallelization techniques but are susceptible to the subtleties of concurrent programming — concurrency bugs. In practice, programmers find it hard to write correct concurrent code due to the potential of concurrency bugs. The non-determinism of thread execution only exacerbates the complexity of detecting and isolating concurrency bugs. *The challenge here is to provide safe and efficient mechanisms to enable a large number of programmers, representing a wide array of applications, to use these parallel shared memory platforms effectively.*

On the other hand, we have a large body of applications that are inherently hard-to-parallelize due to execution order constraints imposed by data and control dependencies. Moreover, a significant number of applications are sensitive to their input data and do not scale well, leaving several cores idle. *The challenge here is to enable such applications to leverage multi and many core architectures efficiently in order to improve their performance.*

Unless we address these challenges, the true potential of these emerging many-core platforms will probably go unrealized. This dissertation presents techniques that address these challenges. We first explore these issues in detail and then outline our solutions and key contributions.

1.1 Safe Concurrent Execution

Large software systems are developed by hundreds or even thousands of programmers, often spanning multiple teams, making it hard to maintain the strict coding discipline required to avoid concurrency bugs. The software industry has been spending billions of dollars each year to ensure software quality and perform testing. Unfortunately, despite these efforts, it is not feasible to exercise all possible code paths and thread interleavings to declare a code bug free [45, 39].

Detecting a concurrency bug does not imply that it can be easily fixed. Concurrency bugs often require careful reasoning to identify the root cause of the problem and not merely where the effect of the bug was manifested in the program code. Additionally, such bugs are hard to reproduce, and the bug fix may require a major redesign. For example, when addressing a deadlock in Mozilla, the programmer introduced another deadlock which took several months to a year to fix [103]. In a different Mozilla bug, the programmers intentionally introduced a data-race to address the actual deadlock [103]. Recent studies have shown that the patches developed to fix a bug are themselves error-prone (70% of the time in their first release) and they introduce new bugs [39]. On average, a concurrency bug fix takes about three fixes (patches) before it is actually fixed [54, 39].

In practice, recurrent concurrency bugs are *data races* and *deadlocks*. While data races can be ameliorated with appropriate synchronization (a challenging problem in itself!), avoiding deadlocks requires convoluted avoidance techniques, which may fail if the order of lock acquisitions is not known a priori [75]. Fine-grain locking significantly exacerbates deadlock avoidance issues, to the point that it is generally eschewed in favor of simpler less competent locking models that are deadlock free. Furthermore, due to the potential for deadlocks, programmers cannot arbitrarily compose locks without knowing the internal locking structure, thus limiting the *composition* of

lock based code.

The non-composability of lock based code has lead to significant research interest in the transactional memory (TM) programming model (e.g., [36, 91]). However, the vast majority of existing large-scale code bases use the relatively well-understood (warts and all!) lock based model [59]. The dominant shared memory programming model continues to be the thread model, and the most common system supporting thread-based parallel programming is POSIX threads (Pthreads). Whether Pthreads is ultimately replaced by other models, a large body of concurrent codes remain based on Pthreads, and it is likely that any future higher-level abstractions will rely on lower-level runtime systems, whose semantics is similar to that of Pthreads. So writing, porting, extending, composing, and debugging Pthreads codes is a critical piece of the puzzle when it comes to exploiting concurrency on modern many-core machines.

Our observation here is that by starting with a lock based model and by transparently eliminating data races and deadlocks, thereby ensuring composability of arbitrary lock based codes, one can provide *safe concurrent execution*. An added advantage to this line of thought is that programmers are already familiar with lock based programming and fixing issues in lock based programming can benefit the large base of lock based applications in use today.

To that end, this dissertation presents several techniques to detect and eliminate concurrency bugs including deadlocks, asymmetric write-write data races, priority inversion, live-locks, order violations, and bugs that stem from the presence of asynchronous signaling and locks. We first present a pure runtime system (Sammati [75, 73]) that is capable of transparently and deterministically detecting and eliminating deadlocks in POSIX multi-threaded applications written in type-unsafe languages, including C and C++. Leveraging the lessons learned, we then extend this work to a comprehensive program analysis and runtime based approach (Serenity [76]) that eliminates deadlocks and related concurrency bugs.

1.1.1 Pure Runtime Approach

Sammati [75] (*agreement* in Sanskrit) guarantees the acquisition of a mutual exclusion lock to be a deadlock free operation. It *transparently* and *deterministically* eliminates deadlocks, without requiring any modifications to the source code, compiler, or the operating system. Sammati is implemented as a pre-loadable library that overloads the standard POSIX threads (Pthreads) interface. It operates on native binaries compiled to the standard Executable and Linkable Format (ELF). Sammati works by associating memory accesses with locks and privatizing memory updates within a critical section. Memory updates within a critical section are made visible outside the critical section on the release of the parent lock(s), viz. the containment property. On the acquisition of every lock, Sammati runs a single cycle deadlock detection algorithm. If a deadlock is detected, the Sammati deadlock elimination algorithm breaks the cycle by selecting a victim, rolling the program state back to the acquisition of the offending lock, and discarding any memory updates. A special containment mechanism ensures that memory updates from a critical section are not visible outside the critical section until a successful release, thus enabling Sammati to simply restart the critical section to recover from the deadlock.

Note that this containment property is in principle similar to transactional memory systems but with the following critical difference. Sammati preserves the mutual exclusion semantics (and more importantly limitations) of existing lock based codes and does not provide any mechanisms to optimistically execute critical sections concurrently as in transactional memory systems.

While conceptually simple, this approach requires an *efficient* and *transparent* mechanisms to (a) identify updates within a critical section, (b) isolate (contain) the memory updates, (c) preserve existing lock semantics, while still permitting containment based deadlock detection and recovery in the presence of nested locks, (d) perform inclusion of memory updates from the contained critical section into the shared program address space, and (e) perform deadlock detection and recovery that deterministically eliminates deadlocks without either deadlocking itself or requiring an outside (external) agent. In this research, we propose and implement techniques that address the above design objectives.

Unlike managed languages such as Java, in the case of weakly typed languages such as C or C++, which allow arbitrary memory accesses, program analysis cannot always determine the exact write set. Hence based on our key observation that privatization can be implemented efficiently and transparently in a runtime environment if each thread has its own virtual address space, we implement containment for threaded codes by creating multiple processes that share their global data regions through a common shared memory mapping. In essence, this creates a set of processes (we refer to them as *cords*) that are semantically equivalent to threads — they share their global data (globals, heap) and have distinct stacks.

Sammati tracks writes to shared data (global data and heap) through access faults. To achieve containment for a page, the Sammati runtime system breaks its binding to the shared memory region and creates a private page mapping at the same virtual address. Any updates to the private page are thus localized to the thread executing the critical section, thereby implementing containment. Intuitively, Sammati implements a lazy privatization scheme that defers privatization to the instant when the first memory update occurs. This is a sufficient [75] condition for standard lock semantics of mutual exclusion locks. On a related note, Sammati includes a novel technique that leverages the large virtual memory address (VMA) provided by 64-bit operating systems in order to counter the side effects of privatization on mutual exclusion and other synchronization primitives.

To ensure containment of memory updates in the presence of nested locks, Sammati makes memory updates visible on the release of *all* locks. This is a necessary and sufficient condition for Sammati's deadlock elimination and recovery. Furthermore, in order to preserve the semantics of mutual exclusion, Sammati tracks the release of nested locks in program order and defer performing the actual release till all locks around a critical section have been released in program order.

When a cord exits a lock context, updates contained in its privatized data must be made visible and reconciled with other cords. In order to perform this *inclusion*, we propose a technique that (a) identifies the exact write set of the lock context and (b) concurrently propagates the updates to memory pages as a commutative operation. The updates are relatively straightforward for a single lock around a critical section — updates are visible at the release of the lock. However, inclusion in

the presence of nested locks is subtle and we propose a set of visibility rules [75] to accommodate nested locks.

Since threads have been transparently converted to cords in Sammati, the key observation here is that each cord (a single threaded process) may only wait on at most one lock. This significantly simplifies deadlock detection, since from the perspective of each cord, all deadlocks are single cycle deadlocks. Furthermore, since all cords share a global address space, the deadlock detection algorithm has access to the holding and waiting sets of each cord. Deadlock detection can hence be performed at lock acquisition with the guarantee that the deadlock detection algorithm itself cannot be deadlocked. The proposed algorithm detects a deadlock and identifies a *victim* lock for deadlock recovery.

The deadlock recovery algorithm then finds the oldest acquisition of the victim lock in program order and uses its associated recovery point for recovery. To recover from the deadlock Sammati (a) discards all memory updates performed by locks acquired later in program order after the victim lock, (b) releases all locks acquired after and including the victim lock (in program order), and (c) restores the stack and processor registers from the recovery point for the victim lock, before transferring control back to deadlock free lock acquisition. Note that the reasoning behind using the recovery point from the oldest (in program order) acquisition of the victim lock is subtle and is explained in Chapter 2.

To study the impact of Sammati on threaded applications, we evaluated its performance using SPLASH [94], Phoenix [80] and synthetic benchmark suites on a 16 core shared memory machine (NUMA) running Linux with 64GB of RAM. We measured the number of locks acquired and lock-acquisition-rate (total locks acquired/total runtime) for all applications used in this study. We also evaluated Sammati by running synthetic programs that were deadlock prone. We find that the native pthreads programs deadlock while Sammati successfully detects and avoids the deadlocks, transparently recovers from them and executes the program to completion.

On the whole, Sammati is promising and performs reasonably well for several applications under study, however we find that the address space protection and privatization costs primarily contribute

to the runtime overhead. Additionally, although a pure runtime approach does not require source code and operates with native binaries, there are some limitations. We present a detailed discussion of the design, implementation, and performance evaluation of Sammati in Chapter 2.

1.1.2 Research Contributions

The primary contribution of this work is a **pure runtime approach that transparently and deterministically eliminates deadlocks — enabling a next generation of safe composable lock based codes**. The proposed approach in detecting deadlocks does not encounter any false positives or false negatives and does not require any modifications to the application source code, compiler, or the operating system. The following are specific contributions:

(i) An execution model that provides the ability to selectively share and isolate state between execution contexts — a platform for containment based deadlock recovery.

(ii) Techniques that transparently and efficiently detect, isolate, and privatize memory updates.

(iii) Semantics for propagating memory updates while preserving program correctness and containment based deadlock recovery.

(iv) Techniques to address potential side-effects of privatization.

(v) An efficient algorithm to detect deadlocks without the presence of an external agent. The worst case time complexity is $\mathrm{O}(n)$, where n is the number of threads currently waiting for a lock.

(vi) Techniques to provide granular deadlock recovery — roll back the program state only to the offending lock as opposed to rolling back the program state all the way to the start of the outermost critical section.

1.1.3 Program Analysis and Runtime Approach

In our attempt to answer *"can we design a system that is competent and practical?"* we iteratively improvised over several of our techniques to bring the runtime overhead from over 300% to (a) a level that is suitable for practical use with real applications, (b) be able to sustain applications that acquire over billions of locks with hundreds of millions of locks/sec, and (c) scale well with the number of threads. We realize this vision as Serenity [76].

Our observation here is that program analysis and instrumentation can guide a runtime to efficiently achieve isolation. Serenity deploys program analysis techniques and employs opportunistic optimizations that associate a lock acquisition with its corresponding release, thereby deriving the *scope* (*the code executed between the acquisition of a lock and its corresponding release*) of the critical section in a program. This enables Serenity's compile time infrastructure to perform efficient load/store instrumentation of critical sections, which is significantly cheaper than instrumenting the entire program. We refer to this analysis as *lock scope analysis*. Our lock scope analysis is motivated by our observation on how programmers typically structure lock based code. We refer to this observation as the *lock scope idiom*.

Serenity performs program analysis using the Low Level Virtual Machine (LLVM) [51] compiler framework. Serenity transforms program source to LLVM intermediate representation (IR) and employs lock scope analysis to perform load/store instrumentation. Serenity replaces the load and store instructions within a critical section in the LLVM's IR with hooks (inlined) to Serenity's runtime system. The hooks pass information to the runtime system about the kind of instruction (load or a store), the target address (precisely known only at runtime), its length and type information, thus allowing Serenity's runtime to provide efficient isolation. Serenity's *shadowing* technique performs privatization efficiently at runtime while preserving program order, guaranteeing sequential consistency among loads and stores, and providing nested shadowing (shadow of shadows) at no additional runtime cost — a key capability required for granular deadlock recovery. Serenity leverages containment and propagation techniques from Sammati and improves upon Sammati's deadlock detection and recovery algorithms. Chapter 3 explores these techniques in detail.

We set out to determine how Serenity performs on real applications. To that end, we used a 64 core NUMA machine running Linux with 256GB of RAM. We applied Serenity to 24 real world lock based applications from the SPLASH [94], Phoenix [80], and PARSEC [12] benchmarks suites and several commonly used desktop and server applications including Pbzip2 [68], Tgrep [61], Squid [95] (a webcache and proxy server), and Sendmail [66]. To evaluate the effectiveness of Serenity we applied it to real applications containing deadlocks including SQLite-3.3.3 [24] and to a set of synthetic benchmarks with artificially seeded deadlocks. Serenity was able to eliminate all the deadlocks, while the rest of the execution continued unperturbed. Our results indicate that Serenity can be used on production systems and it comfortably supports applications that acquire billions of locks with lock rates of over hundreds of millions of lock/sec. Serenity is efficient and its performance is comparable to native thread execution for most applications with modest memory overhead. In Chapter 3 we present a detailed discussion of the experimental evaluation and related work to place Serenity in the context of existing literature.

1.1.4 Research Contributions

The primary contribution of this work is **a practical approach that can be used on production systems to eliminate deadlocks, and concurrency bugs affecting lock based codes**. Similar to our pure runtime approach, this work in detecting deadlocks does not encounter any false-positives or false-negatives and it does not require any modifications to the application source-code, compiler, or the operating system. The proposed approach scales very well and supports applications that acquire billions of locks with lock rates of over hundreds of millions of lock/sec. To the best of our knowledge, there is no other system in the literature that is capable of delivering this level of performance. The following are specific contributions:

(i) Reachability and flow analysis techniques, and opportunistic optimizations to derive the scope of critical sections.

(ii) Efficient runtime shadowing technique to provide isolation of memory updates.

(iii) Techniques to transparently support rollback aware memory management, and I/O within critical sections that may be affected by deadlock recovery, priority inversion, and live-locks.

(iv) Techniques to transparently and efficiently detect and eliminate the priority inversion problem.

(v) Techniques to transparently detect other common concurrency bugs and programming errors affecting lock based code such as asymmetric write-write data races, order violations, asynchronous signaling, and locks and their performance issues.

In addition to the research contributions, we demonstrate via a comprehensive experimental evaluation (Chapter 3) that our proposed approach accomplishes its intended design objectives.

1.2 Safe Concurrent Programming

As the number of cores in modern processor architectures keeps increasing, programmers must use explicit parallelism to improve performance. Alas, a large body of applications are intrinsically unsuitable for mainstream parallelization techniques, due to the execution order constraints imposed by their data and control dependencies. Therefore, realizing the potential of many-core hinges on our ability to parallelize these so called 'un-parallelizable' codes.

Multiple application domains posses this dilemma — how to choose the right algorithm when algorithm performance is dependent on input data. Graph coloring is a good example of this problem. This problem underlies the foundation of diverse domains including job scheduling, bandwidth allocation, pattern matching, and compiler optimization (register allocation). Several state-of-the-art approaches that solve the graph coloring problem employ probabilistic and meta-heuristic techniques. The performance of these techniques vary widely with the input parameters including nature of the graph, number of colors, etc. In addition to this sensitivity to the input, algorithms for the graph coloring problem are hard to parallelize due to inherent data dependencies.

Another example is the numerical solution of partial differential equations (PDEs). PDE solvers are a dominant component of large scale simulations arising in computational science and engi-

neering applications such as fluid dynamics, weather and climate modeling, structural analysis, and computational geosciences. The large, sparse linear systems of algebraic equations that correspond to discretized PDE problems are usually solved using preconditioned iterative methods. Unfortunately, the performance of such solvers can vary widely from problem to problem, even for a sequence of problems that may be related in some way, e.g., problems corresponding to discrete time steps in a time-dependent simulation. The problem is that the best iterative method is not known a priori.

Similar examples can be found in widely used combinatorial problems including sorting, searching, permutations and partitions where theoretical algorithmic bounds are well known, but in practice the runtime of an algorithm depends on a variety of factors including the amount of input data (algorithmic bounds assume asymptotic behavior), the sortedness of the input data, and cache locality of the implementation [5].

Our observation here is that speculative execution at coarse granularities (e.g., code-blocks, methods, algorithms) offers a promising alternative for exploiting parallelism in many hard-to-parallelize applications on multicore architectures. This dissertation abstracts the subtleties of concurrency and provides expressive semantics to exploit *coarse-grain speculative parallelism* in such hard-to-parallelize applications via a *safe concurrent programming* environment.

1.2.1 Coarse-Grain Speculative Parallelism

This dissertation presents Anumita (*guess* in Sanskrit) [74, 72, 73], a simple speculative programming framework where multiple coarse-grain speculative code blocks execute concurrently, with the results from a single speculation ultimately modifying the program state. Anumita consists of a shared library, which implements the framework API for common type-unsafe languages including C, C++ and Fortran, and a user-level runtime system that transparently (a) creates, instantiates, and destroys speculative control flows, (b) performs transparent name-space isolation, (c) tracks data accesses for each speculation, (d) commits the memory updates of successful speculations, and (e) recovers from memory side-effects of any mis-predictions. In the context of high-performance

computing applications, where the OpenMP threading model is prevalent, Anumita also provides a new OpenMP pragma to naturally extend speculation into an OpenMP context. To the best of our knowledge, Anumita is the first system to provide support for exploiting coarse-grain speculative parallelism in OpenMP based applications.

Anumita works by associating the memory accesses made by each speculation flow (e.g., an instance of a code block or a function) in a speculation *composition* (loosely, a collection of possible code blocks that execute concurrently). Anumita localizes these memory updates and provides isolation among speculation flows through privatization of address space. Ultimately, a single speculation flow within a composition is allowed to modify the program state. Anumita presents well-defined semantics that ensure program correctness for propagating the memory updates. Anumita is designed to support a wide range of applications (both sequential and parallel) by providing expressive evaluation criteria for speculative execution that go beyond *time to solution* to include arbitrary *quality of solution* criteria. Anumita is implemented as a language independent runtime system and its use requires minimal (around 8-10 lines) modifications to existing application source code. These modifications are short and often require little to no understanding of the applications themselves.

We evaluated Anumati using several micro benchmarks and three real applications: a multi-algorithmic PDE solving framework [83], a graph (vertex) coloring problem [57] and a suite of sorting algorithms [98]. Our experimental results [74] indicate that Anumita is capable of significantly improving the performance of hard-to-parallelize and input sensitive applications by leveraging speculative parallelism. For instance, in the PDE solver the speedup ranged from 0.84 to 36.19, for the graph coloring problem it ranged from 0.95 to 7.33, and for the sort benchmark it ranged from 0.84 to 62.95. Using Anumita it is possible to obtain the best solution among multiple heuristics. We found that in some cases where heuristics failed to arrive at a solution, the use of speculation guaranteed not only a solution but also one that is nearly as fast as the fastest alternative. Our experimental results demonstrate that Anumita (a) improves the performance of these applications, (b) achieves significant speedup over statically chosen alternatives with modest overhead, and (c) is robust in the presence of performance variations or failure. Anumita's exper-

imental results indicate that it is possible to exploit coarse-grain speculative parallelism without sacrificing performance, portability, and usability.

1.2.2 Research Contributions

The primary contribution of this work is a **simple programming framework to exploit coarse-grain speculative parallelism in otherwise hard-to-parallelize applications**. To the best of our knowledge, this is the first work to introduce the notion of coarse-grain speculation in OpenMP. The following are specific contributions:

(i) A powerful programming model with expressive semantics to exploit coarse-grain speculative parallelism.

(ii) An execution model that provides the ability to selectively share and isolate state between execution contexts — a platform for composing speculations, and providing transparent name-space isolation.

(iii) Techniques to transparently and efficiently detect, isolate, and privatize memory updates.

(iv) Expressive evaluation criteria (temporal and qualitative) to evaluate speculations.

1.3 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 presents a pure runtime technique to eliminate deadlocks. Chapter 3 presents a program analysis and runtime technique to eliminate deadlocks and other concurrency bugs affecting lock based codes. Chapter 4 presents a framework to exploit coarse-grain speculative parallelism in hard-to-parallelize applications. Chapter 5 presents the conclusion of this dissertation. We survey and discuss the related literature for each of these contributions in the individual chapters.

Chapter 2

A Pure Runtime Approach

We present a language independent runtime system (Sammati[75], *agreement in Sanskrit*) that provides automatic deadlock detection and recovery for threaded applications that use the POSIX threads (Pthreads) interface — the de facto standard for UNIX systems. Sammati supports applications written using type-unsafe languages such as C, C++, and Fortran and compiled to the standard Executable and Linkable Format (ELF). Sammati is implemented as a pre-loadable library and it does not require either the application source code or recompiling/relinking phases, enabling its use for existing applications with arbitrary multi-threading models.

Sammati operates by associating memory updates with one or more locks guarding the updates and containing (privatizing) the updates until all locks protecting the updates have been released viz. the containment property. Intuitively, all memory updates within a critical section protected by one or more locks are performed atomically at the release of all surrounding locks. In such a system, deadlock detection can be performed at the acquisition of each lock and recovery merely involves selecting a victim lock and discarding all privatized memory updates performed subsequent to the acquisition of the victim. While the idea is conceptually simple, it requires *efficient* and *transparent* mechanisms to (a) identify updates within critical section, (b) isolate (contain) the memory updates, (c) preserve existing lock semantics, while still permitting containment based deadlock detection and recovery in the presence of nested locks, (d) perform inclusion of memory updates from the contained

critical section into the shared program address space, and (e) perform deadlock detection and recovery that deterministically eliminates deadlocks without either deadlocking itself or requiring an outside (external) agent.

The rest of the chapter is organized as follows. Section 2.1 describes the design and architectural aspects of Sammati. Section 2.2 presents a comprehensive description of the implementation details of the runtime system. Section 2.3 presents a detailed performance evaluation of Sammati's runtime system using the SPLASH [94], Phoenix [80], and synthetic benchmark suites. Section 2.4 discusses the limitations of a pure runtime approach. Section 2.5 discusses related work in the area of deadlock detection and recovery. Section 2.6 summarizes this chapter.

2.1 Design and Architecture

The core goal of Sammati is to *deterministically* and *transparently* detect and recover from deadlocks at runtime in threaded codes. In this section, we describe the design objectives and challenges involved in the design of Sammati.

2.1.1 Privatization and Containment

As discussed previously, to restore from a deadlock successfully, Sammati uses containment (through privatization) to ensure that memory updates (write set) within a critical section are not visible to any other thread until the successful release of the critical section. To implement containment within a critical section we need (a) a mechanism to identify memory updates and (b) a mechanism to privatize the updates. In the case of managed languages such as Java, program analysis can be used to detect the write set within a critical section, which can then be privatized through rewriting or source-to-source translation to implement containment. However, in the case of weakly typed languages such as C and C++, which allow arbitrary pointer access, program analysis cannot always determine the exact write set and conservatively degenerates to privatizing the entire address space, which is prohibitively expensive.

Alternately, a runtime can use page level protection to determine the write set within a critical section. In this approach, all data pages are write-protected on lock acquisition. If the subsequent critical section attempts to modify a page, it results in a segmentation violation signal. The signal handler then gets the page address, privatizes the page and changes the page permissions to read-write. While this solution works for processes that operate in distinct virtual address spaces, it does not work for threaded codes that share a single virtual address space and page protection bits. Consider the case where a thread acquires a lock L and updates two values in page P. Page P is write protected on the acquisition of lock L. To allow the update, the runtime would perform its privatization action and set the page permission for page P to read/write. Assume another concurrent thread of the same program now acquires lock M and updates a different data unit on page P. If the two lock acquisitions happen concurrently before the updates, the first thread that performs the update would change the page permissions of P to read/write. The second thread performing the update would never see a protection fault (since page P is already in read/write mode) and hence would not privatize its update, thereby breaking containment.

In the POSIX threads model, each thread has a distinct stack and threads of a process share their address space. In contrast, distinct processes are fully isolated from each other and execute in separate virtual address spaces. Neither of these models satisfies the isolation and selective state sharing requirements imposed by Sammati. Intuitively, what we need is an execution model that provides the ability to selectively isolate and share state between execution contexts.

Our key observation is that privatization can be implemented efficiently and transparently in a runtime environment if each thread has its own virtual address space. Modern UNIX operating systems already implement threads as lightweight processes with no major performance implications. We exploit this capability by creating multiple processes and share their global data regions through a common shared memory mapping. In essence, this creates a set of processes that are semantically equivalent to threads — they share their global data and have distinct stacks. To achieve containment for a page, we break its binding to the shared memory region and create a private page mapping (*mmap* with MAP_PRIVATE vs. *mmap* with MAP_SHARED) at the same

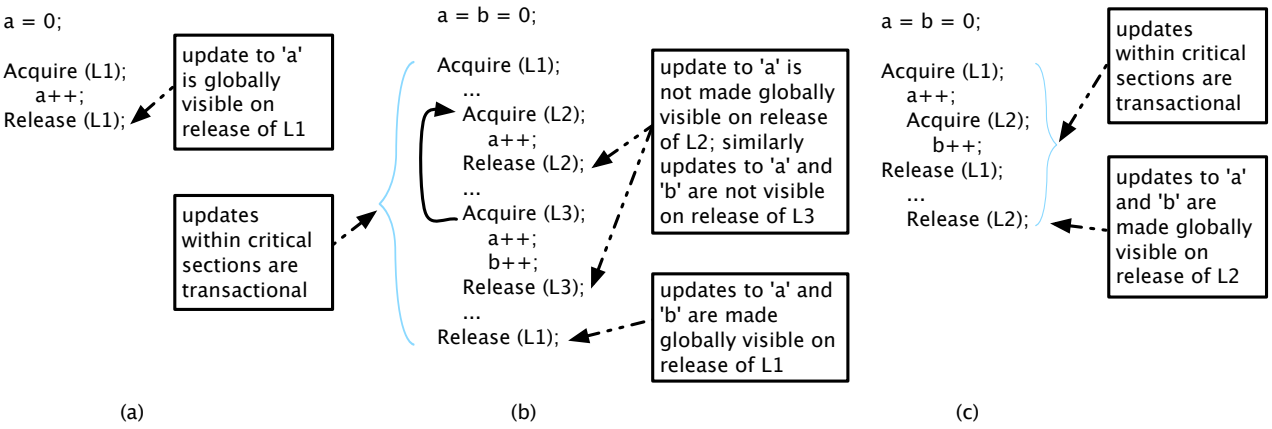


Figure 2.1: Visibility rules for memory updates within a lock context.

virtual address. Any updates to the private page are thus localized to the thread executing the critical section, thereby implementing containment. In the rest of this dissertation, we refer to these control flow constructs as *cords* to distinguish their properties from regular threads that operate within a single virtual address space.

2.1.2 Semantics for Propagating Updates

Propagating the privatized memory updates made within critical section(s) while ensuring program correctness presents several challenges. Consider the example shown in Figure 2.1(a). In the presence of a single lock around a critical section, Sammati propagates the updates on the release of the lock, e.g., L1 in Figure 2.1(a). However, nested locks are more complex. Consider the nested lock sequence shown in Figure 2.1(b). If the memory update to variable a within the critical section protected by lock L2 were made visible immediately after the release of L2 and subsequently a deadlock occurred on the acquisition of lock L3, where the victim was lock L1, there would be no way to unroll the side-effects of making the update to a visible. A secondary issue exists here in associating data with locks. When a unit of data is modified within a critical section protected by more than one lock, it is not possible to transparently determine the parent lock that is uniquely responsible for ensuring mutual exclusion on that data. For instance in Figure 2.1(c), it is not possible to transparently determine what data should be made visible. The variable a is protected

by lock L1; however, the variable b may be protected by L2 or L1.

To ensure containment of memory updates in the presence of nested locks, we employ *transactional* semantics. We employ the following two visibility rules for propagating memory updates.

1. *Memory updates are made visible on the release of all locks.*
2. *Track the release of nested locks in program order and defer performing the actual release till all locks around a critical section have been released in program order.*

Rule (1) is a necessary and sufficient condition for Sammati's deadlock elimination and recovery and rule (2) preserves the semantics of mutual exclusion. We illustrate the rationale behind these two rules using the example shown in Figure 2.1(b). If lock L2 is released at its original location, but its update to a is privatized (since there is another lock L1 around the same critical section), another thread may acquire lock L2, update a and release L2, creating a data race (*write-write* conflict) in otherwise correct code. Internally, Sammati tracks deferred releases within a nested lock sequence and elides the actual lock acquisition if a cord attempts to reacquire a deferred release lock in the same nested lock sequence. Our proposed semantics for propagating memory updates presents certain side-effects and could result in deadlocks [22, 23, 60]. We present a comprehensive solution to this problem in Chapter 3.

2.1.3 Deadlock Detection

A concurrent multi-threaded program can hold a set of locks and simultaneously be waiting on one or more locks. In such a context, deadlocks may arise due to multiple circular dependencies resulting in a multi-cycle deadlock. Eliminating multi-cycle deadlocks requires potentially multiple victims and the deadlock detection algorithm has to execute in a context that is guaranteed to not be a part of the deadlock itself. Additionally, multi-cycle deadlock detection algorithms are typically implemented as an external process that implements distributed deadlock detection and recovery. However, to transparently detect deadlocks we need an efficient mechanism that is capable

Algorithm 1 Deadlock Free Lock (lock L)

```
1:  Inputs: lock L
2:
3:  /* Global data structures */
4:  holding hash table ( lock ←key, pid ←value )
5:  waiting hash table ( pid ←key, lock ←value )
6:  lock_list := list of locks ordered by program order acquisition of locks.
7:
8:  /* Local data structures */
9:  lock S                                          /* globally shared lock across cords */
10: lock W                                          /* local lock identifier */
11:
12: R := Restore point containing the contents of stack frame and processor registers.
13: Set restore point R for lock (L) on rollback.
14: if (returning from a restore point flag is true) then
15:         Restore the stack.
16:         Free the old stack context and reset returning from restore point flag.
17: end if
18: id ⇐my pid
19: Acquire lock (S)                                /* enter runtime's critical section */
20: Try acquiring lock (L)
21: if (lock (L) is acquired successfully) then
22:         Insert in holding hash table (lock (L), id)
23:         Insert (lock (L), restore point (R)) at tail of lock_list
24:         Release lock (S)                        /* exit runtime's critical section */
25: else
26:         Insert id in waiting hash table (id, lock (L))
27:         W ⇐L
28:         Traverse:
29:          candidate ⇐find lock (W) in holding hash table
30:         if ( candidate == id ) then
31:                 recover_from_deadlock (W)       /* we have a deadlock !!! */
32:                 return  to restore point (W)
33:         else
34:                 W ⇐lock that candidate is waiting on
35:                 if ( lock (W) is valid ) then
36:                         goto Traverse /* continue traversing the waits for graph */
37:                 else
38:                         Release lock (S)        /* exit runtime's critical section */
39:                         Acquire lock (L)
40:                         Acquire lock (S)        /* enter runtime's critical section */
41:                         Delete ids entries from waiting hash table
42:                         if (lock(L) is acquired successfully) then
43:                                 Insert lock (L) in holding hash table (lock (L), id)
44:                                 Insert (lock (L), restore point (R)) at tail of lock_list
45:                                 Release lock (S)        /* exit runtime's critical section */
46:                         else
47:                                 Release lock (S)        /* error in acquiring the Lock (L) */
48:                                 Throw error and terminate program
49:                         end if
50:                 end if
51:         end if
52: end if
53: Update internal data structures and return
```

of deterministically eliminating deadlocks without either (a) deadlocking itself or (b) requiring an outside agent.

Since threads are transparently converted to cords in Sammati, the key observation here is that each cord (a single threaded process) may only wait (block) on a single resource (lock). Hence, a *waits-for* graph is sufficient to detect deadlocks. Such a property of lock based codes, significantly simplifies deadlock detection, since from the perspective of each cord, all deadlocks are single cycled deadlocks. Since all cords share a global address space, each thread can access the locking information pertaining to the remaining threads including the set of locks currently owned/held by a thread (*holding set*) and lock on which a thread is currently waiting (*waiting set*). Deadlock detection can hence be performed at lock acquisition a) without requiring an external agent and b) without having to worry about eliminating multi-cycle deadlocks that require multiple victims for deadlock resolution. Detection is also guaranteed not be a part of the deadlock itself.

Algorithm 1 shows the deadlock detection algorithm that executes at the acquisition of every lock. The detection algorithm uses three data structures — a holding hash table that associates locks being held with its owning cord, a waiting hash table that associates a cord with a single lock it is waiting on, and a per cord list of locks ordered (queue) by the program order of acquisition. The list of locks tracks nested locks and is freed at the release of all locks in a nested lock sequence. The deadlock detection algorithm implements a deadlock free lock acquisition and starts off by saving a restore point for deadlock recovery. The restore point contains the contents of the stack and all processor registers and is associated with the lock entry in the per cord list of locks. The algorithm then tries (non-blocking trylock) to acquire the requested lock L. If the acquisition succeeds, it inserts L into the holding hash table and the per cord lock list and returns. If the lock acquisition of L fails, the algorithm finds the cord C that owns L and checks if C is waiting on another lock M. If C is not waiting on any lock, there is no cycle and the deadlock detection algorithm inserts L into the waiting hash table and attempts to acquire L through a blocking lock acquisition. If C is waiting on another lock M, we find the cord that owns M and check to see if it is waiting on another lock and so on. Essentially, this algorithm implements a traversal of a waits-for-graph

Algorithm 2 Deadlock Recovery

```
1:  Input: lock W
2:  /* Global data structures */
3:  holding hash table ( lock ←key, pid ←value )
4:  waiting hash table ( pid ←key, lock ←value )
5:  lock_list := list of locks ordered by program order acquisition of locks.
6:
7:  for all entries starting from head of the lock_list find the first occurrence of lock (W) and do
8:      Discard all the modifications made within the locks from lock (W) to tail of lock_list
9:      Release all locks including lock (W)
10:     Clear relevant entries from holding hash table including entries for lock (W)
11:     Release lock(S)              /* exit runtime's critical section */
12: end for
13: return
```

to detect a cycle. A deadlock is detected if the traversal encounters an entry in the holding hash table with the cord id of the cord running the deadlock detection algorithm. The corresponding lock identifier in the holding hash table is chosen as the victim. Since each thread can wait on at most one lock, the depth of traversal of the deadlock detection algorithm is equal to the number of nodes in the waits-for graph. By representing the graph (holding and waiting sets) using hash tables, our deadlock detection algorithm has a time complexity upper bound of O(n), where n is the number of cords.

Note that in Line 39 of Algorithm 1 the blocking lock acquisition of the lock L is not protected by a secondary lock (doing so would result in serialization of all locks in the program) in this algorithm and hence the initial non blocking trylock may fail, and yet the holding hash table may not have an entry for the owning cord. This condition (an intentional benign race) cannot result in a deadlock. The intuition behind this reasoning is that while there may be multiple cords waiting on the same lock, the cord that acquires the lock successfully is no longer waiting on any lock and hence cannot be part of a cycle.

2.1.4 Deadlock Recovery

The deadlock detection algorithm presented above detects a deadlock and identifies a lock W as the victim for deadlock recovery. Recall that Sammati's runtime system saves a thread's execution context (*setjmp*) and its current stack frame prior to the thread's lock acquisition and maintains a

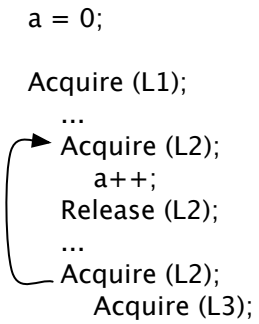


Figure 2.2: In this example, if the acquisition of L3 results in a deadlock and the victim was L2, Sammati's deadlock recovery rolls back to the earliest acquisition of L2.

list of locks acquired by a thread in *program order*. The deadlock recovery algorithm scans the list of locks to find the oldest acquisition of W, in program order and uses its associated recovery point (execution context and stack frame) from the lock list for recovery. To recover from the deadlock we (a) discard all memory updates performed within locks in the lock list including and after W (i.e. locks acquired later in program order after W), (b) release all locks in the lock list acquired after W and including W, (c) remove the locks released in step (b) from the holding hash table and finally restoring the stack and processor registers from the recovery point for W, which transfers control (*longjmp*) back to deadlock free lock acquisition of the victim lock W.

Note that deadlock recovery uses the recovery point from the oldest (in program order) acquisition of lock W. The reasoning behind this is subtle. Consider the example shown in Figure 2.2. A cord C acquires a lock L1, followed by lock L2 and updates a variable *a*. It then releases lock L2, reacquires L2 and acquires another lock L3. The acquisition of L3 results in a deadlock and the deadlock recovery algorithm selects L2 as the victim for rollback. However, if we rolled back to the most recent acquisition of L2 and released L2, thereby breaking the deadlock, the earlier update to variable *a* within L2 would still be privatized and not visible externally. A cord M waiting on L2 can now acquire L2 and change the value of variable *a*, creating an illegal write-write conflict with the privatized copy within cord C.

Deadlock Aware Memory Management

Recovery involves unrolling memory allocations, Sammati is capable of performing garbage collection while recovering from deadlocks. Memory management operations can be classified into allocation and deallocation operations. To handle allocation operations, we maintain a list of allocation operations that occur within a nested lock scope. This list is used to garbage collect the allocations if a lock in the nested lock scope is chosen as the victim for deadlock recovery. Deallocation operations will be buffered (delayed deallocation) until the release of all locks in a nested lock scope. To preserve transparency Sammati overloads the standard POSIX memory management primitives including *malloc*, *calloc*, *realloc*, *valloc*, and *free* and the POSIX thread (Pthread) library. Sammati performs *deadlock aware* memory management while recovering from a deadlock. It tracks all memory allocations made within a critical section. On recovery, Sammati internally frees all such allocations to prevent memory leaks.

2.2 Implementation

We implemented Sammati's runtime as a shared library that is pre-loaded by the dynamic linker (ld)'s LD_PRELOAD environment variable before executing the binary. Sammati implements most of the POSIX threads interface, including thread creation, destruction, mutual exclusion locks, barriers, and condition variables. In this section we describe the key elements and implementation details of Sammati's runtime system.

2.2.1 Shared Address Space

A multi-threaded process has a shared address space, with a distinct stack and a distinct thread-local storage (TLS) region for each thread. To provide efficient address space isolation and containment of memory updates (described in Section 2.1.1) Sammati creates an illusion of a shared address space among processes. Sammati overloads the POSIX thread create (*pthread_create*) call

to create a cord.

Global Data

The constructor in Sammati's runtime system traverses the link map of the application ELF binary at runtime and identifies the zero initialized and un-initialized data in the *.bss* section and the non-zero initialized data in the *.data* section. Sammati then unmaps these sections from the loaded binary, maps them from a SYSV memory mapped shared memory file, and reinitializes the sections to the original values. This mapping to a shared memory file is done by the *main* process before its execution begins at *main*. Since cords are implemented as processes that are forked at thread creation (we actually use the clone() system call in Linux to ensure that file mappings are shared as well), a copy of the address space of the parent is created for each cord and consequently the cords inherit the shared global data mapping. Any modifications made by any cord to global data is immediately visible to all cords.

Heap

In a multithread process the heap is also shared among all threads of a process. To implement this abstraction, we modified Doug Lea's *dlmalloc* [27] allocator to operate over shared memory mappings. This memory allocator internally allocates 16 MB chunks (the allocator's internal granularity), which are then used to satisfy individual memory requests. Each 16MB chunk is backed by a shared memory file mapping and is visible to all cords. Sammati provides global heap allocation by sharing memory management metadata among cords using the same shared memory backing mechanism used for *.data* and *.bss* sections. Similar to the semantics of memory allocation for threads, any cord can allocate memory that is visible and usable by any other cord. When a cord first allocates memory, the memory addresses are allocated in its virtual address space and backed by a shared memory file. If any other cord accesses this memory, it results in a segmentation violation (a map error) since the address does not exist in its address space. Sammati's runtime handles this segmentation violation by consulting the memory management metadata to

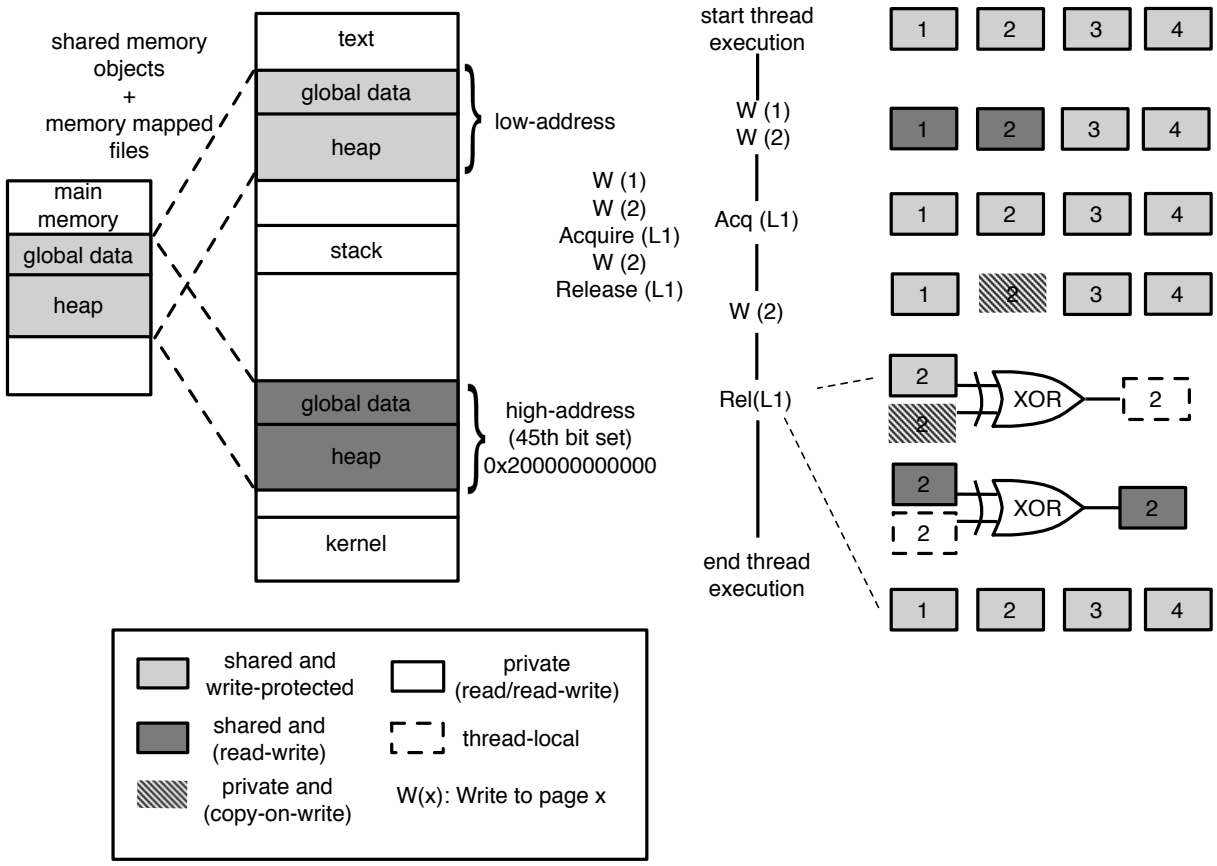


Figure 2.3: (left) Illustrates the virtual memory address (VMA) layout of each process (cord). Sammati provides memory isolation by transforming threads to processes and uses shared memory objects and memory mapped files to share global variables and the process heap among cords. (right) Illustrates how the VMA is manipulated by Sammati with a simple example explained in text.

check if the reference is to a valid memory address allocated by a different cord. If so, it maps the shared memory file associated with the memory thereby making it available. Note that such an access fault only occurs on the first access to a memory region allocated by a different cord, and is conceptually similar to lazy memory allocation within an operating system. To further minimize such faults, we map the entire 16MB chunk that surrounds the faulting memory address. Sammati exposes dynamic memory management through the standard POSIX memory management primitives including *malloc, calloc, realloc, valloc* and *free.*

Stack

Since stacks are local to each thread, Sammati does not share stacks among cords. Each cord has a default stack of 8MB similar to threads. The stack is created at cord creation and it is freed automatically when a cord terminates.

Shared View of Runtime System

In UNIX process semantics, each process has its own copy of the data segment of the shared libraries. Consequently, Sammati's runtime is not shared among cords by default. To circumvent this issue and to maintain a shared view of the runtime, each newly created cord automatically executes an initialization routine that maps the shared state of Sammati's runtime prior to executing its thread start function. Figure 2.3 illustrates the virtual address space layout of a cord.

2.2.2 Detecting Memory Updates Within Critical Sections

To successfully rollback on deadlock, the runtime system must precisely identify memory updates performed within critical sections. We define two contexts of execution for a cord; a cord is said to be in a *lock context* if it acquires a lock and it remains in the lock context until it releases the lock. In case of nested locks, a thread remains in a lock context until all the locks it acquired previously are released.

Address Space Protection

Sammati's runtime employs address space protection to write-protect (PROT_READ) a cord's virtual memory address (VMA) pages of the shared address space (global data and heap). If a cord attempts to modify (write to) the shared data, the runtime system handles the access fault (SEGV_ACCERR) and makes a note of the page and the current context of execution i.e., the current critical section. The runtime maintains a unique list of pages that were modified within

each critical section. Read accesses to the shared data do not produce access faults and execute as they would otherwise. The permissions of the page are then set to read-write so that the cord can continue its execution. Sammati includes two variants of address space protection.

1. *Variant-1:* On entry of a critical section, the runtime system write protects all the pages of the shared address space (global data and heap). By leveraging the address space protection semantics described above, the runtime system precisely identifies the updates within a lock context. On exit of a critical section, the runtime system restores the permissions of all the pages of shared address space by unprotecting them.

2. *Variant-2:* The runtime system at the start of the program execution, write-protects the entire shared VMA. Hence, any updates to memory outside of the critical section are tracked through access faults. On a lock acquisition, only the set of pages that were modified prior to acquiring a lock are write-protected, instead of protecting the entire shared VMA. In essence, this approach tracks the write-set of a cord between lock release and lock acquisition (ordinary memory accesses) and only write-protects this write-set.

The efficacy of each variant depends on (a) total size of the memory footprint, (b) the total number of locks acquired, and (c) the number of updates performed within a critical section. The choice between the variants is dependent on these characteristics of the applications.

2.2.3 Isolating Memory Updates

In Variant-1 since all pages in the shared VMA are write protected, when a cord modifies a shared VMA page from within a *lock context*, it is detected by the occurrence of a segmentation violation (access error). Sammati's runtime handles the access violation and isolates the updates by remapping the faulting page from the shared memory (MAP_SHARED) backing to private (MAP_PRIVATE) mode. In the private mode, updates to the page from the cord in the lock context are no longer visible to other cords, effectively privatizing the page. Sammati's runtime then creates a copy of the page (called a twin page), changes the page permission to read/write and

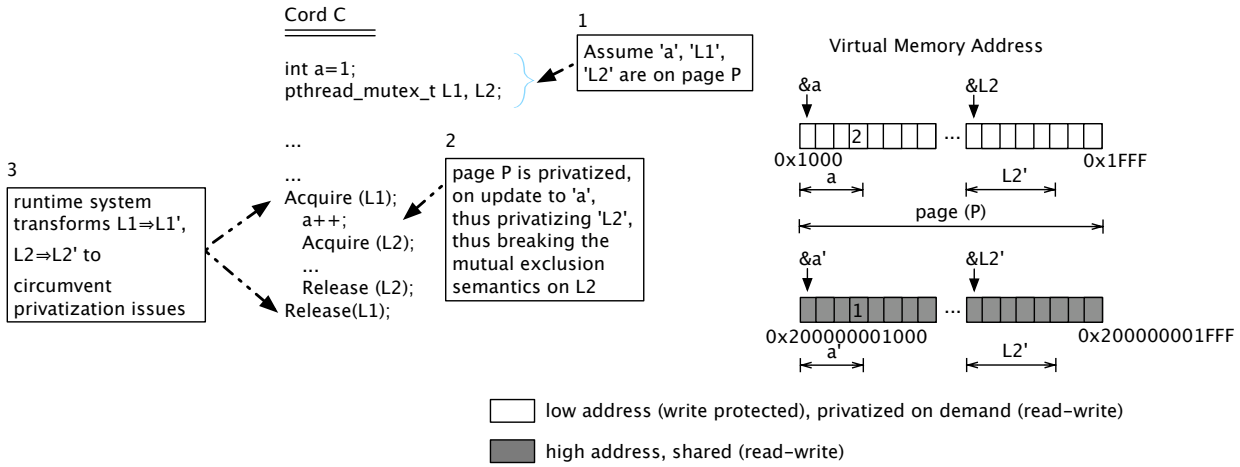


Figure 2.4: Subtle issues with privatization.

returns from the segmentation violation handler, allowing the cord to continue its execution. The twin page is used to detect the actual memory updates on the page, which are then committed when the cord exits a lock context. We note that the space overhead of this approach is $O(W)$, where W is the write set (in pages) within a lock context.

Lazy Privatization

Intuitively, Sammati implements lazy privatization of memory updates that defers privatization to the first instant the memory update occurs. Conservatively privatizing the entire address space at the acquisition of the first lock in a nested lock sequence (this was in fact our original solution), resulted in a far too high runtime cost for applications with fine-grain locks. Since standard lock semantics of mutual exclusion locks do not require such semantics, we chose to implement the more efficient, lazy privatization approach.

2.2.4 Preserving Synchronization Semantics

Sammati's runtime preserves the synchronization semantics of multi-threaded codes among cords (recall, implemented as processes) by transforming all mutex exclusion locks (*pthread_mutex_t*), barriers (*pthread_barrier_t*), and condition variables (*pthread_cond_t*) within the program to process-

shared (PTHREAD_PROCESS_SHARED) locks, which enables their use among cords.

Recall that in the case of nested locks, a cord remains in lock context until all the locks it acquired previously are released. On unlock, Sammati marks a lock for release but defers the actual release of the lock until all locks in the nested lock sequence have been released in program order (discussed in Section 2.1.2).

A subtle side effect of memory isolation through privatization (discussed in Section 2.2.3) occurs because synchronization primitives such as locks, barriers and condition variables in a multi-threaded program are generally global, i.e., they reside in the shared VMA irrespective of how they are initialized (statically declared globals or dynamically allocated on the heap). For instance, consider the example illustrated in Figure 2.4 involving a mutex lock. If a cord C acquires a lock and subsequently modifies a page P, P is privatized. If P contains any definitions of lock variables (which may happen if P contains parts of the .data section), they end up being privatized as well. If such a privatized lock is subsequently used in a nested lock sequence by cord C, it no longer provides mutual exclusion outside cord C since any updates to the lock (such as acquisition/release) are privatized and not visible outside C. A simple solution to this problem would have been to modify the application source code to allocate memory for all mutual exclusion locks from a distinct shared memory zone that is not subject to privatization. However, this requires source code modifications and conflicts with our goal of being a transparent runtime solution to deadlock recovery.

To address this side effect of address space privatization, we present a novel approach that leverages the large virtual memory address (VMA) provided by 64-bit operating systems. Linux allows 48 bits of addressable virtual memory on x86-64 architectures and we exploit this aspect of large addressable VMA available to a process. Recall that our runtime system maps globals and the heap (described in Section 2.2.1) using shared memory objects and memory mapped files. Using the same shared memory objects and memory mapped file, Sammati creates an identical secondary mapping of the global data sections and heap at a *high address* (45th bit set) in the VMA of each cord. The application is unaware of this mapping, and performs its accesses (reads/writes) at the original low address space. In effect, the high address mapping creates a *shadow* address space for

all shared program data, and modifications (unless privatized) are visible in both address spaces (shown in Figure 2.3). The high address space shadow is always shared among cords and it is never privatized.

To perform synchronization operations (e.g., mutual exclusion such as in Figure 2.4), Sammati at runtime transforms the address of a mutual exclusion lock by setting the high address bit and performs the lock operation in the shadow address space. Since the shadow address space is not subject to privatization, lock acquisitions and releases are visible across all cords, correctly implementing mutual exclusion.

2.2.5 Committing Memory Updates

When a cord exits a lock context, any of its updates contained in its privatized data must be made visible and reconciled with other cords. In order to perform this *inclusion*, we need to identify the exact write set of the lock context. Hence, for every page modified within a lock context, we compute an XOR difference (byte wise XOR) between the privatized version of the page and its copy (twin) that was saved before any modifications were made within the lock context. The XOR difference identifies the exact bytes that were changed. Distributed shared memory (DSM) systems ([4, 17, 46]) employ the twin/diff technique for identifying false sharing. Inspired by such systems we leverage this technique to efficiently track the updates.

To perform inclusion, we apply the XOR difference to the high address shadow region of the VMA (shown in Figure 2.3) by computing the XOR of the difference and the high memory page, which makes the updates visible to all cords. Sammati then reverts the privatization by discarding the privatized pages and remapping their shared versions.

We note that since the semantics of mutual exclusion locks prevent two cords from modifying the same data under different locks, updates from concurrent lock releases would be to different regions within a memory page. Hence, the operation of applying XOR differences to the shadow address space is a commutative operation and is thus implemented as a concurrent operation. If the original

program contains data races, Sammati preserves the race. We explain how Sammati is capable of detecting write-write races in Section 2.2.7.

We present a simple example to illustrate how Sammati manipulates the VMA of each cord while providing isolation, privatization and inclusion. Consider the scenario as shown in Figure 2.3 where a thread has four shared pages when it starts its execution. Initially, all the shared pages (1, 2, 3, 4) are write-protected. When a thread attempts to write to pages outside a lock context, the pages (1, 2) are then given write access. On entering a lock context, only pages that were modified previously are write-protected (pages 1, 2). If a thread attempts to write to a page (2) within a lock context, the page is privatized (lazy privatization) and a copy of it is created (twin). Before exiting a lock context, the runtime system evaluates the modifications by computing an XOR difference of the private page against its twin. It then checks for any write-write races before applying the difference to the page in the shared high-address space.

2.2.6 Condition Variables and Semaphores

POSIX condition variables provide a synchronization primitive that enables a thread to atomically release a lock and wait on a condition to be signaled. More than one thread is permitted to wait on the same condition signal. On receiving a condition signal, one of the waiting threads atomically reacquires the lock and resumes execution.

To implement POSIX condition signaling primitives (wait, signal, and broadcast) and semaphores, Sammati treats them as a commit point, thus making all updates within the critical section visible prior to invoking the underlying POSIX signaling primitive. On resuming execution from a condition wait, Sammati treats the resumption as a new lock acquisition. It marks the lock as non-recoverable since an irrevocable action has been performed within a critical section, and does not attempt to privatize the updates anymore. Sammati offers limited support for recovery in the presence of condition wait/signal primitives protected by more than one lock. Recall to preserve the semantics of condition variables, Sammati propagates the updates (in program order) performed prior to the wait/signal operation. Such semantics conflict with the privatization mech-

Table 2.1: SPLASH Benchmarks.

Application	Description
Barnes	Barnes-Hut algorithm to simulate interaction of a system of bodies (N-body problem)
FMM	Fast Multipole Method to simulate interaction of a system of bodies (N-body problem)
Ocean-CP	Simulates large-scale ocean movements based on eddy and boundary currents
Water-nsquared	Simulates forces and potential energy of water molecules in the liquid state
FFT	Computes one-dimensional Fast Fourier Transformations
LU-NCP	Factors (2D array) a dense matrix into the product of a lower and upper triangular matrices
LU-CP	Factors (array of blocks) a dense matrix into the product of a lower and upper triangular matrices
Radix	Performs integer radix sort

anism (discussed in Section 2.1.1) required to facilitate recovery. Consequently while Sammati is capable of detecting deadlocks in the presence of condition variables in nested locks, it is incapable of recovering from deadlocks if they involve rolling back asynchronous events.

2.2.7 Detecting Write-Write Races

Sammati can detect and report *write-write* races that occur between (a) guarded and concurrent unguarded updates to a shared value and (b) improperly guarded updates, where a single data value is guarded by two or more different locks. Sammati identifies these data races while committing the updates to memory (Section 2.2.5) by checking every word in the *diff* page. If the word is non-zero, then it indicates that the cord has modified data within a lock context. Sammati then compares the word corresponding to the non-zero value in its *twin* with its equivalent in the shadow address space that is shared across all cords. In essence this comparison checks to see if some other cord executing concurrently has modified a word that should have been uniquely protected by the current lock context. If the two values are not equal then this indicates that the same word was modified within the current lock context as well as by one or more cords.

We note Sammati does not detect data races that might potentially happen; instead it precisely identifies data races that did happen during the execution. Additionally, since Sammati detects the race at inclusion, it does not have enough information to identify all the cords involved and/or who caused the conflict.

Table 2.2: Phoenix Benchmarks.

Application	Description
Histogram	Generates a histogram of frequencies of pixels values (R,G,B) in an image
Kmeans	Iteratively performs data clustering of N-dimensional data points
Linear Regression	Performs linear approximation of 2D points
Matrix Multiply	Computes the product of two matrices
PCA	Performs principal component analysis on a matrix

2.3 Experimental Evaluation

We evaluated Sammati's runtime performance using two POSIX threaded benchmark suites (SPLASH [94], Phoenix [80]), and several synthetic benchmark suites. The SPLASH suite (described in Table 2.1) contains applications from several domains including high-performance computing, signal processing, and graphics. We chose to report the results from benchmarks that have (a) a runtime of at least a few seconds to avoid statistical noise from the scheduler and (b) applications that compile and run on a 64-bit machine. The Phoenix suite (described in Table 2.2) contains applications from enterprise computing, artificial intelligence, image processing, and scientific computing domains. The synthetic benchmarks contain programs written to create deadlocks both deterministically and randomly, and finally, examples of deadlocks in the literature [9, 43, 75].

We chose the SPLASH and PHOENIX benchmarks for several reasons. First, their performance has been well studied in the literature. Second, SPLASH was originally intended as a shared memory system benchmark suite akin to SPEC and includes a variety of applications with different memory models and locking regimes. Third and most importantly, we note that the lock acquisition rates of some of the applications is extremely high, e.g., Barnes (640K locks/sec), FMM (265K locks/sec), and Water (70K locks/sec). We define the lock acquisition rate as the ratio of total locks acquired by a program to its runtime (runtime of vanilla application). Such extremely high lock rates stress test Sammati to the extreme since the lock rate is one of the primary determinants of its overhead. In contrast, enterprise applications such as databases are largely bound by disk IOP rates of a few hundred to a few thousand per second, which we believe would lead to lower lock acquisition rates.

Table 2.3: Classification of SPLASH and PHOENIX benchmark suites based on lock acquisition rate.

Category	Lock Acquisition Rate	Benchmarks
Set 1	50K/sec - 700K/sec	Barnes, FMM, and Water-nsquared
Set 2	10/sec - 50K/sec	Ocean, Radix, and PCA
Set 3	0.5/sec - 10/sec	FFT, LU-contiguous partition, and LU-non contiguous partition
Set 4	0/sec - 0.5/sec	Histrogram, Kmeans, Linear Regression, and Matrix Multiply

2.3.1 Experimental Setup

We performed all our experiments on a 16 core shared memory machine (NUMA) running Linux 2.6.32 with 64GB of RAM. The test system contains four 2 GHz Quad-Core AMD Opteron processors. We ran each application under four scenarios. First, we evaluated Sammati's pure runtime approaches, i.e., Variant-1 (tracks lock context accesses) and Variant-2 (tracks ordinary accesses), where we pre-load our runtime system using LD_PRELOAD. We also ran the vanilla Pthread application. For each scenario, we ran a benchmark 5 times and we present the average of the 5 independent runs. We measured the total runtime (wallclock time) for each application using the UNIX *time* command.

2.3.2 Performance Analysis

We classify the 13 benchmarks from SPLASH and PHOENIX into 4 sets based on the lock acquisition rate of each application. We present a summary of this classification in Table 2.3. For each benchmark we measured several important characteristics including the number of locks, the lock context write-set, pages write-protected, pages restored (un-write-protected), pages privatized and shared, etc., to understand the performance implications of using Sammati.

Runtime Overhead

Recall that we presented two approaches (a.k.a variants, described in Section 2.2.2) to detect memory updates performed within critical sections. In Variant-1, Sammati tracks the *lock context accesses*. To accomplish this objective, Sammati write-protects the entire shared address space on

a lock acquisition and tracks access faults to determine the pages modified within a lock context. Sammati then creates a copy of the page, thereby privatizing the page to ensure containment of memory updates. On the release of a lock, Sammati computes the XOR difference to shadow memory (high address) to propagate the memory updates, maps the page back to shared state, and finally un-write-protects the entire shared address space.

$$Overhead_{Sammati(Variant-1)} \propto (\textit{number of locks} \times [\textit{cost of address space protection} + \textit{cost of address space un\_protection}] + \textit{write\_set} \times [\textit{cost of observing a write access} + \textit{cost of privatization} + \textit{cost of creating copy} + \textit{cost of propagating updates} + \textit{cost of un\_privatizing}])$$

In Variant-2 Sammati tracks *ordinary accesses*. Sammati's runtime begins program execution by write-protecting the entire shared address space. Any consequent newly allocated memory is also write-protected enabling Sammati to track the pages modified in ordinary regions through access faults. On a lock acquisition Sammati write protects the pages modified in an ordinary region. Similar to Variant-1, Sammati tracks the access faults to determine the pages modified in a lock context and creates a copy of the page, privatizing the page to ensure containment of memory updates. On the release of a lock, Sammati computes the XOR difference to shadow memory (high address) to propagate the memory updates and restores the page back to being shared and write-protected.

$$Overhead_{Sammati(Variant-2)} \propto (\textit{cost of observing ordinary write accesses} + \textit{number of locks} \times [\textit{cost of protecting ordinary accesses}] + \textit{write\_set} \times [\textit{cost of observing a write access} + \textit{cost of privatization} + \textit{cost of creating copy} + \textit{cost of propagating updates} + \textit{cost of un\_privatizing}])$$

Set #1

Figure 2.3.2 illustrates the performance of Sammati and Pthreads for applications in Set-1. As shown in Table 2.4, applications such as Barnes, FMM and Water acquire a reasonably large

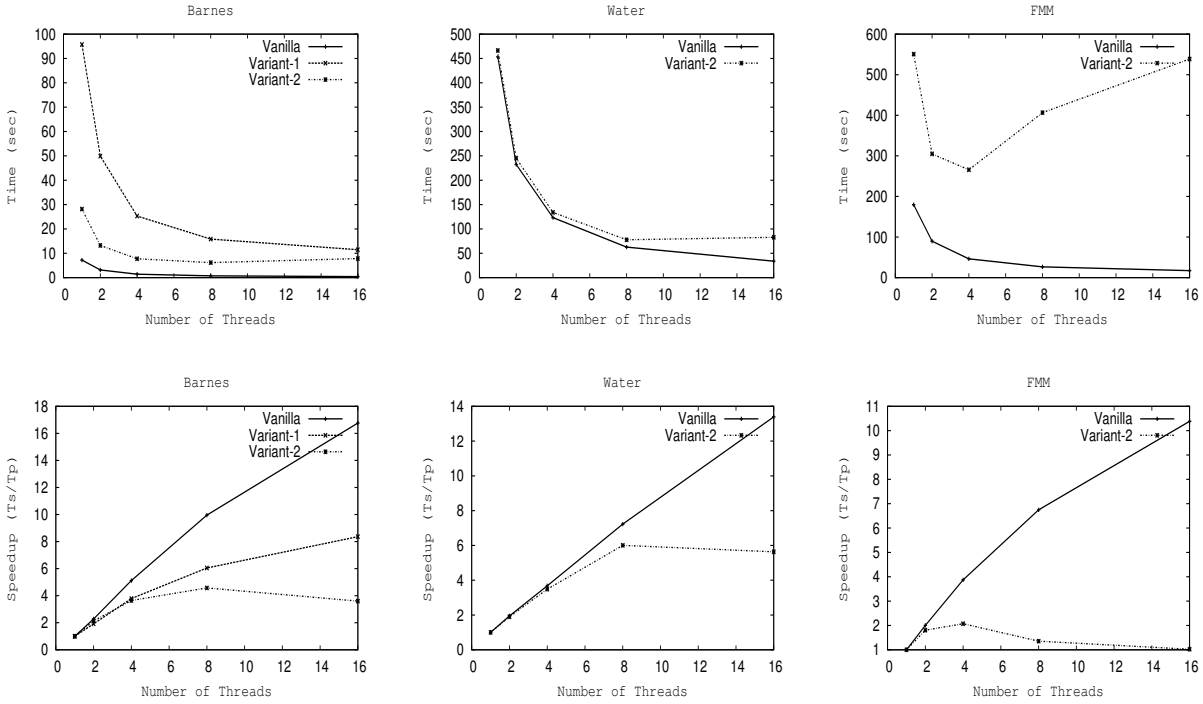


Figure 2.5: Performance of applications from Set-1 (extremely high lock acquisition rate, typically 50K/sec - 700K/sec) with Sammati on a 16 core system.

Table 2.4: Characteristics of applications in Set-1.

Benchmark	Threads	Total Locks	Lock Rate (locks/sec)	Pages Modified in Lock Context			
				Total	Min	Max	Avg
Barnes	1	275216	38182.02	843703	1	20	3
	2	275222	87095.57	843520	0	39	6
	4	275256	195494.32	843822	0	77	12
	8	275356	380325.97	843751	0	155	24
	16	275494	640683.72	843778	0	308	48
FMM	1	4517437	25156.13	4728864	0	3	1
	2	4521269	50409.96	4736848	0	6	2
	4	4544061	98004.163	4782361	0	12	4
	8	4560144	171382.44	4814982	0	25	8
	16	4588580	265297.18	4871931	0	49	16
Water-nsquared	1	266568	588.20	270728	1	2	1
	2	533071	2292.68	541391	1	4	2
	4	799902	6489.52	812382	1	8	4
	8	1333564	21266.80	1354364	1	16	8
	16	2400888	70960.81	2438328	1	32	16

Table 2.5: Profile of applications in Set-1 with Sammati (Variant-1).

Benchmark	Threads	Total Pages			
		Write-protected	Access faults	Map shared	Un-write-protected
Barnes	1	15155044256	843703	843703	15155044256
	2	15156475540	843520	843520	15156475540
	4	15160549968	843822	843822	15160549968
	8	15148985696	843751	843751	15148985696
	16	15231512272	843778	843778	15231512272

Table 2.6: Profile of applications in Set-1 with Sammati (Variant-2).

Benchmark	Threads	Total Pages			
		Write-protected	Access faults	Map shared	Un-write-protected
Barnes	1	277866	1121296	843703	277593
	2	286433	1129787	843625	286162
	4	298746	1142375	843898	298477
	8	317262	1160509	843512	316997
	16	353403	1196857	843711	353146
FMM	1	33352821	38138865	4728864	33410001
	2	34315248	39161456	4736848	34424608
	4	35040077	40000100	4782361	35217739
	8	35742953	40854289	4814982	36039307
	16	36848173	42243104	4871931	37371173
Water	1	906035	1176764	270728	906036
	2	1721148	2262540	541391	1721149
	4	3351374	4163757	812382	3351375
	8	6611826	7966191	1354364	6611827
	16	13132730	15571059	2438328	13132731

number of locks ($\approx 10^6$) and they have extremely high lock acquisition rates. For instance, Barnes acquires 640K locks/sec, FMM acquires 265K locks/sec, and Water acquires 70K/sec for 16 threads. Furthermore, all the three applications modify a significant amount of data (pages) within a lock context. Barnes modifies 8.4×10^5 pages, FMM modifies 4.8×10^6 pages and Water modifies 2.4×10^6 pages. Consequently, both the variants of Sammati incur a significant runtime overhead and result in poor speedup.

Since in Variant-1, Sammati write-protects the entire shared address space on every lock acquisition to detect memory updates within a critical section, and unprotects the address space on lock release, Sammati incurs a significant runtime overhead due to the address space protection. Barnes write-protects over 3×10^{10} pages. The high costs of address space protection prevent FMM and Water from making any meaningful progress, resulting in poor CPU utilization. We set a cutoff limit on the runtime of 20 min, hence we omit the results of Variant-1 for FMM and Water in Table 2.5.

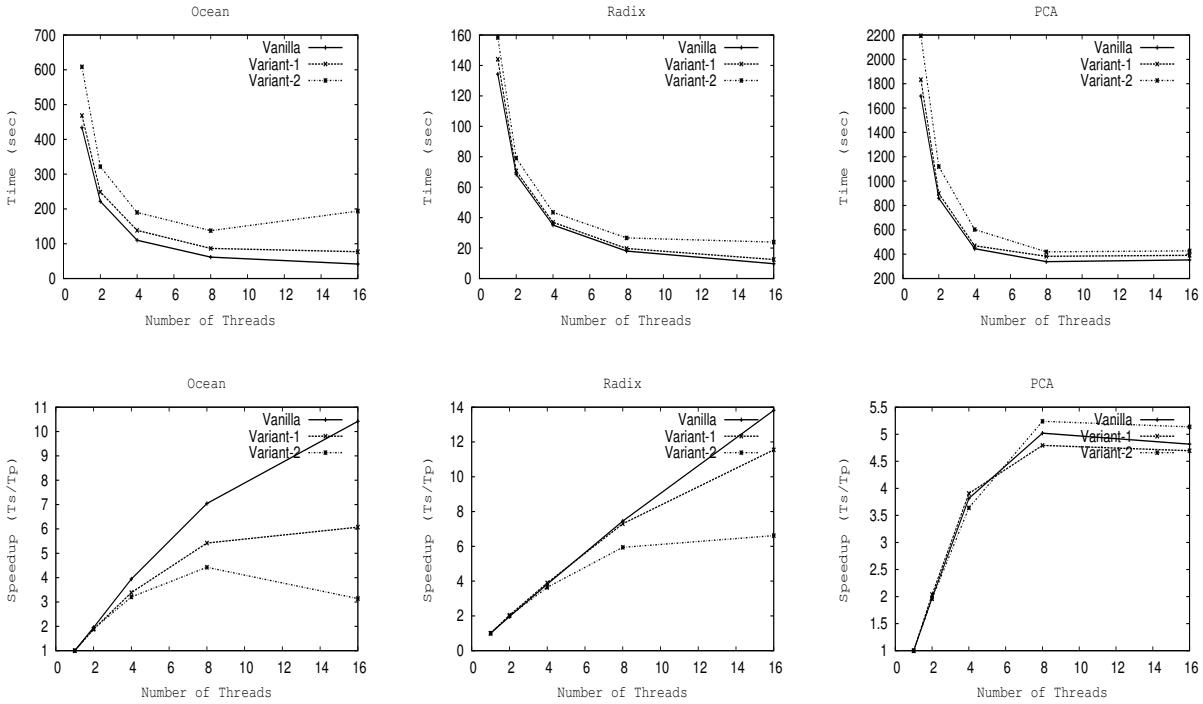


Figure 2.6: Performance of applications from Set-2 (moderate lock acquisition rate, typically 10 - 50K/sec) with Sammati on a 16 core system.

Sammati write-protects and un-write-protects an order of magnitude (10^4) fewer pages under Variant-2. Hence, Variant-2 (shown in Figure 2.3.2) incurs significantly lesser overhead compared to Variant-1. Recall that in Variant-2 all shared data is maintained in read-only form. When updates to shared data occur outside a lock context, we store the page address in a write-set list and change the page permissions to read/write. At the acquisition of the next lock, we only change the page permissions of pages in the write-set list to read only, thereby avoiding the cost of write protecting the entire data space. Variant-2 is biased towards fine-grain locking where the lock context writes are small (shown in Table 2.6), which is true for Barnes, FMM and Water.

The write-set in a lock context is invariant in both the variants of Sammati since the number of pages modified within a lock context is identical for a given problem. The total number of pages modified in a lock context (shown in Table 2.4) is identical to the number of pages map shared (shown in Tables 2.5 and 2.6). This indicates that the overhead due to the write-set in

Table 2.7: Characteristics of applications in Set-2.

Benchmark	Threads	Total Locks	Lock Rate (locks/sec)	Pages Modified in Lock Context			
				Total	Min	Max	Avg
Ocean	1	173	0.40	173	1	1	1
	2	346	1.56	268	0	2	0.78
	4	692	6.29	422	0	4	0.61
	8	1384	22.47	331	0	8	0.24
	16	2768	66.47	496	0	16	0.18
Radix	1	4	0.03	4	0	1	1
	2	17	0.25	15	0	2	1
	4	43	1.22	28	0	4	0.65
	8	95	5.27	66	0	8	0.7
	16	199	20.47	133	0	16	0.67
PCA	1	10001	5.89	10001	1	1	1
	2	10002	11.63	10002	1	2	2
	4	10004	22.47	10004	1	4	4
	8	10008	29.58	10008	1	8	8
	16	10016	28.40	10016	1	16	16

both Sammati variants is identical. The choice between variants is determined by the number of ordinary accesses and lock context accesses. When the write-set between lock contexts is large, the cost of handling the access error and changing the page permissions outside lock contexts, as in Variant-2, incurs more overhead compared to Variant-1. The results from Set-1 show that Sammati is capable of performing deadlock detection and recovery even when subject to extreme conditions.

Set #2

Figure 2.3.2 illustrates the performance of Sammati and Pthreads for applications in Set-2. Ocean acquires approximately $2.7K$ locks, and Radix acquires 200 locks and performs 159 condition signals (shown in Table 2.10), and PCA acquires $10K$ locks. The lock acquisition rate (shown in Table 2.7) is significantly lower compared to applications in Set-1. Consequently, Sammati's performs relatively well with modest overhead for most applications in Set-2 compared to applications in Set-1

Ocean has a reasonably large memory footprint ($\approx 14GB$) compared to the remaining benchmarks in Set-2, consequently, even though it acquires fewer locks and modifies very little data ($\approx$ 500 pages for 16 threads) under lock context, it incurs a noticeable reduction in speedup due to the cost of address space protection and un-protection. Ocean accesses a significant amount of data

Table 2.8: Profile of applications in Set-2 with Sammati (Variant-1).

Benchmark	Threads	Total Pages			
		Write-protected	Access faults	Map shared	Un-write-protected
Ocean	1	628473535	173	173	628473535
	2	1257285458	268	268	1257285458
	4	2515604764	422	422	2515604764
	8	5034055032	331	331	5034055032
	16	10076928912	496	496	10076928912
Radix	1	4210964	4	4	7369187
	2	17896597	15	15	29476748
	4	45267863	28	28	83166539
	8	100010395	66	66	177913229
	16	209495459	133	133	376881278
PCA	1	1966286609	10001	10001	1966286609
	2	1966483218	10002	10002	1966483218
	4	1966876436	10004	10004	1966876436
	8	1967662872	10008	10008	1967662872
	16	1969235744	10016	10016	1969235744

Table 2.9: Profile of applications in Set-2 with Sammati (Variant-2).

Benchmark	Threads	Total Pages			
		Write-protected	Access faults	Map shared	Un-write-protected
Ocean	1	34004498	35053769	173	35053596
	2	34012951	35062312	255	35062057
	4	34611493	35661236	365	35660871
	8	34629415	35679417	592	35678825
	16	35326938	36377572	648	36376924
Radix	1	1572987	2097221	4	2097217
	2	1575085	2099823	13	2099810
	4	1577260	2102520	31	2102489
	8	1581628	2107920	59	2107861
	16	1590355	2118721	122	2118599
PCA	1	50053311	50063312	10001	50053311
	2	50053311	50063313	10002	50053311
	4	50053311	50063315	10004	50053311
	8	50053311	50063319	10004	50053311
	16	50053311	50063327	10016	50053311

Table 2.10: Profile of POSIX condition variables (signals/waits) in Radix.

Threads	Condition Variables
1	3
2	11
4	36
8	74
16	159

($\approx 3.7 \times 10^7$ pages) outside the lock context and only a few hundred pages within a lock context. Since Variant-2 tracks (page granularity) ordinary accesses, it incurs more overhead compared to Variant-1.

Sammati's Variant-1 on average write-protects and un-write-protects approximately twice the number of pages with increasing threads in Ocean, for example, 1×10^{10} with 8 threads and 2×10^{10} pages for 16 threads. Hence, we find Variant-1 incurs a performance overhead compared to native pthread execution as we increase the number of threads. This runtime cost of address space protection precludes Variant-1 from scaling with increasing number of threads and consequently results in a reduction in speedup.

Radix uses condition signals (shown in Table 2.10). Recall that Sammati propagates the lock context memory updates prior (in program order) to performing the actual condition signal/wait operation. The runtime performance of Variant-1 for Radix is comparable to native pthread execution. Radix performs several ($\approx 10^4$) orders of magnitude more ordinary accesses (writes) over lock context writes, consecutively, Variant-2 incurs more overhead than Variant-1.

The performance of Variant-1 for PCA is identical to native pthread execution and the performance of Variant-2 is comparable to native thread execution and Variant-1. Similar to Radix, PCA has a relatively low write-set (10016 pages) and acquires fewer locks with low lock acquisition rate. The number of ordinary accesses ($\approx 5 \times 10^7$), lock context access (10016 pages), and the number of pages write-protected ($\approx 1.9 \times 10^9$ pages) remain constant with increasing number of threads as shown in Tables 2.8 and 2.9. Consequently, both variants of Sammati scale well.

Set #3

The performance of applications in Set-3 as shown in Figure 2.3.2 is comparable to native pthread execution. LU-CP and LU-NCP have relatively few of ordinary accesses and lock context accesses (shown in Tables 2.12 and 2.13), thus, significantly reducing the overall cost of address space protection. Additionally, these applications also acquire few locks and have a significantly lower

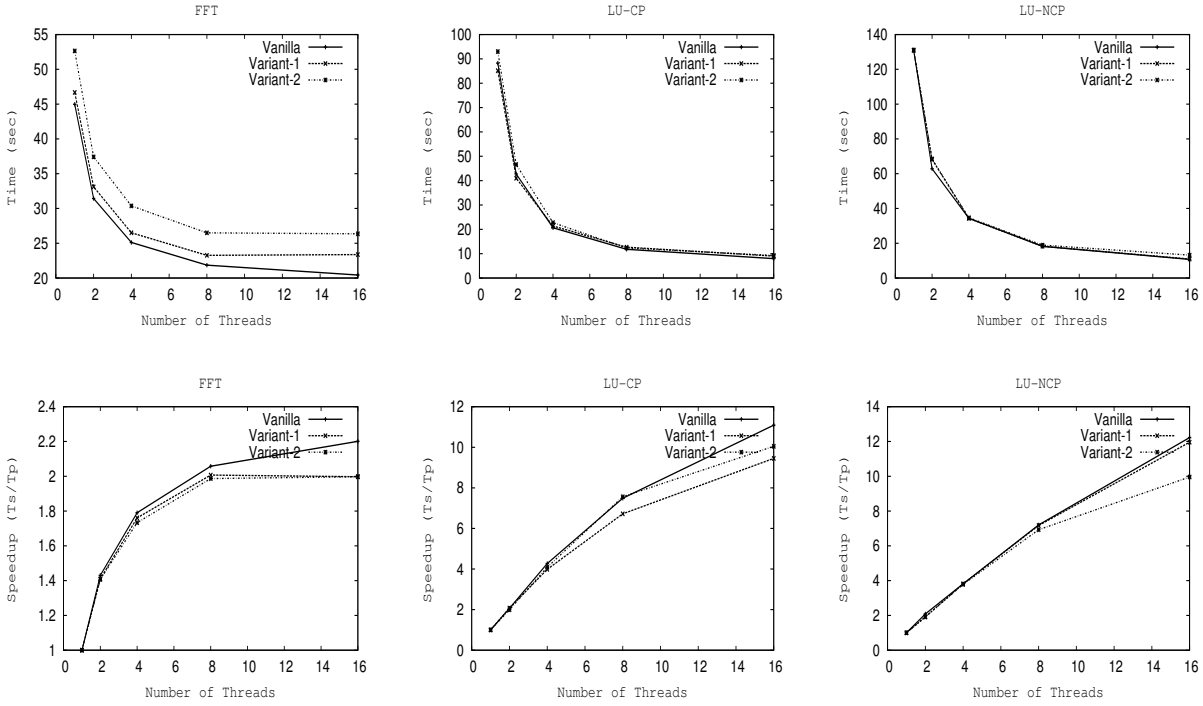


Figure 2.7: Performance of applications from Set-3 (low lock acquisition rate, typically 0.5 - 10/sec) with Sammati on a 16 core system.

Table 2.11: Characteristics of applications in Set-3.

Benchmark	Threads	Total Locks	Lock Rate (locks/sec)	Pages Modified in Lock Context			
				Total	Min	Max	Avg
FFT	1	1	0.02	1	1	1	1
	2	2	0.06	2	1	2	2
	4	4	0.16	4	1	4	4
	8	8	0.37	8	1	8	8
	16	16	0.78	16	1	16	16
LU-CP	1	1	0.01	1	1	1	1
	2	2	0.05	2	1	2	2
	4	4	0.19	4	1	4	4
	8	8	0.68	8	1	8	8
	16	16	2.01	16	1	16	16
LU-NCP	1	1	0.01	1	1	1	1
	2	2	0.03	2	1	2	2
	4	4	0.12	4	1	4	4
	8	8	0.44	8	1	8	8
	16	16	1.50	16	1	16	16

Table 2.12: Profile of applications in Set-3 with Sammati (Variant-1).

Benchmark	Threads	Total Pages			
		Write-protected	Access faults	Map shared	Un-write-protected
FFT	1	790634	1	1	790634
	2	1581268	2	2	1581268
	4	3162536	4	4	3162536
	8	6325072	8	8	6325072
	16	12650144	16	16	12650144
LU-CP	1	36867	1	1	36867
	2	73738	2	2	73738
	4	147492	4	4	147492
	8	295048	8	8	295048
	16	557552	16	16	557552
LU-NCP	1	36865	1	1	36865
	2	73730	2	2	73730
	4	147460	4	4	147460
	8	294920	8	8	294920
	16	589840	16	16	589840

Table 2.13: Profile of applications in Set-3 with Sammati (Variant-2).

Benchmark	Threads	Total Pages			
		Write-protected	Access faults	Map shared	Un-write-protected
FFT	1	524397	1048785	1	1048784
	2	524397	1048821	2	1048819
	4	524397	1048888	4	1048884
	8	524397	1049024	8	1049016
	16	524397	1049296	16	1049280
LU-CP	1	32918	65695	1	65694
	2	32920	82149	2	82147
	4	32924	91174	4	91170
	8	32932	97200	8	97192
	16	32947	103154	16	103138
LU-NCP	1	32787	65555	1	65554
	2	32787	98317	2	98315
	4	32787	163841	4	163837
	8	32787	294889	8	294881
	16	32787	556985	16	556969

lock acquisition rate (shown in Table 2.11) compared to applications from Set1 and Set2, thus resulting in negligable runtime overhead. The number of pages write-protected and un-write-protected in Variant-1 and Variant-2 are identical for LU and LU-NCP, hence, they have similar performance characteristics compared to native thread execution.

FFT has a slightly higher runtime overhead compared to LU-CP and LU-NCP and its overall performance is comparable to native thread execution. The overhead stems from the additional cost of address space protection. In FFT Sammati write-protects and un-write-protects approximately 2 orders of magnitude more pages than LU-CP and 3 orders of magnitude more pages than LU-NCP.

Set #4

Figure 2.3.2 illustrates the performance of Sammati and Pthreads for applications in set-4. The applications in set-4 do no acquire any locks. We nevertheless use these benchmarks in our experimental analysis to measure the overhead of our cords infrastructure (described in Section 2.2). The results show that Sammati's cords mechanism incurs no performance overhead and Sammati's performance is comparable to native thread execution.

Memory Overhead

Sammati's memory overhead stems from Sammati's metadata and the transient memory overhead due to privatization of memory updates.

$$Memory\ Overhead_{Sammati} = Sammati's\ metadata + write\_set \times cost\ of\ privatization\ (creating\ copy)$$

Sammati's metadata is relatively small at approximately $1 - 2$ MB independent of the number of cords. The privatization memory overhead is incurred when the application is in a critical section, as discussed in Section 2.2.3. This overhead stems from maintaining twin copies of a page, which are then used to compute XOR differences during inclusion. Note that the twin pages are freed at the end of the critical section, i.e., this memory overhead is transient. The memory footprint of the

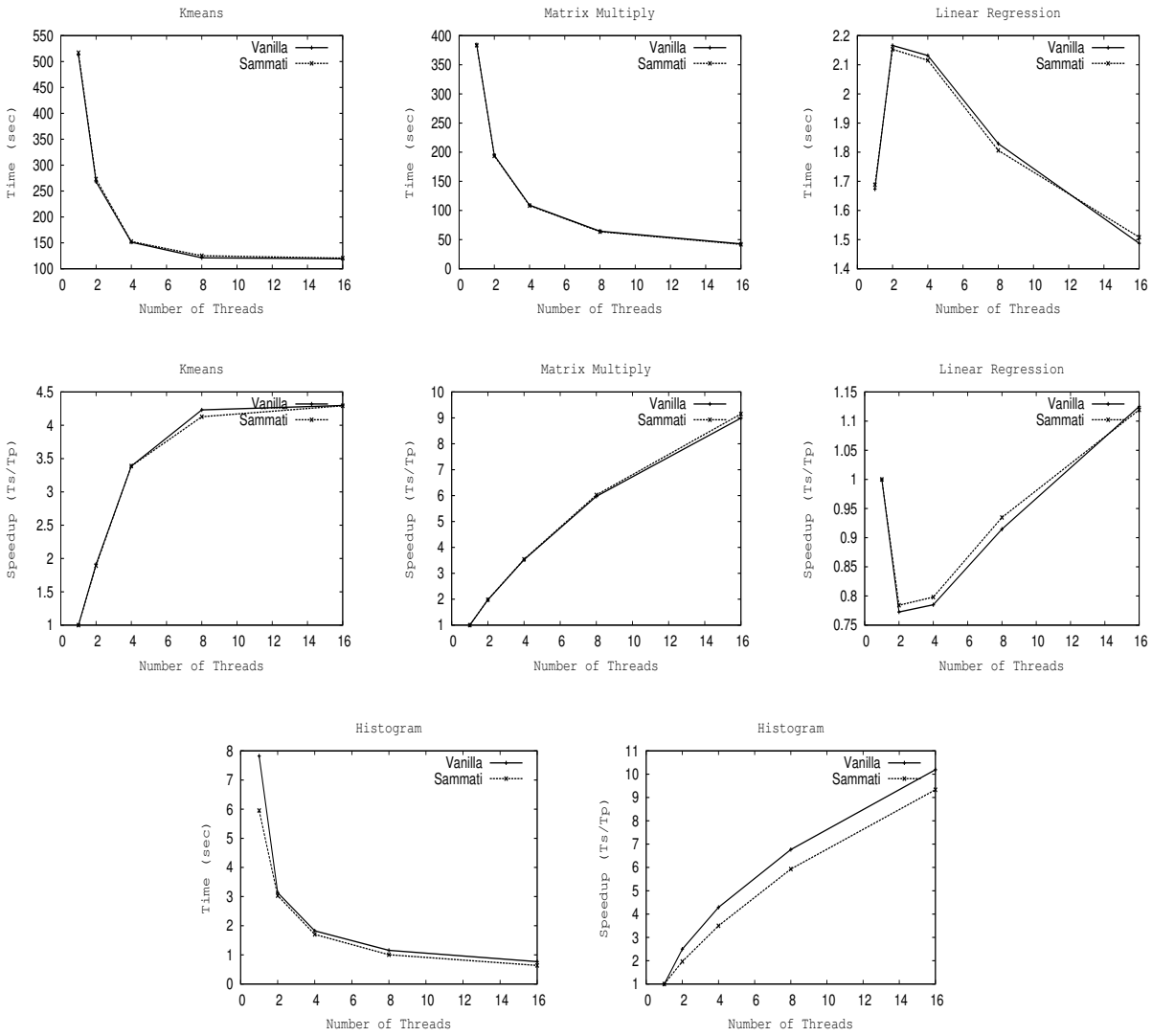


Figure 2.8: Performance of applications from Set-4 (extremely low lock acquisition rate, typically 0 - 0.5/sec) with Sammati on a 16 core system.

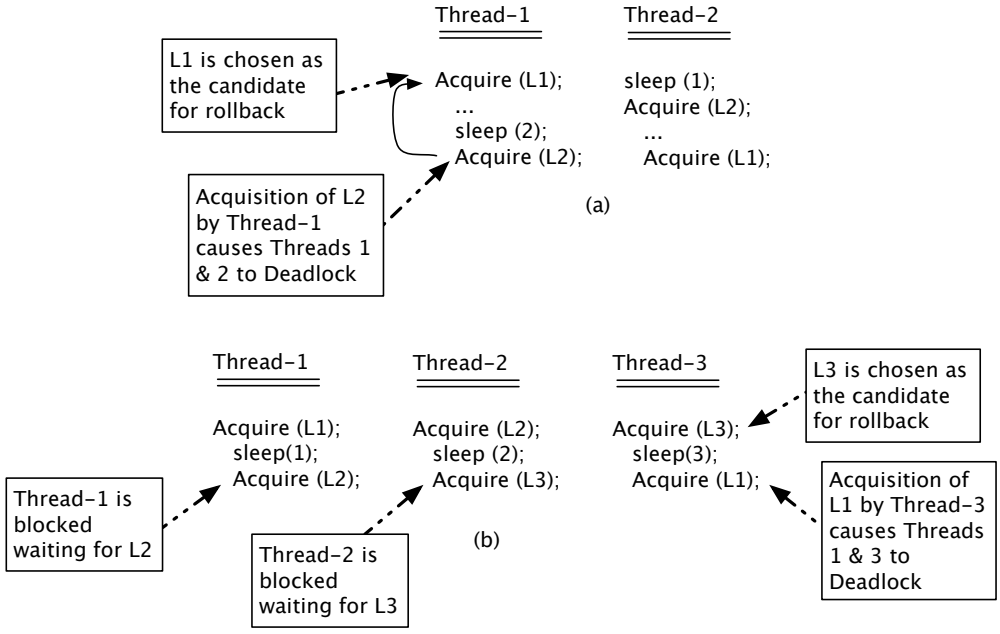


Figure 2.9: (a) Illustrates a simple deadlock between two threads due to cyclic lock acquisition. (b) Depicts a more complex example of deadlock involving more than two threads.

twin pages is proportional to the write-set in pages within a critical section. In principle, this is similar to the memory overhead of software transactional memory systems, except that Sammati operate at page granularity. To quantify this overhead, we measured the minimum, maximum and average number of twin pages within any critical section, which yields the upper bound on the transient memory overhead of Sammati. We present this information in Tables 2.4, 2.7 and 2.11.

A majority of the applications, including Ocean, Radix, PCA, FFT, LU-CP, and LU-NCP, modified only a few pages within any given critical section. The sum of the maximum number of twin pages for all cords was ≈ 16 pages. Barnes, FMM, and Water from Set-1 modified 308, 49, and 32 pages, respectively, for 16 cords. Consequently, Barnes incurred the highest memory overhead of 1.203 MB ($308 \times 4K$) of memory, FMM incurred a memory overhead of 196 K ($49 \times 4K$), and Water incurred an overhead of 128 K ($32 \times 4K$).

2.3.3 Deadlock Detection and Recovery

We created a synthetic benchmark suite that contains programs prone to deadlocks. We designed these programs to deterministically and randomly deadlock during the course of their execution. Additionally, we subjected Sammati to synthetic programs containing examples of deadlocks, taken from existing literature including [75, 9, 43, 42, 108]. In Figure 2.9 we present examples from our synthetic benchmark suite. In the first example, two threads (shown in Figure 2.9(a)) acquire locks (L1 and L2) in different orders that could potentially result in a cyclic dependency among threads depending on the ordering of the threads. In order to induce a deadlock, we added a *sleep* statement to thread 1 after the acquisition of lock L1. This results in a deterministic deadlock among the two threads. In Figure 2.9(b) we illustrate a more complex example involving a cyclic dependency of lock acquisition among multiple threads. The native Pthreads program hangs on such deadlocks. Sammati detects such deadlocks, recovers from them transparently and executes the program to completion.

2.3.4 Summary of Results

The experimental results indicate that Sammati is capable of handling applications that employ extreme fine-grain locking (e.g., 640K/sec for Barnes, 265K/sec for FMM, 70K/sec for Water). Sammati incurs a transient memory overhead of 2x over the total write set of a critical section. The memory overhead ranged between 64K to 1.203MB across all the applications used in this study. On the whole, while Sammati performs reasonably well across the spectrum of applications from the SPLASH and Phoenix suites, we find that address space protection and privatization costs primarily contribute to Sammati's runtime overhead. The performance of Sammati's cords mechanism is almost identical to the performance of native pthreads applications and incurs no overhead for applications with relatively low lock rates. We show that is possible to efficiently design and implement a runtime system that is capable of transparently detecting and recovering from deadlocks.

2.4 Limitations of Pure Runtime Approach

Although Sammati's pure runtime approach can deterministically and transparently detect and recover from deadlocks without requiring any modifications to the application, complier, or the operating system, there are some limitations.

2.4.1 Thread-Local Storage (TLS)

Sammati does not recover program state involving Thread-Local Storage (TLS) data in the event of a deadlock. We briefly discuss the semantics of TLS data (denoted by *__thread* in type information of a variable) and discuss the problem. In programs that employ TLS data, the dynamic linker, *ld*, allocates the zero initialized and un-initialized data in the *.tbss* section and the non-zero initialized data in the *.tdata* sections of the object file (.o). The linker creates an *initial* image of the TLS data (*.tdata* and *.tbss*) prior to the program execution starting at *main* and passes a copy of this image to each thread. In the presence of shared libraries (.so), the image is created when the shared library is loaded by the loader (lazy binding). Each thread receives a copy of the saved TLS state and any updates to the TLS are local to each thread.

To successfully recover from a deadlock, the entire program state of the thread should be recovered, including the global data, heap, stack, and TLS data. Unfortunately, recovering the TLS data is challenging. Since the TLS data in shared libraries and other program dependancies are not known a priori, it is practically not feasible to determine the TLS region transparently, thus precluding Sammati from employing address space protection to track updates to TLS. Chapter 3 presents a comprehensive solution to address this issue.

2.4.2 Deadlock Recovery

Sammati can detect *all* deadlocks. However, it cannot recover from certain deadlocks if recovery involves unrolling non-idompotent actions such as I/O, condition signals, semaphores, etc. This is

an open research problem.

2.4.3 Ad-hoc Synchronization

Sammati's privatization semantics conflict with applications that employ ad-hoc synchronization within critical sections. Consequently, Sammati cannot support such applications. We explore this topic in more detail in Chapter 3.

2.4.4 Address Space Protection Overhead

The high cost of address space protection and un-protection may preclude certain applications that have relatively large memory footprint and lock rate to benefit from Sammati.

2.5 Related Work

Sammati is primarily designed to address deadlocks and provide a platform for an effective composition of lock-based codes. Additionally, Sammati is capable of detecting certain kinds of data races (discussed in Section 2.2.7). In this section we briefly discuss literature related to deadlock detection and recovery, and data races.

2.5.1 Deadlock Detection and Recovery

Static Analysis

Several systems [15, 29, 30, 97, 64, 88, 90, 107, 71, 102, 32] based on program analysis were proposed to determine deadlocks. While program analysis can identify certain deadlocks based on information obtained at compile time, unfortunately, it cannot identify all deadlocks in weakly typed languages such as C or C++. Furthermore, such an approach may generate false positives in identifying deadlocks resulting in spurious recovery actions. In contrast, Sammati is implemented as a pure

runtime system with optional compile time extensions. Sammati's runtime employs a deterministic algorithm to detect and eliminate deadlocks with no false positives or negatives. Furthermore, Sammati does not require any modifications to the source code and it is completely transparent to the application.

Dynamic Analysis

Dynamic analysis tools [52, 3, 8, 34, 35, 43, 45, 77, 105] detect deadlocks at runtime. We discuss several in detail.

Li et al. [52] proposed Pulse, an operating system technique to dynamically detect deadlocks. Pulse scans for processes that are blocked for prolonged periods of time. To identify deadlocks Pulse speculatively executes the blocked processes to identify dependencies. Pulse builds a resource graph and traverses it to detect cycles (deadlocks). Pulse can also detect deadlocks that occur due to semaphores and pipes in addition to locks. To ensure safety and program correctness, Pulse cannot perform I/O while speculatively executing processes, which can lead the program to traverse potentially incorrect code-paths, resulting in false positives. Additionally, Pulse suffers from several false negatives and cannot detect all deadlocks that occur at runtime. For instance, if the granularity of monitoring for deadlocks is large, or if the speculative process executes code paths that are different from the future events that the blocked process would perform when awakened. In contrast Sammati, does not require any modifications to the operating system and does not involve any false positives or negatives. Additionally, Sammati is capable of recovering from deadlocks, unlike Pulse.

Harrow [34] proposed Visual Threads, a framework to check concurrency bugs at runtime. Visual Threads instruments the binary of the program and collects traces of events (lock acquisitions, releases, etc.). It then uses the events to model the execution of the application using a state machine and employs runtime checks to detect bugs including deadlocks. Bensalem et al. [8, 7] proposed a runtime verification algorithm to detect potential deadlocks in applications written in Java. In their approach they instrument the bytecode to collect a program execution trace

composed of a sequence of events. Their algorithm applies a set of rules on the stream of events at runtime to detect potential deadlocks. The authors claim that their approach reduces the number of false positives. Agarwal and Stoller [3] proposed a runtime technique to detect potential deadlocks that arise due to semaphores, condition variables, and locks. In their approach they collect execution traces for the program and generate feasible permutations of the program traces to detect potential deadlocks. In contrast to such systems, Sammati does not collect any traces; instead it deterministically detects deadlocks at runtime resulting in no false positives.

Joshi et al. [43] proposed a technique called Deadlockfuzzer to detect deadlocks. Their approach employs an imprecise randomized scheduler to create deadlocks with a certain probability. They propose an extension of the Goodlock [35] algorithm to detect potential deadlocks in multi-threaded applications. In contrast, Sammati employs a deterministic alogrithm to detect deadlocks and is capable of recovering from deadlocks, unlike Deadlockfuzzer.

Jula et al. [45] proposed Dimmunix to enable applications to develop immunity to deadlocks. Dimmunix captures the signatures of deadlocks as they occur during execution and it aims to avoid entering into the same pattern that resulted in a deadlock. Dimmunix's runtime maintains a history of deadlocks and intercepts all lock acquisition and release operations. For every lock acquisition, Dimmunix sends a request message to its deadlock avoidance thread to determine if it is safe to acquire a lock. The deadlock avoidance thread employs a resource allocation graph to represent a program's synchronization state, and control flows to identify code paths that led to a deadlock. Dimmunix avoids deadlocks by maintaining the state information of each deadlock pattern that occurs at runtime and aims to prevent such future occurrences through deadlock prediction. Dimmunix is capable of handling a wide-range of applications including those written in Java, C, and C++. Unfortunately, it is susceptible to false positives.

Qin et al. [77] proposed Rx, a technique to recover programs from software bugs. In their study, they checkpoint the application periodically and upon a software failure, rollback the program to the most recent (in program order) checkpoint, and re-execute the program under a new environment (perturbed original environment by artificially introducing noise, e.g., delay freeing of buffers,

asynchronous signaling, etc.). Rx requires modifications to the kernel and the application. Rx is capable of detecting a wide range of software bugs. In contrast, Sammati detects only deadlocks and certain kinds of data races, but does not require any modifications to the operating system. Additionally, Sammati performs efficient deadlock recovery without requiring a complete application checkpoint and the associated overhead.

In another study, Wang et al. [105, 106] proposed Gadara, a technique to avoid deadlocks. Gadara employs program analysis to develop a model of the program. It then employs discrete control theory to process the control flow that avoids deadlocks in the model. Gadara then instruments the program source with hooks to the runtime. These hooks control the execution flow of the program avoiding potential deadlocks.

Recently, Gerakios et al. [31] proposed a deadlock avoidance technique. In their approach, they employ program analysis to collect the order of lock acquisitions and releases. They propose that a deadlock can be efficiently avoided if information is available on the lock currently being requested, and the set of locks acquired between the lock acquisition and its subsequent release, referred to as future set of the lock. To avoid deadlocks, in their approach, they grant a lock only when both the requested lock and its future lockset are available. Their approach suffers from the several limitations of program analysis. In contrast, Sammati employs a pure runtime approach that uses a deterministic algorithm to detect and eliminate deadlocks. Sammati does not rely on source analysis or require any modifications to the source code.

Berger et al. [9] proposed Grace, a runtime system that eliminates concurrency bugs including deadlocks. Grace employs sequential composition of threads with speculative execution to achieve speedup. Grace supports applications written to leverage fork-join parallelism. Grace treats locks as no-ops and consequently eliminates deadlocks. In contrast Sammati is capable of supporting a broad range of applications, not necessarily those limited to fork-join parallelism.

Joshi et al. [42] proposed CheckMate to detect a broad range of deadlocks resulting from locks, condition variables, and other forms of synchronization primitives. CheckMate collects a program trace during a deadlock free run and records operations such as lock acquisitions, releases etc.,

relevant to finding deadlocks. It then employs a model checker to explore possible thread interleavings from the information collected in the program trace. The model checker checks for potential deadlocks. CheckMate is a predictive dynamic analysis tool, so it is prone is false positives and false negatives.

Several techniques [62, 63, 89, 16, 28] based on increasing thread interleaving through scheduler noise were proposed to increase the chances of detecting bugs while testing concurrent programs. In contrast to such systems, Sammati employs a deterministic algorithm to detect and eliminate deadlocks that occur during a particular execution of a program. Sammati does predict potential deadlocks that might arise during other interleavings of program execution.

2.5.2 Transactional Memory (TM)

Transactional memory [36, 91] introduces a programming model where synchronization is achieved via short critical sections called transactions that appear to execute atomically. The core goal of transactional memory is to achieve composability of arbitrary transactions, while presenting a simple memory model. Similar to lock based codes, transactions provide mutual exclusion. However, unlike lock based codes, transactions can be optimistically executed concurrently, leading to efficient implementations. Sammati may be viewed as a pessimistic STM without optimistic concurrency. Other studies [84, 103] have focused on the usefulness of TM and applying the utility of transactions to meet synchronization requirements. Recently, Volos et al. [103] studied the effectiveness of TM in fixing concurrency bugs. In their study they developed custom bug fixes with transactional semantics to fix concurrency bugs. In contrast, Serenity is a debugging tool.

2.6 Summary

In this chapter we presented Sammati, a runtime system for transparent deadlock detection and recovery in POSIX threaded applications written in type-unsafe languages such as C and C++. We implemented the runtime system as a pre-loadable library and its use does not require either

the application source code or recompiling/relinking phases, thereby enabling its use for existing applications with arbitrary multi-threading models. We discussed the design, architecture, and limitations of Sammati. We presented the results of a performance evaluation of Sammati using SPLASH, Phoenix and synthetic benchmark suites. Our results indicate that Sammati performs reasonably well even in the presence of fine-grain locking.

Chapter 3

A Program Analysis and Runtime Approach

In Chapter 2 we presented a pure runtime approach for detecting and eliminating deadlocks. Sammati operates on native binaries; it neither requires access nor any modifications to source code. While this works for certain applications, the high costs of address space protection and privatization may preclude certain applications that have reasonably large memory footprint and lock rate to benefit from Sammati. Furthermore, Sammati comes with other limitations (discussed in Section 2.4).

In this chapter we present techniques to address prior limitations and realize our vision as Serenity [76]. Serenity leverages containment and propagation techniques from Sammati and improves upon Sammati's deadlock detection and recovery algorithms. Our observation here is that program analysis and compile time instrumentation can guide a runtime to efficiently achieve memory isolation. By enforcing the following two design constraints — availability of program source and recompilation/relinking of program source which are otherwise relaxed in our pure runtime approach, we can build a practical system that can eliminate deadlocks in real-world applications and achieve wide spread acceptance.

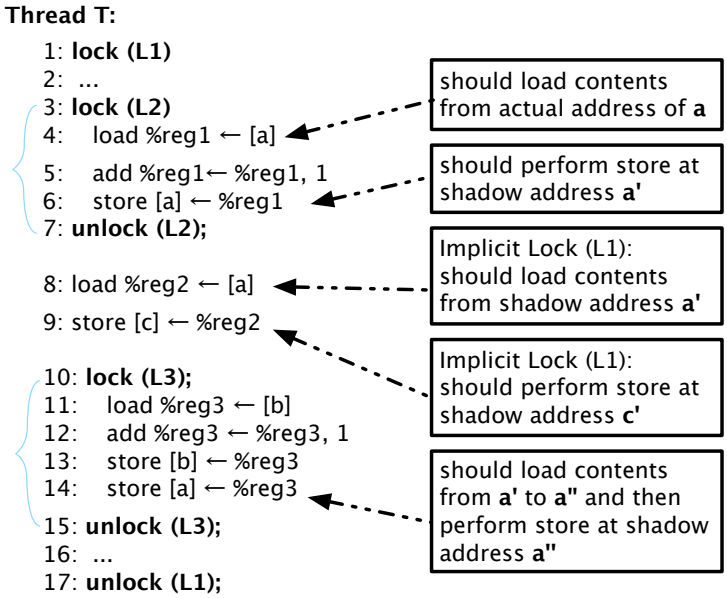


Figure 3.1: Challenges in isolating memory updates.

The rest of the chapter is organized as follows. Section 3.1 presents the runtime system of Serenity. Section 3.2 presents the compile time extensions employed by Serenity. Section 3.3 presents a comprehensive performance evaluation of Serenity. Section 3.4 discusses the limitations of our approach. Section 3.5 reviews existing systems for deadlock detection and recovery, noting differences with our approach. Section 3.6 compares this work with the transactional memory systems. Section 3.7 describes future directions and Section 3.8 summarizes this chapter.

3.1 Runtime System

Serenity's runtime system is responsible for containment and propagation of memory updates, deadlock detection and recovery, and detection and recovery from several other programming errors.

3.1.1 Isolating Memory Updates

To achieve efficient isolation and be able to support granular deadlock recovery and resolve priority inversion, we need mechanisms to a) associate memory updates with specific locks (for instance,

in a nested lock sequence, updates within each lock should be distinctly identified and associated with the corresponding lock), and b) isolate these memory updates from being visible until all locks protecting a critical section are released.

In languages that permit direct memory accesses through pointers, associating memory updates with distinct locks cannot be done precisely at compile time. Providing efficient isolation and containment of memory updates within critical sections presents additional challenges. Consider the example shown in Figure 3.1. If the variable a is written under lock *L2* by a thread T, then the system should prevent the update from being visible to other concurrently executing threads. Second, the isolation mechanism must also guarantee program order in that any successive reads to variable a by thread T should return the value most recently written to a by T. Third, to enable granular deadlock recovery the isolation mechanism must distinguish and isolate data modified under distinct locks, e.g., the variable a in Figure 3.1 is modified under two locks, *L2* and *L3*.

To implement an efficient isolation mechanism, Serenity defines two contexts of execution for a thread, *lock context* if it is executing a critical section protected by one or more locks, and *ordinary context* if it does not hold any locks. To perform isolation, in a lock context Serenity's runtime replaces stores to the actual target address with stores to a different virtual address called the *shadow address*. Hence any memory updates are localized to the thread executing the critical section, thereby implementing containment. To distinguish memory updates under distinct nested locks (Figure 3.1), each nested lock creates a new mapping between target addresses and shadow addresses, in effect shadowing the shadow address. Reads (*loads*) are serviced from the most recently (in program order) shadowed address if one exists; otherwise they are read from the actual virtual address, thus ensuring program order.

In the discussion above, we overlooked the memory alignment used by the shadowing mechanism. Modern ISAs such as x86 permit arbitrary byte alignment of memory addresses and arbitrary length stores (up to the ISA's word length). The use of bitfields in languages such as C further complicates the problem. Our observation here is that the LLVM IR provides an idealized 64 bit ISA. Aligning all target addresses (and shadow mappings) to 64 bits and then tracking bit level updates within the

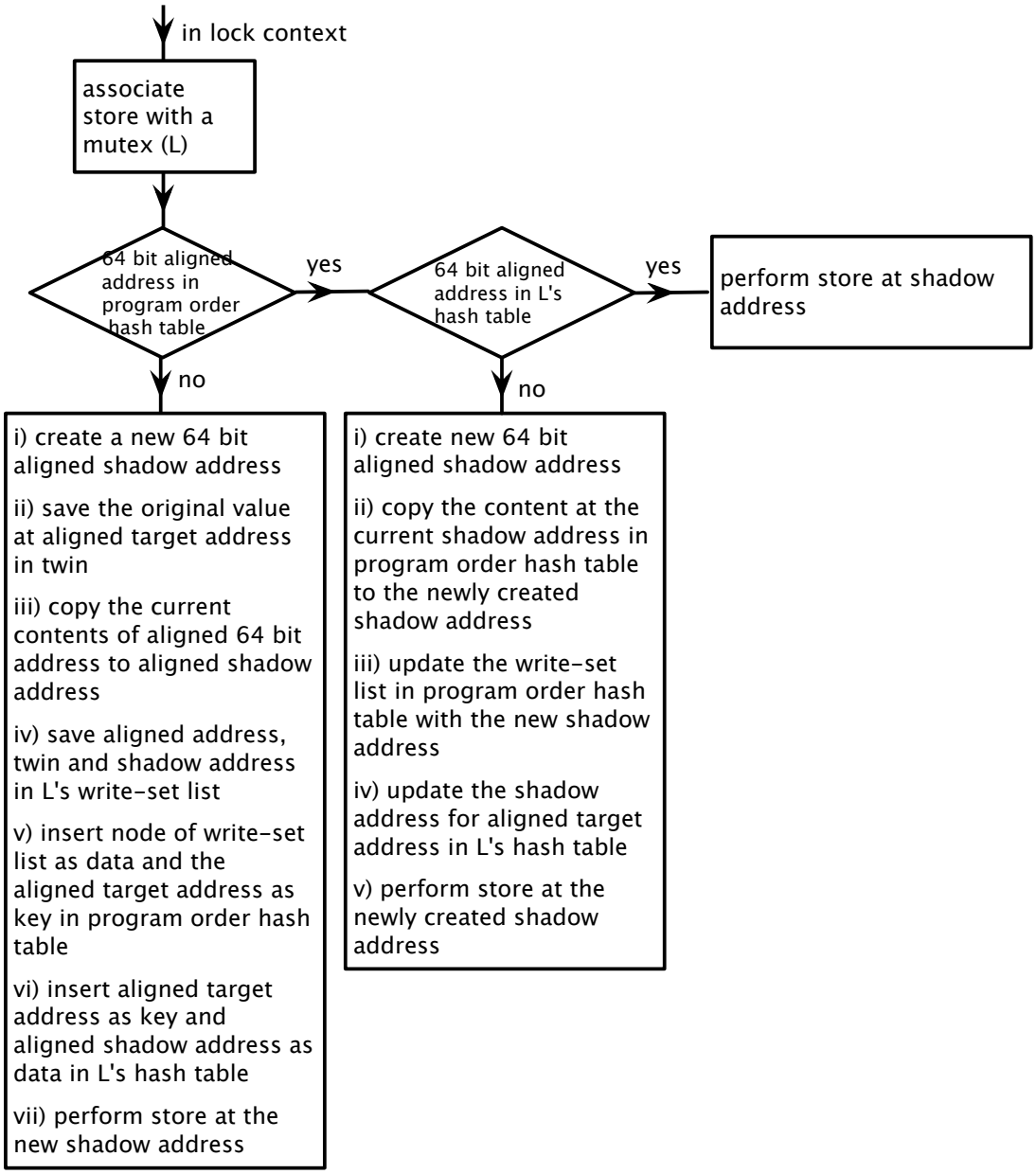


Figure 3.2: Isolating Store

64 bit boundary enables us to implement shadowing at arbitrary address and length granularity. Bit level updates are efficiently tracked using a word length version of the twin/diff technique used in distributed shared memory (DSM) systems (TreadMarks [4]) for identifying false sharing. In this technique (DSM systems used this technique at 4K page granularity, we use it at 64 bit granularity), before a store is performed, we copy the original contents (64 bits) to a memory location called the *twin*. After the store is performed, the exact set of bits modified by the store can be determined by computing the XOR of the updated contents with the twin. Since the maximum length of an update in the LLVM 64 bit IR cannot exceed the register length of 64 bits, this technique efficiently captures arbitrary updates at bit level granularity, without requiring byte-level shadowing or interval trees that can handle partially overlapped load and store operations. This shadowing technique represents the optimized culmination of several failed attempts and is nearly an order of magnitude faster than the best known shadowing methods [81].

In the rest of this discussion, we use the term critical section to refer to the entire scope of the outermost lock in a nested lock sequence, e.g., scope of lock *L1* in Figure 3.1.

Shadowing Stores

Figure 3.2 illustrates how Serenity implements shadowing of stores within a critical section. Given a target address on the heap, its length and type, the runtime system identifies the latest lock acquisition (L) in program order and associates the store with L. The runtime then computes a 64 bit aligned version of the target address (i.e., target address & 0xfffffffffffffff8). The system checks if the aligned address is associated with any other lock previously acquired within the same critical section. In order to accomplish this, we employ a *program order hash table* common to all locks acquired by the thread within a single critical section. As the name implies, the program order hash table tracks the mapping between a target address and a shadow address in program order.

If an entry exists in the program order hash table, this implies that a store was performed previously on the aligned address within the same critical section either under lock L or in the context of a different lock. In order to efficiently perform this lookup, we use a hash table associated with each

instance of a lock (including implicit locks), with the aligned address as key and its corresponding 64 bit shadow address as data. If an entry exists in L's hash table, the runtime performs a store on the previously assigned shadow address. Else, to distinguish between updates to the same address under different locks, the runtime (a) creates a new shadow address and copies the contents from the previous shadow address per the program order hash table, (b) performs the store at the newly created shadow address, and (c) updates the program order hash table and L's hash table with the new shadow address mapping.

The absence of an entry in the program order hash table indicates that this is the first store to the address within the critical section. In this case, the runtime creates a new shadow address mapping and copies the contents at the 64 bit aligned target address to the newly created 64 bit shadow address. It then (a) saves the original contents at the aligned target address in a *twin*, (b) performs the store at the shadow address, and (c) updates L's hash table and the program order hash table with the new shadow address mapping for the aligned target address.

Since a large critical section may have a significant number of memory updates, we need to minimize the cost of creating and destroying the tables. The cost of creating hash tables is reduced by maintaining a *free list* of hash tables. To reduce the overhead of deleting all the entries from the hash tables on successful completion of a critical section, we use a novel aging technique. Each hash table and each hash entry has a monotonically increasing epoch variable. When the runtime obtains a hash table from the free list it increments the epoch of the hash table, and every time an entry is inserted in the hash table, the runtime sets the hash entry's epoch to the epoch of the hash table. If an entry already exists in the hash table for a given key and if the entry's epoch is less than the hash table's epoch, the runtime simply reuses the shadow address. In effect, the epoch variable implements a Lamport clock that defines the age of the hash entry. At the completion of a critical section, the hash table is simply returned to the free list without clearing its entries. This technique allows us to reap older hash entries (defined as hash entry's epoch $<$ hash table epoch) optimistically and avoids the significant cost of clearing hash table entries on inclusion. This data structure, which has $O(1)$ construction and destruction time, more than halved the overhead of

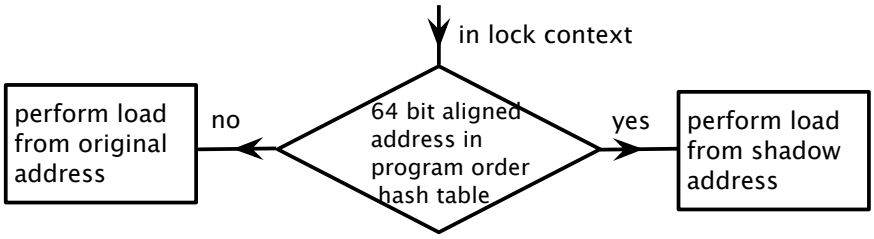


Figure 3.3: Isolating Load.

load/store instrumentation in Serenity.

Shadowing Loads

Figure 3.3 illustrates how Serenity's runtime handles memory load operations. Given the target address, its length and the type of the value being returned, the runtime computes the 64 bit aligned address of the target address and checks the program order hash table to see if a prior store has occurred on the aligned address. Recall that the program order hash table always points to the most recently created shadow address for the aligned address. If an entry exists in the program order hash table then the runtime returns the value from the shadow address reflecting the most recent (in program order) store to that address. In the absence of an entry in the program order hash table, the load operation directly reads the target address.

3.1.2 Propagating Memory Updates

Propagating updates in the presence of nested locks presents several challenges. Turning again to the example in Figure 3.1, if the update to variable a protected by lock *L2* were made visible immediately after the release of *L2*, and subsequently a deadlock occurred on the acquisition of lock *L3*, where the victim was thread T (holding *L1*), there would be no way to unroll the side-effects of making the update to variable a visible. This would prevent Serenity from performing granular deadlock recovery. Furthermore, in the presence of lock inversion it is not possible to transparently determine the actual lock (or deduce the programmer's intent) that is uniquely responsible for

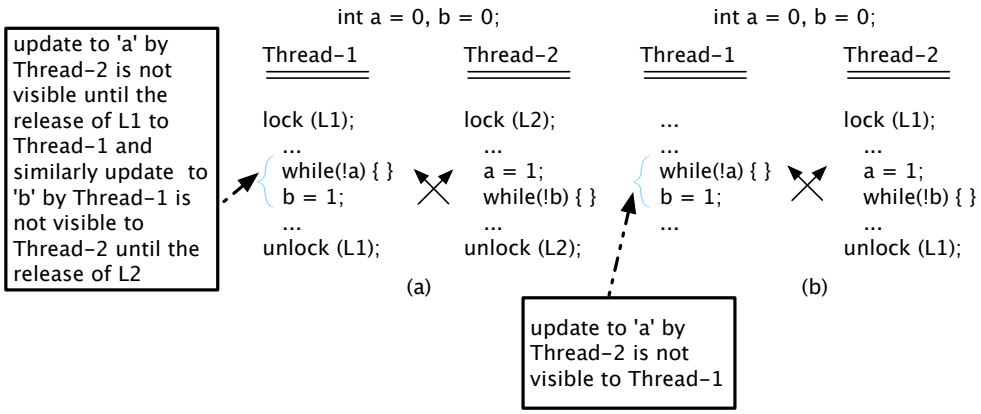


Figure 3.4: Side-effects of privatization and propagation semantics could result in a deadlock.

ensuring mutual exclusion on data.

To circumvent these issues, Serenity leverages the propagation semantics of Sammati discussed in Section 2.1.2. We recap them for the sake of completeness. Serenity makes the modified state visible to other concurrently executing threads only when all locks within a nested lock sequence have been released. Additionally, to preserve the semantics of mutual exclusion, Serenity tracks the release of locks in program order in a nested lock sequence and defers performing the intermediate lock releases till all locks around a critical section have been released in program order. In the presence of deferred releases, the runtime system elides a subsequent lock acquisition if a thread attempts to reacquire a deferred release lock in the same nested lock sequence. Serenity first performs inclusion of all memory updates (discussed in Section 3.1.3) prior to releasing the locks.

Recall that Sammati's propagation semantics are prone to certain side-effects. Transforming lock semantics to transactional semantics is shown to result in deadlocks [22, 23, 60] in the presence of ad-hoc synchronization in critical sections.

Side-effects of Propagation Semantics

In Figure 3.4 we present a few examples based on Blundell's et al.'s work [22, 23] on the subtleties of transactional memory and atomicity semantics. Consider the example shown in Figure 3.4 (a). Threads 1 and 2 run to completion in the absence of Serenity's privatization. Recall that

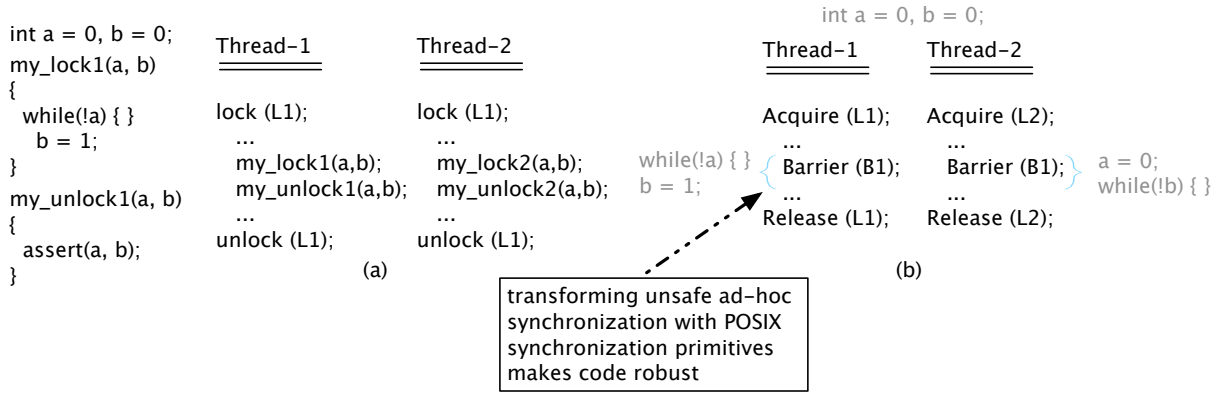


Figure 3.5: Simple transformations makes the example described in Figures 3.4 (a) and (b) circumvent the subtleties of Serenity's propagation semantics and also makes the code safe.

privatization is used for containment of memory updates to facilitate recovery in the event of a deadlock. However, in the presence of privatization, since the update to variable '*b*' by Thread-1 is not made visible until the release of the lock *L1*, Thread-2 is unaware of the update and finds the value of '*b*' to be always 0. Similarly, the update to '*a*' by Thread-2 is also not made visible to Thread-1, resulting in a deadlock between threads 1 and 2. In Figure 3.4 (b) we present a similar example as Figure 3.4 (a) except that the ad-hoc synchronization by Thread-1 is not protected by a critical section.

In practice, codes that employ ad-hoc synchronization either in the presence of critical sections or otherwise are not safe and such a programming practice is shown to result in several concurrency bugs [108]. Furthermore, such codes assume and rely on certain memory consistency guarantees to propagate the updates. For instance, the update to variable '*a*' by Thread-2 may not be propagated to other concurrently executing threads (e.g., Thread-1) potentially running on other cores unless the program includes instructions to flush the memory updates immediately after the update. Hence, it is possible that the examples shown in Figures 3.4 (a) and (b) may fail to run and result in a deadlock on architectures that do not provide such guarantees, even in the absence of Serenity.

Providing support for mixed locking regimes (i.e., employing ad-hoc synchronization within critical sections) conflicts with any runtime system that supports transparent (containment through privatization without modifying source code) deadlock recovery. Consequently, Serenity is incapable

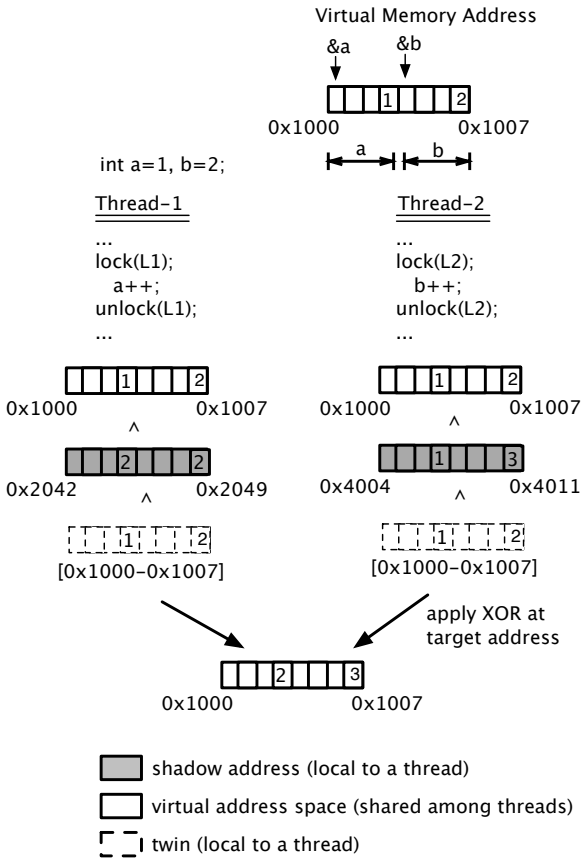


Figure 3.6: Committing memory updates

of running such codes while providing the guarantee of a transparent deadlock recovery. Rather than using ad-hoc synchronization schemes such as these, we recommend the use of traditional POSIX spin-locks, condition variables and signaling, and barriers for synchronization. As shown in Figure 3.5 (b), Serenity can then guarantee forward progress and transparent deadlock recovery.

Alternatively, in the event that the ad-hoc synchronization is an absolute necessity, the programmer can provide Serenity with a wrapper around the ad-hoc synchronization in the form of a user-defined lock as shown in Figure 3.5 (a).

3.1.3 Committing Memory Updates

To commit the updates to the program shared address space, Serenity must first determine the exact set of bytes modified within a critical section. To accomplish this, for every address in the critical section's program order hash table, Serenity computes a bitwise exclusive-or (XOR) difference between the contents at the shadow address and the twin. Recall that the twin is the copy of the contents at the target address that was saved prior to performing a *store* at the shadow address. This resulting difference precisely identifies the modified bits pointed to by the 64 bit aligned address. Serenity then updates the contents of the target address with the exclusive-or between the bytes it modified and the contents at the target address, thus making its updates visible to other threads. Serenity then frees the shadow addresses and twins. Since the operation of computing and applying the XOR differences to the target address space is a commutative operation, Serenity does not require any explicit serialization.

In Figure 3.6 we present a simple example to illustrate concurrent commit of updates by two different threads. The variables 'a' and 'b' are global and access to 'a' by thread *T1* is protected by a lock *(L1)*; access to 'b' by another thread *T2* is protected by a lock *L2*. Further, assume that both 'a' and 'b' are integers, each occupying 4 bytes in length and that variable 'a' is stored starting at a virtual address of 0x1000 and 'b' is stored at an address of 0x1004. When *T1* performs a store on variable 'a', under a critical section (*L1*), it first computes the 64 bit aligned address from the virtual address, i.e, 0x1000. It then creates a shadow of the 8 bytes starting at 0x1000 at a shadow address, say 0x2042, and saves a copy of the 8 bytes at the aligned virtual address 0x1000 in a twin. Finally it performs a store starting at the byte-offset within the shadow address (also 64 bit aligned) as shown in Figure 3.6. Similarly, *T2* may concurrently create a shadow starting, at say 0x4004, and a twin while performing its store on variable 'b'. On release of *L1*, *T1* computes its modified bytes and then applies them to the virtual address 0x1000, which is shared by all threads, thereby propagating its updates to the remaining threads. Concurrently, *T2* may release *L2*, and commit its updates similar to *T1* without impacting the correctness of the updates from *T1*.

3.1.4 Deadlock Detection

Serenity implements an efficient version of a *waits-for* graph described in Chapter 2.1.3. We briefly discuss it for the sake of completeness. Serenity employs a *waits-for* graph G (T, →), where an edge (→) from Thread T_i to Thread T_j in G implies that T_i is waiting for T_j to release a lock that T_i needs. Cycles can occur between two (e.g., $T_i \rightarrow T_j \parallel T_j \rightarrow T_i$) or more (e.g., $T_i \rightarrow T_j \parallel T_j \rightarrow T_k \parallel T_k \rightarrow T_i$) threads. Serenity employs two key data structures to efficiently traverse G: the set of locks currently owned (held) by a thread, and the lock on which a thread is currently waiting. These are represented by the two hash tables *holding set* (*key: lock, value: thread id T*) and *waiting set* (*key: thread id T, value: lock*), respectively. Since threads share a global address space, each thread can access the lock ownership of all other threads.

The idea is that a thread T_i can detect a deadlock at lock acquisition by first performing a (non-blocking) trylock to acquire the requested lock L. If the acquisition succeeds, T_i inserts L into its holding set. If the acquisition fails (i.e., $T_i \rightarrow T_j$), we use the holding set to find the thread T_j that owns the lock L and check if T_j is waiting on another lock S held by some thread T_k, i.e., $T_j \rightarrow T_k$. If T_j is not waiting on any lock, then there is no cycle (hence no deadlock, but contention) and the algorithm inserts L into the waiting set and T_i safely attempts to acquire L through a blocking lock acquisition. If T_j is waiting on a lock S held by T_k, the algorithm checks to see if T_k is waiting on another lock and so on. Intuitively, Serenity traverses the waits-for-graph to detect a cycle, which indicates a deadlock. If a deadlock is detected, the corresponding lock held by thread T_i in the holding set is ordinarily chosen as the victim (unless thread T_i performed any disk-I/O within the critical section). We explain how serenity handles disk-I/O in Section 3.1.5.

Sammati serializes accesses to waiting and holding sets to preserve correctness of the waits-for graph. Such serialization imposed at every lock acquisition by a thread has a significant impact on the runtime performance of an application. Hence, Serenity implements a more relaxed approach while guaranteeing the same determinism in detecting and recovering from a deadlock. A necessary pre-condition for a deadlock is that a thread has to *hold one or more locks* and be *waiting on at least one lock*. Our observation here is that a deadlock cannot occur when a thread attempts to

acquire the first lock protecting a critical section, since in this case, the thread does not hold any prior locks. We use this observation to avoid running the deadlock detection and recovery algorithm on the acquisition of the first lock.

3.1.5 Deadlock Recovery

Serenity leverages the deadlock recovery algorithm discussed in Section 2.1.4. For the sake of completeness, we recap the recovery mechanism here. Once the deadlock detection algorithm identifies a lock L for rollback, the algorithm then selects the oldest acquisition of L in the lock list and uses its recovery point (execution context and stack) to recover from the deadlock.

Thread State and Heap

To recover from the deadlock, Serenity first discards all the memory updates[1] performed in program order associated with locks in the lock list including and after L. Note that here the memory updates include the TLS regions, unlike Sammati. Serenity at compile time instruments all memory accesses performed within a critical section including thread-local variables. Consequently, Serenity can recover updates to TLS data. Second, it releases L and all the locks acquired after L in program order of acquisition. Third, it removes all locks released from the holding set and restores the execution context (*longjmp*) to the recovery point of L, which transfers control back to deadlock free lock acquisition of the victim lock L. Serenity performs *deadlock aware* memory management while recovering from a deadlock. Serenity exposes dynamic memory management through the standard POSIX memory management primitives including *malloc*, *calloc*, *realloc*, *valloc*, *free*, and the *mmap* family of calls. Serenity tracks all memory allocations and deallocations made within a lock context. During recovery Serenity internally frees the allocations to prevent memory leaks. Serenity buffers (delays) deallocations until all the locks protecting a critical section are released.

[1]includes updates to globals (.data, .bss), heap, stack, and thread local storage (TLS) regions of the virtual memory.

Disk I/O

Serenity serializes disk I/O operations performed within critical sections. If a deadlock arises between two threads that both performed I/O then it would be impossible to recover either of them. Serenity defines a thread to be in an I/O context if it performs an I/O operation within a lock context. A thread remains in an I/O context until it releases all the locks protecting its critical section, essentially preventing multiple threads from entering into an I/O context. Serenity does not serialize ordinary (non-lock context) I/O operations and I/O operations that happen between a lock context and an ordinary region since they cannot result in deadlocks. In the event of a deadlock, Serenity forces a non-I/O context thread to rollback, thus biasing the deadlock recovery algorithm to selecting a victim that has not issued an I/O operation. Additionally, Serenity treats thread/process creation and termination operations performed within a critical section as performing a disk I/O operation and precludes such threads from having to rollback by employing a bias against their selection during recovery. Serenity's runtime tracks all disk I/O primitives such as *read*, *write*, *fread*, *fwrite*, etc., and all thread creation/termination primitives such as *fork*, *clone*, etc., to ascertain if such operations are performed within a lock context.

3.1.6 Detecting Programming Errors

Serenity is capable of detecting certain concurrency bugs that commonly occur in practice in lock based codes. We briefly discuss these capabilities of Serenity.

Detecting Asymmetric Write-Write Data Races

Serenity can detect and identify all *write-write* data races that occur during an execution due to a) protected and concurrent unprotected updates to shared data and, b) improperly protected updates where the same shared data is modified concurrently under two or more distinct locks. The update performed in a critical section can be obtained by the exclusive-or difference between the twin and its corresponding content at the shadow address (discussed in 3.1.3). A non-zero value

at byte granularity indicates that the critical section has modified the corresponding byte. While committing the updates, by comparing the modified bytes in the twin with its equivalent byte in the target address space, Serenity determines if a particular byte has been modified both within the critical section performing inclusion and outside it.

Priority Inversion

Priority inversion is a well-known problem in real-time lock based programming. It occurs when a low priority thread acquires a lock L and is then preempted by a high-priority thread, which then blocks on the acquisition of L. The common solution to this problem is a heuristic called priority elevation, where the high priority thread elevates the priority of the low priority thread and relinquishes processor control in an attempt to get the low priority thread to complete its critical section and relinquish L. Priority elevation only works if the deadline for the high priority thread is sufficiently far in the future for it to relinquish control and not miss the deadline. Since Serenity contains all memory side-effects within a lock context, if a high priority thread is waiting on a low priority thread, the runtime can simply signal the low priority thread to abort the lock. Here aborting a lock in the low priority thread involves the same actions as deadlock recovery—discard memory updates, release the lock, restore program stack and restore control to just before lock acquisition in the low priority thread. We can delay the acquisition of the lock by the low priority thread until the high priority thread acquires the lock. The high priority thread can then acquire the necessary lock and make progress towards its deadline.

Asynchronous Signaling

In the presence of POSIX asynchronous signaling, programmers often erroneously assume a particular execution order among the threads. For instance, a thread may send a signal to another thread prior (in program order) to the recipient thread performing a wait (cond_wait) operation. Serenity tracks the POSIX condition signaling primitives. By observing the program ordering among condition variables (waits and their corresponding signals), Serenity can monitor lost signals or

broadcasts. We note that this form of *order violation* may not always be malicious; Serenity can be configured to ignore such an occurrence, throw a warning message, or raise an error in the presence of bugs.

Locks and Performance

Serenity can also detect programming errors involving locks, e.g., a thread attempting to release a lock without its corresponding acquisition. Furthermore, recent studies [38, 108] have shown that concurrency bugs can cause significant performance issues in addition to incorrect program behavior. Serenity can also provide useful performance statistics such as the number of memory accesses (loads/stores) performed within critical sections, and information on lock contention. Such information can significantly improve the overall quality of lock based code.

Order Violations

We note that Serenity in its current implementation does not detect order violations. In this discussion we explore how Serenity can be extended to assist the programmer in developing robust code that is free of order violations. In practice it is hard to detect order violations transparently since the notion of *order* is specific to a particular program context. Detecting order violations requires careful reasoning, understanding, and analysis of the program, often at best known only to the programmer. We propose to implement a programming API to extend Serenity's runtime to identify order violations, and to preserve program ordering. Presently with existing synchronization mechanisms (e.g., semaphores and condition signaling) even after the programmer has identified parts of the program that require ordering, translating those semantics in practice to code is tedious and often error prone. We propose to abstract the subtleties of such low-level programming by implementing an API. The idea is to allow the programmer to express the temporal notion of *happens-before* in a program. At runtime, Serenity can then check if a happens-before ordering is violated. Conceptually, the proposed API may be perceived as *assert* statements that check for program ordering.

Live-Locks

We note that Serenity in its current implementation does not detect live-locks. In this discussion we explore techniques to resolve common forms of live-locks that occur in practice. Live-locks are similar to deadlocks except that instead of being in a blocked state, the execution state of threads involved in a live-lock changes constantly with respect to each other. However, the end result of both deadlocks and live-locks is identical — they both prevent the program from making any meaningful progress.

Serenity can be extended to detect and recover from live locks that happen a) while recovering from deadlocks, and b) when threads repeatedly yield to each other in an attempt to acquire a sequence of locks. First, in the presence of deadlocks, during recovery, if Serenity's runtime detects that a deadlock is repeated, then it will select a different victim lock to avoid recurrent live-locks. Second, to address live-locks of type (b) we propose the following technique. The idea is to track the lock acquisitions (trylock and blocking), releases, and any voluntary yields (sched_yield, sleep, etc.) and construct a signature of the live-lock. To detect live-locks we propose to evaluate these signatures at runtime, obtained from all concurrently executing threads. To resolve the live-lock, we can employ Serenity's semantics of deadlock recovery, and employ techniques including privatization, and visibility rules. We can equip Serenity's runtime to unroll the updates performed within locks, causing the threads involved in live-lock to roll back to the first lock (in program order) in their respective live-lock signature. We then attempt to identify a schedule that avoids live-locks and guarantees progress in program execution. We propose to implement a user-level thread scheduler to schedule the threads based on the observed lock signatures of individual threads. In the absence of such a schedule, we propose to serialize the thread execution based on the observed set of signatures, thus guaranteeing program execution by recovering from live-locks.

3.2 Compile Time Extensions

Serenity's compile time instrumentation forms the basis of providing isolation of memory updates (discussed in Section 3.1.1). Serenity instruments the memory accesses in the LLVM's IR with hooks to Serenity's runtime. These hooks pass information about the kind of instruction (load or a store), the target address (precisely known only at runtime), its length and type information, thereby allowing Serenity's runtime to provide efficient isolation of critical sections.

A conservative approach to identify the memory updates performed by a program and associate them with critical sections is to instrument all memory accesses and privatize only the memory updates that happen within a lock context at runtime. While such an approach is sound and complete, it could potentially result in a non-trivial runtime overhead and may not be suitable for all applications. The key challenge here is to minimize the cost of instrumentation by instrumenting only the critical sections and avoiding ordinary regions.

Unfortunately, it is impossible to provide a sound and complete solution to this problem due to pointer aliasing. Consequently, to reduce the cost of instrumentation, Serenity employs opportunistic optimization via its lock scope analysis, that attempts to derive the scope (the code executed between the acquisition of a lock and its corresponding release) of locks in a program's control flow graph. This analysis, if successful, yields the set of basic blocks in the control flow graph that can be instrumented to capture all memory updates performed within a critical section, as opposed to instrumenting the entire program.

3.2.1 Lock Scope Idiom

Serenity's opportunistic optimization is motivated by the following observation. We carefully analyzed close to 50 widely used open-source real world applications to understand how programmers typically use locks and structure their code in writing lock based code. Some of these applications include SQLite, Firefox 5.0, MySQL, Thunderbird, Apache, OpenLDAP, OpenOffice, Squid, Sendmail, PBzip2, and several other desktop and server applications, and all the applications in the

Phoenix, PARSEC, and SPLASH benchmark suites. We found an interesting property exhibited by *all* of them —*lock acquisition and the corresponding release happen within the same procedure in the program and critical sections were scoped similar to transactions in that there was no aliasing of lock variables between a lock acquisition and its corresponding release.* This **lock scope idiom** significantly reduces the complexity of lock scope analysis to mostly intra-procedural (most common case) rather than inter-procedural analysis.

3.2.2 Lock Scope Analysis

Intuitively, Serenity treats the scope of a critical section as an atomic transaction and instruments all loads and stores within a scope. Serenity's lock scope analysis is implemented using LLVM. The analysis starts with constructing a control flow graph (CFG) for every procedure in the program that acquires/releases lock(s), with the basic blocks in the procedure as the vertices and edges representing the program control flow. The CFG (V, E, Entry, Exit(s)), where each vertex V is a basic block (bb_i) and there exists an edge E such that $V_{bb_i} \rightarrow V_{bb_j}$ if bb_j may immediately follow bb_i for some execution sequence. If $V_{bb_i} \rightarrow \emptyset$ then bb_i is a leaf node in the CFG and it is treated as an exit point of a procedure.

Serenity defines a lock to be *procedurally-scoped* if and only if (a) the acquisition of a lock precedes its release, (b) every path from a basic block containing the lock acquisition to the exit(s) of the procedure contains the corresponding lock release, and (c) every path from a basic block containing the lock acquisition to the exit(s) of the procedure contains no store to any location x such that x $\in$ AliasSet (lock).

If every lock use in a procedure is procedurally-scoped then a procedure is well-scoped. By extension a program is well-scoped if every procedure is well-scoped, or equivalently if every lock use in the program is procedurally-scoped.

We note that while Serenity's lock scope analysis may be imprecise, it is ultimately conservative and guarantees safety. In situations where it cannot conclusively establish the well-scopedness of

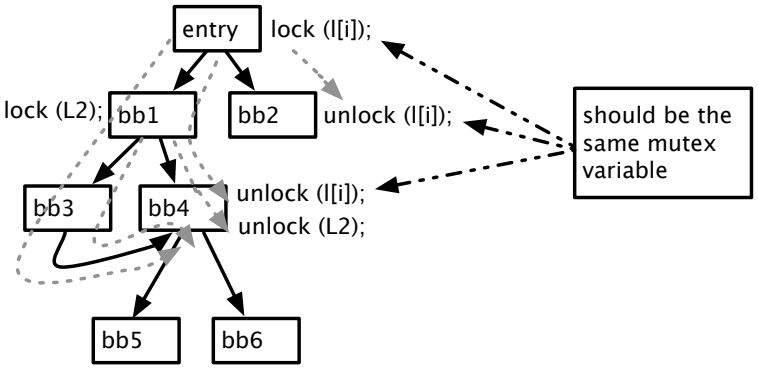


Figure 3.7: Challenges in deriving scope of critical sections.

a program, Serenity gives up on its opportunistic optimization and falls back to conservatively instrumenting the entire program, thus sacrificing performance but not correctness.

If a program is well-scoped, then Serenity instruments all loads and stores in the path from lock acquisition to release traversed in the CFG including across any procedure calls that lie within its scope. In the presence of function pointers, Serenity performs *signature analysis* to identify the set of all possible matches based on the signature (return type, number of arguments and their type information) of a procedure thereby instrumenting them and their call chain. Note that in the presence of function pointers, this approach is conservative and may instrument procedures that may not actually be called within a critical section. We note that any such imprecise instrumentation does not affect program correctness or the determinism of deadlock detection and recovery.

3.2.3 Implementation

In practice, identifying the scope of critical sections poses several challenges. First, we must be able to establish equivalence between lock and unlock objects, i.e., detect that the acquisition and release happen on the same mutual exclusion variable (e.g., lock and unlock on *l[i]* in Figure 3.7) to accurately derive the scope of a critical section. Second, in the presence of branches ($V_{entry} \rightarrow V_{bb1}$ or $V_{entry} \rightarrow V_{bb2}$) as shown in Figure 3.7, the acquisition of lock *L2* depends on how the branch is evaluated, which may not be known precisely at compile time. Consequently, we need to efficiently evaluate all possible control flows since acquisition of a lock. Third, if a critical section

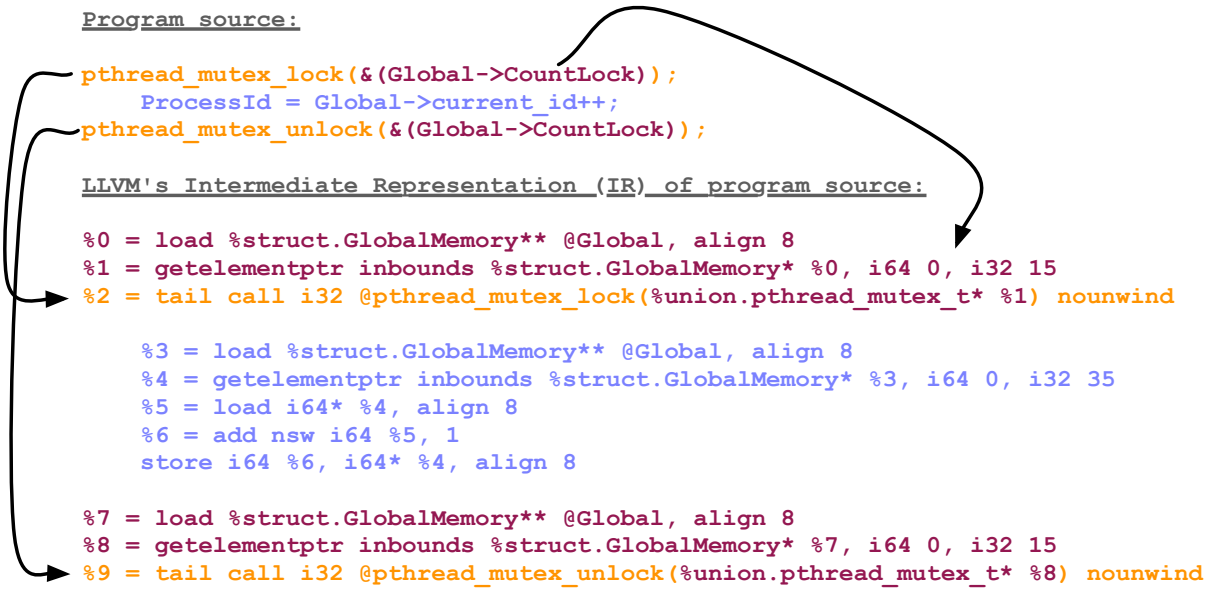


Figure 3.8: Operand Equivalence Algorithm.

is protected by nested locks, then each lock in the sequence of nested locks has its own scope and we should be able to determine the scope of locks even in the presence of nested locks and/or lock inversion. Finally, to determine if a given lock is procedurally-scoped, we need a traversal algorithm. Unfortunately, traversing all possible paths from a given vertex to every exit point (leaf vertex) in a CFG has exponential time complexity.

Operand Equivalence Algorithm

Identifying the equivalence of a mutual exclusion variable in acquisition and release in the presence of pointers, arrays (e.g., *l[i]* in Figure 3.7), derived data types such as structures (e.g., *L2* in Figure 3.7 could be a member variable), and unions is non-trivial. We note that neither LLVM [51] nor any other publicly available compiler infrastructure provides such a capability.

To establish equivalence, Serenity first obtains the operand of the lock acquisition and the operand of the lock release in the LLVM IR. For each of the operands (shown as %1 and %8 in the LLVM IR in Figure 3.8), Serenity recursively back tracks and creates a list of all the instructions in the IR generating the operand, e.g., loads, stores, GEP instructions [51] (used to resolve member variables

of structures, unions, etc.). For example, the lock operand (%1) depends on %1 and %0 in the IR. Likewise, the unlock operand (%8) depends on %8 and %7. Serenity then performs a pair-wise comparison of instructions obtained from acquisition and release. The comparison checks for the instruction type, the nature of the instruction's operation, and the type of the instruction's operands. To ensure safety in establishing equivalence, Serenity performs a conservative and strict comparison, even though some of the comparisons may be relaxed. Additionally, Serenity checks if operands (%1 and %8) are not modified anywhere else in the procedure. If the instruction streams are identical, then Serenity can safely establish the equivalence between the lock acquisition and its corresponding release. If the comparison of instruction streams producing the lock and unlock differ, it is highly likely that the lock and unlock do not refer to the same object. Serenity detects this mismatch and conservatively instruments the entire program. The execution of the program is still correct and deadlock free — Serenity is ultimately conservative. Based on our discussions with the LLVM developers, to our knowledge, Serenity's operand equivalence algorithm is the only approach available for LLVM. We tested our algorithm on several applications, including those described in Section 3.3 and we found no cases of value aliasing of lock objects. This is probably due to the fact that unlike typical program variables, lock objects/variables cannot be copied, since they contain state associated with the operating system kernel and system libraries.

Reachability Analysis and CFG Traversal Algorithm

Given the association of a lock acquisition with its release we now present an efficient vertex traversal transformation that only visits each vertex once on a possible path from the basic block containing the lock acquisition to a leaf vertex.

Serenity starts with a CFG (defined in Section 3.2.2). Serenity initially marks all basic blocks that contain callsite(s) to lock acquisition as candidates and iterates over all candidates in the CFG. For each lock L in a candidate basic block V_{bbi}, Serenity checks if the candidate basic block itself contains a corresponding release of L. If so, the lock is well scoped since both the lock acquisition and release on L are in the same basic block and the lock acquisition precedes its release in V_{bbi}. In

the absence of a release of L within the same basic block V_{bbi}, Serenity checks if V_{bbi} is a leaf vertex. If so, we have an exit point of the procedure with a lock L and without a corresponding release of L. Serenity marks the lock as *not procedurally-scoped* and thereby the program as *not well-scoped* (Recall that all locks in the program should be well-scoped) and conservatively instruments the entire program.

The traversal employs a FIFO queue Q (Q is initialized to $\emptyset$) and starts at a lock acquisition callsite V_{bbi} that is not an exit point (leaf vertex). The algorithm appends each adjacent vertex (V_{bbj}) of V_{bbi} in the CFG to Q iff i) V_{bbj} is not in Q, ii) V_{bbj} is not visited, and (iii) V_{bbj} is not V_{bbi}. The algorithm marks V_{bbi} as visited and dequeues the next entry from the queue, returned in say V_{bbk}. The traversal algorithm then checks if the dequeued entry (V_{bbk}) contains a release of lock L. If it contains a release of L then the algorithm marks V_{bbk} as visited and does not add any of the adjacent vertices of V_{bbk} to Q. If V_{bbk} does not contain a release of L and if it is not a leaf vertex then the algorithm proceeds by enqueueing the adjacent vertices of V_{bbk} to Q as discussed above. If V_{bbk} does not contain a release of L and if it is a leaf vertex then this implies that we could not find a release of L within the same procedure, so lock L is *not procedurally-scoped*. This traversal continues through the FIFO queue until it is empty. *If the queue is empty after the traversal, then all paths from the entry point to all exit points of a procedure have been explored and the lock is procedurally-scoped.*

Intuitively, this algorithm traverses all reachable vertices from a lock acquisition callsite to ensure that corresponding releases precede function exit. It can be trivially shown that the time complexity of this algorithm to check if a given lock L is well scoped is O($|E|$). We reset the visited information of basic blocks in the CFG and repeat this algorithm for all locks in a program, thus effectively determining if a program is well-scoped. Figure 3.7 illustrates the traversal for locks *l[i]* and *L2*. The algorithm traverses the paths $V_{entry} \rightarrow V_{bb2}$, $V_{entry} \rightarrow V_{bb1} \rightarrow V_{bb4}$, $V_{entry} \rightarrow V_{bb1} \rightarrow V_{bb4}$ to identify if lock *l[i]* is procedurally scoped. If we remove the unlock call in *bb2*, the traversal will find the path $V_{entry} \rightarrow V_{bb2}$ to be not procedurally scoped, since we reached V_{bb2} (leaf node) without having encountered a call to unlock.

3.3 Experimental Evaluation

In our evaluation we set out to determine how Serenity performs on real applications. To that end, we applied Serenity to 24 real world lock based applications. Then we compared the overall execution time of the original and instrumented versions of the applications. To evaluate the effectiveness of Serenity we applied it to applications containing deadlocks, and to a set of synthetic benchmarks with artificially seeded deadlocks. Serenity was able to eliminate all the deadlocks, while the rest of the execution continued unperturbed.

3.3.1 Experimental Setup

We performed an experimental evaluation of Serenity on a 2.3 GHz 64 core shared memory (NUMA) machine with 256 GB of RAM and a x86_64 Linux 2.6.32 kernel. We compiled all the evaluation benchmarks using the LLVM-2.9 compiler infrastructure with -03 optimizations. For each benchmark, we compared the unmodified (vanilla) case with performance with Serenity. We verified the output produced by the benchmark under both vanilla and Serenity. We report the average of five runs under each configuration.

To evaluate the performance and scalability of Serenity, we employed the widely used SPLASH [94], Phoenix [80], and PARSEC [12] benchmarks suites. Additionally, we used several commonly used desktop and server applications including Pbzip2 [68], Tgrep [61], Squid [95] (a webcache and proxy server), and Sendmail [66], a general purpose email routing software. The benchmark suites are well studied in the literature and include applications from a wide range of domains with various threading models, locking regimes and lock rates (ratio of total locks acquired to the total runtime of the native application). The lock acquisition rates of some of the applications in this suite of benchmarks are extremely high, e.g., Fluidanimate (120M locks/sec), Barnes (1M locks/sec), FMM (522K locks/sec), Water (217K locks/sec), Facesim (246K locks/sec), and Tgrep (97K locks/sec). We note that such extremely high lock rates stress test Serenity to the extreme. Finally, to evaluate Serenity's ability to recover from deadlocks we applied Serenity to SQLite, HawkNL, and synthetic

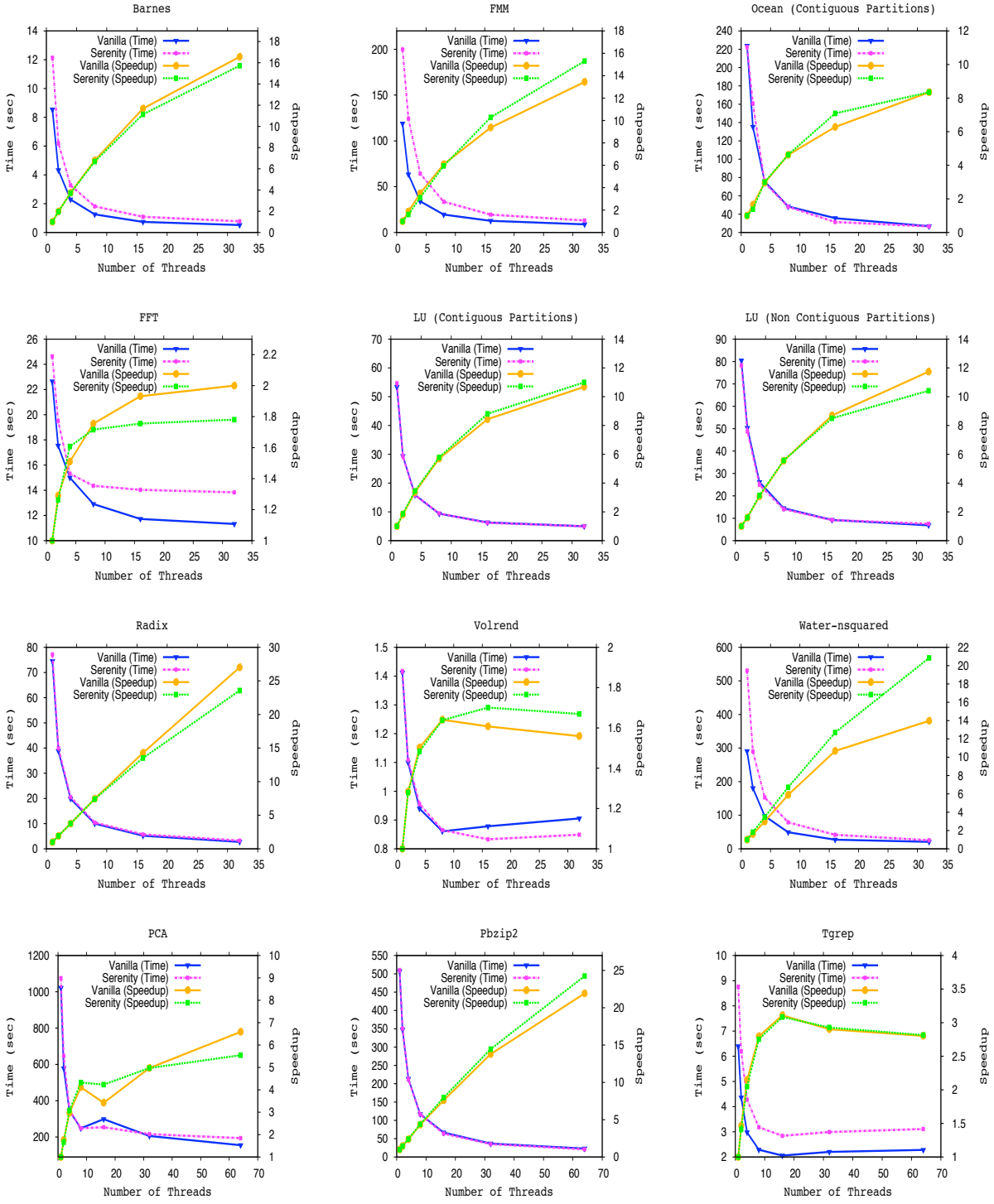


Figure 3.9: Performance of SPLASH benchmarks, Phoenix PCA, Pbzip2, and Tgrep.

Table 3.1: Characteristics of SPLASH applications.

Benchmark	#Locks	#Stores	#Loads	#Lock rate
FFT	3.20E+01	3.20E+01	9.60E+01	2.83E+00
Barnes	5.50E+05	7.28E+06	1.03E+07	1.07E+06
Ocean	5.54E+03	8.77E+02	1.66E+04	2.07E+02
FMM	4.62E+06	1.09E+08	2.54E+08	5.22E+05
LU-CP	3.20E+01	3.20E+01	9.60E+01	6.40E+00
Water	4.54E+06	4.07E+07	2.62E+08	2.18E+05
LU-NCP	3.20E+01	3.20E+01	9.60E+01	4.67E+00
Radix	4.07E+02	2.64E+02	6.60E+02	1.47E+02
Volrend	7.68E+04	7.65E+04	2.32E+05	8.48E+04

programs containing examples of deadlocks taken from existing literature including [75, 9, 44, 108].

For the SPLASH suite we report results from benchmarks that have (a) a runtime of at least a few seconds to avoid statistical noise from scheduler overhead, and (b) applications that compiled on a 64 bit machine. For the Phoenix suite, we present the results of the *PCA* benchmark, since it is the only benchmark that contains locks. In the absence of locks, there is no difference in performance between Serenity and native thread execution. In the PARSEC benchmark suite, we omit *bodytrack* and *ferret* since they use read-write locks. Serenity's program analysis is agnostic to the type of the lock; however, Serenity's runtime needs minor additions (mostly an engineering exercise) to support such locks. We omitted *freqmine* since it does not use Pthreads. The native versions of *raytrace* and *vips* in the PARSEC suite did not compile. We ran SPLASH benchmarks with a maximum of 32 threads, since they do not support more than 32 threads; for the rest of the benchmarks we were able to run with 64 threads.

3.3.2 Performance and Scalability

Figure 3.3.1 compares the overhead and scalability of Serenity with native thread (Pthread) execution for the SPLASH benchmarks. For each benchmark we measured the application characteristics including the total number of locks acquired, loads and stores performed in lock context, lock rate and the total runtime. Table 3.1 summarizes these characteristics for the SPLASH suite. The performance of Serenity is comparable to Pthreads for most applications, with the exception of *Barnes*, *FMM* and *Water*. The overhead of Serenity depends on 1) the number of critical sections

Table 3.2: Characteristics of PARSEC, Phoenix, and desktop applications.

Benchmark	#Locks	#Stores	#Loads	#Lock rate
Canneal	6.40E+01	8.01E+04	8.01E+04	7.30E-01
Dedup	1.04E+06	4.60E+08	1.34E+09	2.68E+04
Streamcluster	1.08E+04	1.68E+02	1.06E+04	4.22E+01
Facesim	2.35E+07	1.53E+08	3.59E+08	2.46E+05
Fluidanimate	3.75E+09	7.40E+09	1.50E+10	1.20E+08
x264	2.03E+05	0.00E+00	0.00E+00	2.01E+04
PCA	1.01E+04	1.01E+04	1.01E+04	6.49E+01
Pbzip2	1.07E+03	2.03E+03	4.22E+03	4.62E+01
Tgrep	2.24E+05	5.59E+05	8.94E+05	9.80E+04

and 2) the data accessed/modified within a critical section, which influences the runtime cost of instrumentation and shadowing.

Barnes, FMM, and Water acquire a large number of locks and access non-trivial amounts of data in critical sections. For instance, Barnes acquires over 550K locks and performs a total of 10.3M loads and 7.2M stores. FMM acquires over 4.6M locks and performs a total of 254M loads and 108M stores. Water acquires approximately 4.5M locks, and performs 254M loads and 40M stores. Even under such lock and data access rates, we note that Serenity's performance overhead is comparatively modest. Serenity's scalability with the number of threads is almost identical to the vanilla scalability, which is a significant improvement over the scalability of comparable STM systems [18]. As an aside, we note that such high lock rates are more common in HPC applications, which are memory and compute bound. Enterprise applications are largely I/O bound, which reduces their lock rates significantly.

In Figure 3.3.2 we present the results of the PARSEC benchmark suite and summarize the application characteristics in Table 3.2. Dedup and Fluidanimate incur significant overhead. Dedup acquires approximately 1M locks and performs 1.3B loads and 460M stores. Fluidanimate acquires 3.7B locks and performs 14.9B loads and 7.3B stores. In Figure 3.3.1 we present the results of the Phoenix PCA benchmark along with Pbzip2 and Tgrep. Serenity achieves performance almost identical to native thread execution on these benchmarks due to their relatively low lock rates and low memory updates performed within critical sections.

For Squid [95] and Sendmail [66] we are unaware of any publicly available test suite. We set up

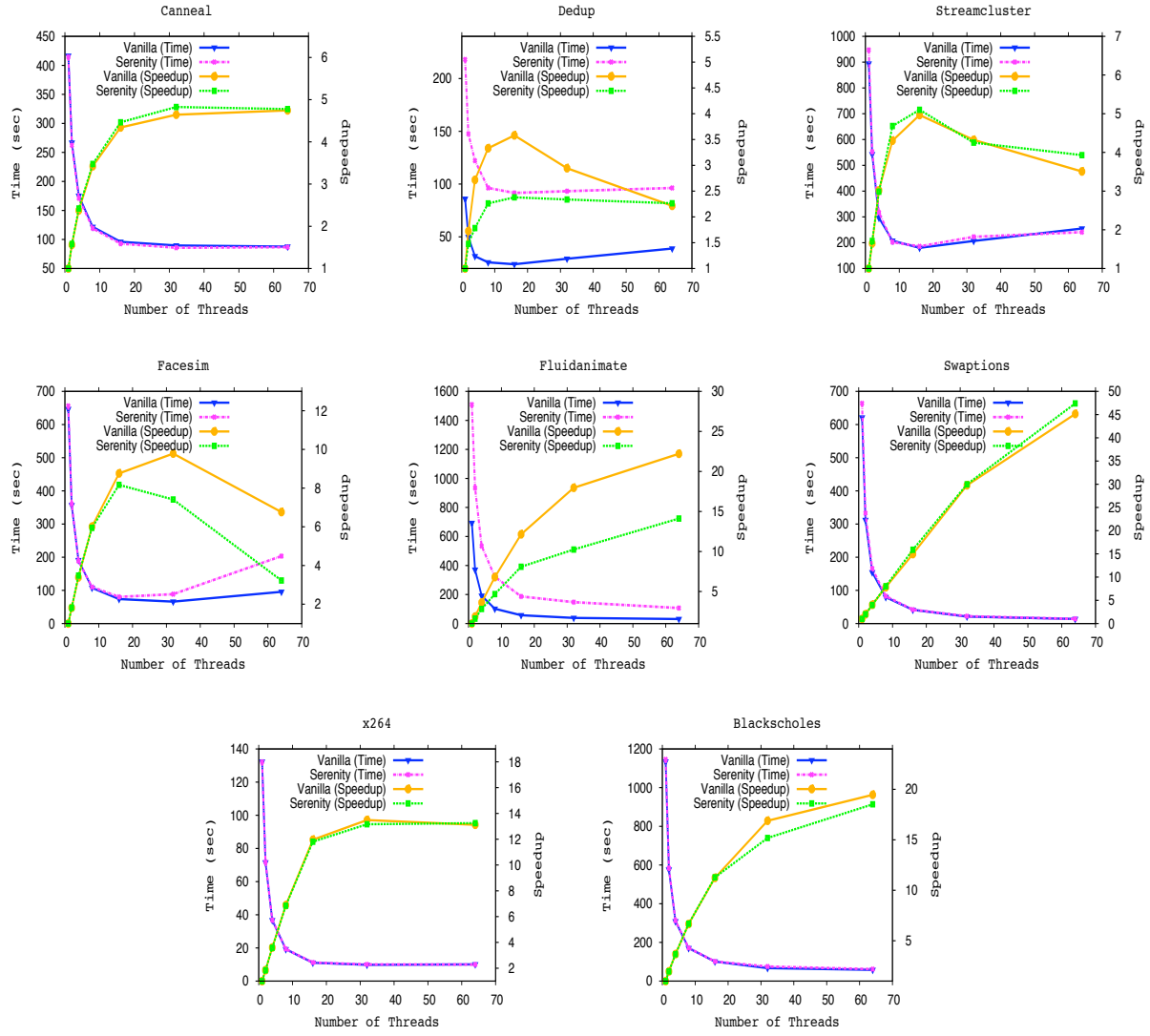


Figure 3.10: Performance of PARSEC benchmarks.

the Squid proxy server to run on the 64 core NUMA machine with the default Squid configuration (cache size, directories, etc.), used the firefox browser and wget to generate internet traffic, and verified the log files to validate the generated traffic. The performance of Serenity is identical to the original program; the log files produced in each experiment were also identical to each other. Sendmail includes a library called Milter (sendmail Mail Filter API) to enable development of custom email filters. We used the test email filter programs that come with Sendmail's distribution (libmiter) and found performance of Serenity to be identical to native execution.

3.3.3 Deadlock Detection and Recovery

Finally, we evaluated Serenity using applications with known deadlocks. We used a test harnesses from Dimmunix [45] to reproduce the deadlock in SQLite-3.3.3 [24] and HawkNL. Additionally, we used a synthetic benchmark suite (discussed in Section 2.3.3) that implements deadlock cycles of varying diameter, involving a configurable number of threads. While the native Pthreads programs deadlocks on each of these benchmarks, Serenity successfully detects and recovers from the deadlocks, executing the program to completion.

3.3.4 Memory Overhead

Serenity's memory overhead stems from two components — metadata and shadowing. Serenity's metadata is relatively small at approximately (1-2 MB) per thread. Shadowing overhead is incurred when the application is in a critical section. This overhead is proportional (2x) to the amount of data modified within a critical section. First, Serenity maintains a twin of size 8 bytes to the 64 bit aligned address, which is used to compute the exclusive-or difference of the bytes modified by a thread (Section 3.1.3). Second, Serenity creates a shadow of length 8 bytes of the 64 bit aligned address to provide isolation of updates within a critical section. We note that both these allocations are freed at the end of a critical section and this overhead is transient.

Sammati employs privatization at the granularity of a page (4K) while Serenity employs shadowing

Table 3.3: Effectiveness of lock scope analysis across all the applications used in this study.

	Effectiveness of Instrumentation		
Instruction Type	Min	Max	Median
Load	2.01×10^{-07}	1.00×10^{02}	2.48×10^{-04}
Store	1.39×10^{-07}	1.00×10^{02}	4.25×10^{-03}

at 8 byte granularity. Consequently, Sammati incurs significantly more memory overhead than Serenity.

3.3.5 Efficiency of Lock Scope Analysis

To measure the effectiveness of our lock scope analysis (discussed in Section 3.2.2) we computed the percentage of loads performed within a critical section relative to the total number of loads performed by the application. This indicates the percentage of the total load instructions that were actually instrumented — the effectiveness of lock scope analysis. The lower the percentage, the higher the effectiveness of serenity's lock scope analysis. We also measured these statistics for the store instructions. Table 3.3 summarizes these characteristics. Pbzip2 and Tgrep were not well-scoped and consequently Serenity instrumented them completely. Hence we find the maximum to be 100% in Table 3.3. For over 80% of the applications, Serenity instrumented less than 0.54% of the loads and less than 2.2% of the stores. The mean percentage for loads and stores for well scoped programs is 0.34% and 1.73%, respectively. These results indicate that Serenity's lock scope analysis is effective and efficient.

3.4 Limitations

Serenity comes with several limitations and we discuss them in detail in this section.

3.4.1 Recompilation

To successfully recover from a deadlock Serenity requires all critical sections in the program to be instrumented, including any external application dependencies such as shared libraries. Consequently, it may require recompilation of all libraries that a program depends upon. Most shared libraries can be directly instrumented and we are working on splitting *libc* into user-level only functions (such as *memcpy*, *memcmp*, string and math functions) and functions that invoke system calls that require more comprehensive isolation.

3.4.2 Unsupported Applications

Serenity supports the POSIX interface by default. However, it does not support applications that employ ad-hoc synchronization without minor modifications to the source code. In practice, codes that employ ad-hoc synchronization either in the presence of critical sections or otherwise are not safe and such a programming practice could result in several concurrency bugs [108]. Furthermore, such codes assume and rely on certain memory consistency guarantees from the underlying architecture to propagate the updates, In practice, these assumptions may not hold, resulting in incorrect program behavior. It is important to understand that providing support for mixed locking regimes (i.e., employing ad-hoc synchronization within critical sections) conflicts with any runtime system that supports transparent (containment through privatization without modifying source code) deadlock recovery. Consequently, Serenity is incapable of providing guaranteed transparent deadlock recovery for such codes. Rather than using ad-hoc synchronization schemes such as these, we recommend the use of standard POSIX primitives for synchronization. Alternatively, in the event that the ad-hoc synchronization is an absolute necessity, the programmer can provide Serenity with a wrapper around the ad-hoc synchronization in the form of a user-defined lock.

3.4.3 Programmer Input

Serenity does not require the programmer to modify any source code and Serenity by default supports POSIX mutual exclusion locks. However, if the lock/unlock operations are encapsulated in functions/classes, the programmer must explicitly provide this information (as a part of a config file) to Serenity's compile time analysis. This does not require any understanding of the application and we found that using *cscope* we were able to perform this step within a couple of minutes.

3.4.4 Deadlock Recovery

Although Serenity can deterministically detect all deadlocks, it cannot recover from certain deadlocks if they involve unrolling non-idempotent irrevocable actions performed within critical sections. Handling arbitrary non-idempotent I/O within the context of a restartable critical section is an open problem.

3.5 Related Work

We briefly discuss related work on deadlock detection and recovery to place Serenity in the context of existing literature.

3.5.1 Deadlock Detection and Recovery

Pure static analysis techniques (e.g., RacerX [29], Lock Lint [97]) can identify certain types of deadlocks at compile time. Unfortunately, they cannot identify all deadlocks in type-unsafe languages, rendering them susceptible to false positives. Predictive dynamic analysis tools such as Visual Threads [34], DeadlockFuzzer [44], CheckMate [42], and others [7, 3] collect program traces and rely on techniques such as model checkers, state machines, and feasible permutation generation to explore possible thread interleavings. Deadlock detection in such systems is probabilistic, potentially resulting in false positives and false negatives.

Deadlock avoidance techniques [105, 31] employ program analysis to collect the order of lock acquisitions and releases in a program and attempt to avoid deadlocks by delaying lock acquisitions. Gadara [105] relies on discrete control theory to identify potential deadlocks at runtime; it requires program annotations by the programmer, and is susceptible to false positives. The approach of Gerakios et al. [31] grants a lock only when both the requested lock and its future lockset are available.

Tools such as Dimmunix [45] attempt to build resistance to deadlocks in multithreaded programs. Dimmunix maintains the state information of each deadlock pattern that occurs at runtime and aims to prevent future occurrences through deadlock prediction. Pulse [52] employs speculative execution to detect deadlocks; it requires modifications to the operating system and runs as a system daemon. Pulse speculatively executes processes and attempts to construct a resource graph to detect deadlocks. Dimmunix and Pulse do not provide any support for recovery in the event of a deadlock. Grace [9] eliminates a wide range of concurrency bugs. Grace employs lock elision, sequential composition, and speculative execution of threads. Unfortunately, Grace only supports applications that employ fork-join parallelism.

Tools such as Sammati [75] and Rx [77] are capable of detecting and recovering from deadlocks. Sammati employs a pure runtime approach to detect and recover from deadlocks. Sammati incurs significant runtime overhead and does not scale well with applications that have a high lock acquisition rate ($> 800K$ locks/sec). Additionally, Sammati cannot recover thread-local data (TLS), disk I/O, and other memory side-effects in the event of a deadlock. Rx [77] can detect and recover from a wide range of bugs through checkpointing. Upon a software failure, it rolls back the program to the most recent (in program order) checkpoint and re-executes the program under a new environment, i.e., the original environment perturbed by artificially introducing noise. Rx requires modifications to both the kernel and the application.

In contrast to such systems, Serenity does not collect program traces, and it deterministically detects and eliminates deadlocks at runtime without any false positives. Serenity's use does not require any modifications to the operating system, compiler, or the program source code. Additionally, it

is capable of supporting a wide range of applications and not necessarily those limited to fork-join parallelism. Serenity does not rely on address space protection or page level privatization, and it is capable of supporting applications that acquire billions of locks at rates of hundreds of millions of locks/sec. Serenity performs efficient deadlock recovery without requiring a complete application checkpoint and its associated overhead. Finally, Serenity provides a more comprehensive and broadly applicable deadlock elimination scheme than systems such as Sammati [75], Rx [77], and Grace [9].

Serenity may be viewed as transactional memory (TM) without optimistic concurrency. The relationship to TM systems is discussed in Section 2.5.2.

3.6 Explicit Locking vs Transactions

The non-composability of lock based code has been the primary motivation for transactional memory systems [36, 91]. Transactional memory model delineates un-named regions via transactions and employs compile time analysis and runtime techniques to infer serialization. Similar to lock based codes, transactions provide mutual exclusion but unlike lock based codes, transactions can be optimistically executed concurrently, supposedly leading to improved performance. TM starts with an approach based on concurrency, while a lock based model fundamentally defines serialization.

There are several merits to explicit locking and transactions but so far transactions are yet to prove their worth. The interactions between transactions and non-transactional code is still ill-defined. Blundell et al., [23] introduced the notion of *weak* and *strong* atomicity to define the memory semantics of interactions between transactions and non transactional code and show that such interaction can lead to data races, program errors or even deadlocks. Defining the semantics of the memory model and the interaction between transactional and non transactional code is an on going area of research [37, 92, 93]. Most TM systems require language support with special programming language constructs [33, 65, 109] or API [87] to provide TM semantics. Alternatively, some TMs rely on special memory allocation primitives [37] and wrappers [65] to support transactional memory

semantics.

In contrast, the vast majority of existing large-scale code bases use the relatively well-understood (warts and all!) lock based model [59]. Hence addressing issues with the dominant shared memory programming model i.e., the POSIX threads (Pthreads) model has a broader impact. It is important to note even though there are several parallels between explicitly transactional software TM systems and systems such as Sammati and Serenity. Unlike TM systems, both Sammati (discussed in chapter 2) and Serenity (discussed in chapter 3) support applications written using lock based model. Furthermore, neither Sammati nor Serenity require any modifications to application source code, compiler, and the operating system.

3.7 Future Work

Serenity's compile time analysis can be extended to include inter-procedural context sensitive control flow analysis, and value aliasing of lock variables. Serenity's runtime can be improved by providing a comprehensive roll back support for non-idempotent operations performed within critical sections. Serenity can be extended to include support for POSIX read-write locks. Finally, Serenity can be extended to include API to detect order violations.

3.8 Summary

We presented Serenity, a software system that transparently eliminates deadlocks in applications written in type-unsafe languages such as C and C++. Our experimental results indicate that Serenity achieves its core goal of deadlock elimination at a performance level that enables widespread adoption. Serenity incurs a transient memory overhead proportional (2x) to the amount of data modified within a critical section. We believe that by providing usable and efficient deadlock detection and recovery for threaded codes, we provide a critical tool to programmers designing, implementing, and debugging complex applications for emerging many-core platforms. More broadly,

this research will assist in improving the productivity of application developers.

Chapter 4

Coarse-Grain Speculative Parallelism

In this chapter we present Anumita (*guess* in Sanskrit) [74, 72, 73] a framework for exploiting coarse-grain speculative parallelism in hard-to-parallelize applications whose performance is highly dependent on input data. Anumita provides programming constructs for C, C++, Fortran, and OpenMP and a supporting runtime system that abstracts the subtleties of concurrent programming and relieves the programmer from the complexity of creating, managing and retiring speculations. Additionally, Anumita provides transparent name-space isolation. Speculations in Anumita may be composed by specifying surrogate code blocks at any arbitrary granularity, which are then executed concurrently, with a single winner ultimately modifying program state. Anumita provides expressive semantics for winner selection that go beyond *time to solution* to include user-defined notions of *quality of solution*. Performance results from several applications show the efficacy of using coarse-grain speculation to achieve (a) robustness when surrogates fail and (b) significant speedup over static algorithm choices.

The rest of the chapter is organized as follows. Section 4.1 outlines the motivation for this work. Section 4.2 presents the programming model and constructs used to express coarse-grain speculative execution. Section 4.3 presents how these constructs can be implemented efficiently without sacrificing performance, portability and usability. Section 4.4 presents our experimental evaluation. Section 4.5 surveys the related work. Section 4.6 describes future directions and Section 4.7

presents our conclusions.

4.1 Motivating Problems

Coarse-grain speculative parallelism is most useful for applications with two common characteristics: (1) there exist multiple possible surrogates (e.g., code blocks, methods, algorithms, algorithmic variations) for a particular computation, and (2) the performance (or even success) of these surrogates is problem dependent, i.e., relative performance can vary widely from problem to problem, and is not known a priori. Whether or not there exist efficient parallel implementations of each surrogate is an orthogonal issue to the use of coarse-grain speculation. If only sequential implementations exist, speculation provides a degree of useful parallelism that is not otherwise available. If parallel surrogate implementations do exist, speculation still provides resilience to hard-to-predict performance problems or failures, while also providing an additional level of parallelism to take advantage of growing core counts, e.g., by assigning a subset of cores to each surrogate rather than trying to scale a single surrogate across all cores.

We discuss two motivating examples in detail. (Performance results for these problems are given in Section 4.4).

4.1.1 Graph Coloring Problem

In graph theory, vertex coloring is the problem of finding the smallest set of colors needed to color a graph $G = (V, E)$ such that no two vertices $v_i, v_j \in V$ with the same color share an edge e. Graph coloring problems arise in several domains including job scheduling, bandwidth allocation, pattern matching and compiler optimization (register allocation). Several state-of-the-art approaches that solve this problem employ probabilistic and meta-heuristic techniques, e.g., simulated annealing, tabu search and variable neighborhood search. Typically, such algorithms initialize the graph with a random set of colors and then employ a heuristic algorithm to attempt to color the graph using the specified number of colors. Depending on the input graph, the performance of these techniques

varies widely. Obviously, there will be cases where no coloring can be found (when the specified number of colors is too small) by some or all methods. In addition to this sensitivity to the input, algorithms for the graph coloring problem are hard to parallelize due to inherent data dependancies. Parallel implementations that exist employ a divide and conquer strategy by dividing the graph into subgraphs and applying coloring techniques on the subgraphs in parallel. During reduction, conflicting subgraphs are recolored. Despite such efforts, the challenge still persists to develop efficient parallel algorithms for vertex coloring.

4.1.2 Partial Differential Equations (PDEs)

As a second example, consider the numerical solution of partial differential equations (PDEs). This is one of the most common computations in high performance computing and is a dominant component of large scale simulations arising in computational science and engineering applications such as fluid dynamics, weather and climate modeling, structural analysis, and computational geosciences.

The large, sparse linear systems of algebraic equations that result from PDE discretizations are usually solved using preconditioned iterative methods such as Krylov solvers [86]. Choosing the right combination of Krylov solver and preconditioner, and setting the parameter values that define the details of those preconditioned solvers, is a challenge. The theoretical convergence behavior of preconditioned Krylov solvers on model problems is well understood. However, for general problems the choice of Krylov solver, preconditioner, and parameter settings is often made in an ad hoc manner. Consequently, iterative solver performance can vary widely from problem to problem, even for a sequence of problems that may be related in some way, e.g., problems corresponding to discrete time steps in a time-dependent simulation. In the worst case, a particular iterative solver may fail to converge, in which case another method must be tried. The most conservative choice is to abandon iterative methods completely and simply use a direct factorization, i.e., some variant of Gaussian Elimination (GE). Suitably implemented, GE is essentially guaranteed to work, but in most cases it takes considerably longer than the best preconditioned iterative method. The problem

is that the best iterative method is not known a priori.

4.1.3 Combinatorial Problems

One could list many other examples that are good candidates for coarse-grain speculation. Similar analysis can be extended to other widely used combinatorial problems including sorting, searching, permutations and partitions. Even for a simple problem such as sorting, where theoretical algorithmic bounds are well known, in practice the runtime of an algorithm depends on a variety of factors including the amount of input data (algorithmic bounds assume asymptotic behavior), the sortedness of the input data, and cache locality of the implementation [5].

4.2 Speculation Programming Model

For a coarse-grain speculation model to be successful, it should satisfy several usability and deployability constraints. First, the model should be easy to use, with primarily sequential semantics, i.e., the programmer should not have to worry about the complexities and subtleties of concurrent programming. Speculation is not supported by widely used languages or runtime systems today. Hence, in order to express speculation, the programmer is burdened with creating and managing speculation flows using low-level thread primitives [70]. Second, the speculation model should enable existing applications (both sequential and parallel) to be easily extended to exploit speculation. This includes support for existing imperative languages, including popular type-unsafe languages such as C and C++. Third, the model should be expressive enough to capture a wide variety of speculation scenarios. Finally, to ensure portability across platforms, the speculation model should not require changes to the operating system. Furthermore, we need to accomplish these objectives without negatively impacting the performance of applications that exploit speculation.

A general use case for Anumita is illustrated in Figure 4.1. The example shows an application with three threads, two of which enter a *speculative region*. (The simplest case would involve a single-threaded code that enters a single speculative region.) Each sequential thread begins

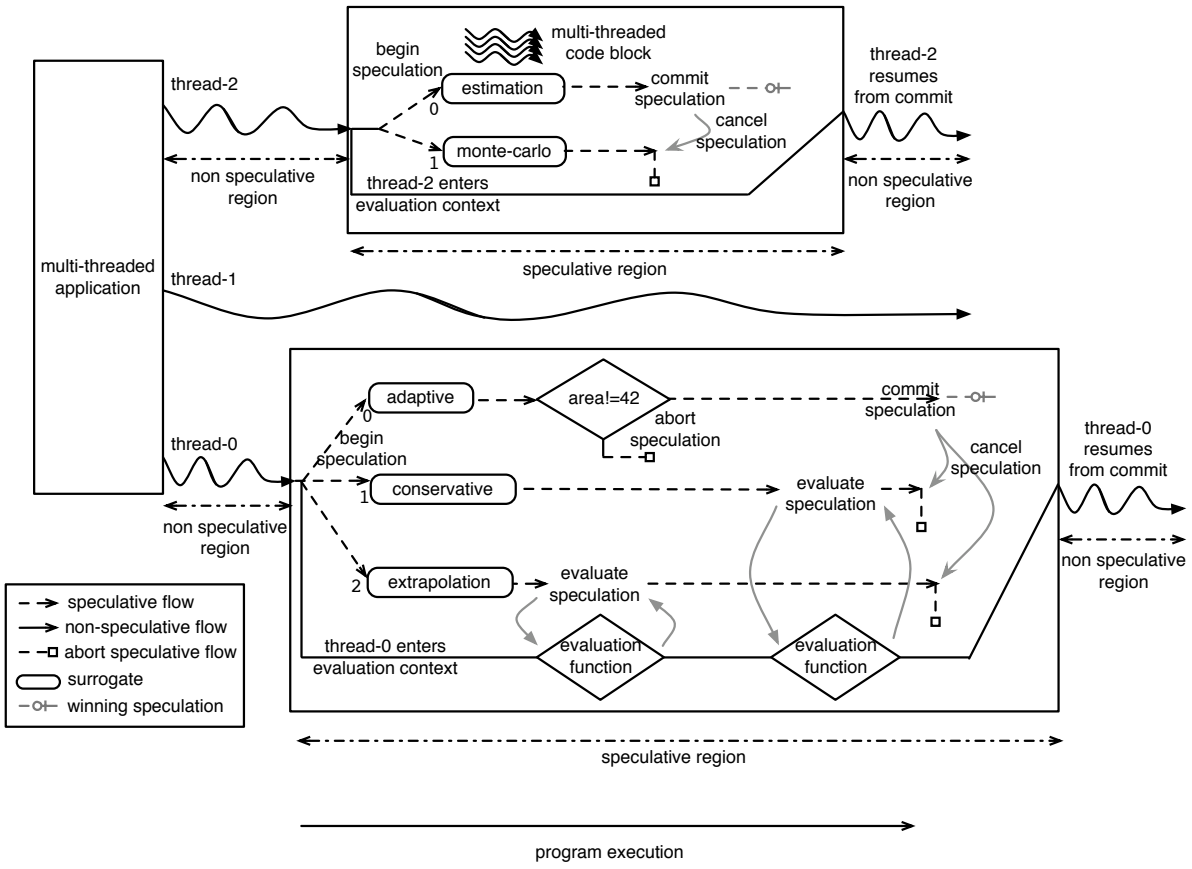


Figure 4.1: A typical use case scenario for composing coarse-grain speculations. Anumita supports both sequential and multi-threaded applications.

execution non-speculatively until a speculative region is encountered, at which time n speculative control flows are instantiated, where n is programmer-specified. Each flow executes a different surrogate code block. We refer to this construct as a "*concurrent continuation*," where one control flow enters a speculative region through an API call and n speculative control flows emerge from the call. Anumita achieves parallelism by executing the n speculative flows concurrently. In Figure 4.1, the concurrent continuation out of thread 0 is a composition of three surrogates, while the continuation out of thread 2 has two surrogates. Note that individual surrogates may themselves be multithreaded, e.g., surrogate *estimation* in the continuation flowing out of thread 2. Although not shown in the figure, Anumita also supports nested speculation, where a speculative flow in turn creates a speculative composition.

To mitigate the impact of introducing speculation into the already complex world of concurrent programming, no additional explicit locking is introduced by the speculation model. In other words, a programmer using the Anumita API to add speculation to a single-threaded application does not have to worry about locking or synchronization of any kind. Of course, if the original application was already multithreaded, then locking mechanisms may already be in place, e.g., to synchronize among the three threads in Figure 4.1 in non-speculative regions.

Each speculative flow operates in a context that is isolated from all other speculations, thereby ensuring the safety of concurrent write operations. Anumita presents a shared memory model, where each speculative flow is exactly identical to its parent flow in that it shares the same view (albeit write-isolated) of memory, i.e., global variables, heap and more importantly, the stack.

The Anumita programming model provides a flexible mechanism for identifying the winner and committing the results of a speculation. The first flow to successfully commit its results is referred to as the winning speculation. However, the decision to commit can be made in a variety of ways. The model easily supports the simplest case, where the first flow to achieve some programmer-defined goal cancels the remaining speculative flows and committs its updates to the parent flow, which resumes execution at the point of commit. Surrogate *estimation* illustrates this case in Figure 4.1. Alternately, speculative flows may choose to abort themselves if they internally detect a failure

mode of some kind, e.g., surrogate *adaptive* in the figure, when `area != 42`. More generally, each surrogate may define success in terms of an arbitrary user-defined evaluation function, passed to an evaluation interface supplied by the parent flow (labeled "evaluation context" in Figure 4.1). The evaluation context safely maintains state that it can use to steer the composition, deciding which surrogates should continue and which should terminate. In our example, surrogates *conservative* and *extrapolation* use the evaluation interface to communicate with their parent flow.

4.2.1 Program Correctness

Any concurrent programming model needs well-defined semantics for propagation of memory updates. Anumita supports concurrency at three levels: (1) between surrogates in a speculative composition, (2) between threads in a single multithreaded surrogate, and (3) between threads in non-speculative regions of an existing multithreaded application. We consider each in turn.

Unlike the traditional threads model, where any conflicting accesses to shared memory must be properly synchronized, Anumita avoids synchronization and its associated complexity by providing isolation among speculative flows through privatization of the shared address space (global data and heap). Furthermore, a copy of the stack frame of the parent flow is passed to each speculative flow. Since updates are isolated, "conflicting" accesses do not require synchronization. Anumita's commit construct implements a relatively straightforward propagation rule: *for a given composition, only the updates of a single winning speculative flow are made visible to its parent flow at the completion of a composition.* Furthermore, compositions within a single control flow are serialized, in that a control flow cannot start a speculative composition without completing prior compositions in program order. Cumulatively, these two properties are sufficient to ensure program correctness in sequential applications (a single control flow) even in the presence of nested speculations. We do not present a formal proof of correctness here; however, the rationale behind the proof is that since the updates of exactly one of the valid outcomes is committed and since each speculation was isolated while arriving at this result, relaxing the requirements of explicit synchronization does not affect program correctness.

```
speculation_t *spec_context;
int num_specs = 2, rank, value = 0;

/* initialize speculation context */
spec_context = init_speculation();

/* begin speculative context */
begin_speculation (spec_context, num_specs, 0);

/* get rank for a speculation */
rank = get_rank (spec_context);

switch (rank)
{
  case 0:
    estimation(...);
    break;

  case 1:
    monte-carlo(...);
    break;

   default:
   printf ("invalid rank\n");
   break;
}

/* commit the speculative composition */
commit_speculation (spec_context);
```

Figure 4.2: Pseudo code for composing speculations using the programming constructs exposed by Anumita. In the absence of an evaluation function, the fastest surrogate (by time to solution) wins.

```
/* custom evaluation function */
boolean goodness_of_fit (speculation_t *spec_context, void *ptr)
{
  double error = 0.0, *fit = (double *) ptr;

  error = actual - *fit;
  if (error > 0.0005)
   {
     return ABORT;
   }

   return CONTINUE;
}

....
switch (rank)
{
  case 0:
    area = adaptive_quadrature (...);

    ptr = get_ir_memory (spec_context);
    memcpy (ptr, area, sizeof(double));

    retval = evaluate_speculation (spec_context, goodness_of_fit, ptr);
    if (retval == ABORT)
      abort_speculation (spec_context);
    break;

  case 1:
    area = conservative_method();
    if (area != 42)
      cancel_speculation (spec_context, 0);
    break;

  case 2:
    for (t=0; t<100; t++)
     {
       area = extrapolation_method();

       ptr = get_ir_memory (spec_context);
       memcpy (ptr, area, sizeof(double));

       retval = evaluate_speculation (spec_context, goodness_of_fit, ptr);
       if (retval == ABORT)
         abort_speculation (spec_context);
     }
   break;
}
....
```

Figure 4.3: Pseudo code for evaluating speculations in Anumita.

Surrogates in Anumita may themselves be multithreaded, requiring lock based concurrency control between threads, e.g., surrogate *estimation* in Figure 4.1. Since surrogates are replacements for each other, we would not expect a surrogate to have synchronization dependencies with one of its sibling surrogate, e.g., *estimation* with *monte-carlo* in the figure. Hence the correctness of multithreaded surrogates reduces to the standard case of threaded shared-memory concurrent programming.

While the above properties ensure program correctness for concurrent continuations flowing out of a single control flow, we also need to define how speculative flows can be composed in a multithreaded environment. Anumita allows multiple speculative and non-speculative regions to execute concurrently, e.g., the regions associated with threads 0 and 2, along with thread 1 in Figure 4.1. However, the model does not support synchronization between speculative flows from different speculative regions, or between speculative flows and other non-speculative application threads. (We note that this restriction is also true for value speculation systems such as [70].) Hence, correctness of such codes stems from the correctness of the original multithreaded code, since each speculative region exhibits transaction-like semantics with respect to other threads, i.e., no memory updates from a given speculative region are visible to other threads until the commit process finishes, at which point all the updates are complete, and control resumes in a non-speculative region.

Externally visible I/O actions are not handled by Anumita in its current form. We are working on extending Anumita to support disk I/O. However, Anumita performs speculation aware memory management and garbage collection from failed speculations. This mechanism correctly hides the side-effects of system calls such as sbrk, etc.

4.2.2 Syntax and Semantics

Figures 4.2 and 4.3 show pseudocode corresponding to the scenario illustrated in Figure 4.1 for composing speculations using Anumita. Table 4.1 defines the Anumita API. A speculative composition is *initialized* by a call to `init_speculation`, which returns a speculation context. A composition is *instantiated* by a call to `begin_speculation`, which implements a concurrent continuation of the parent flow. Each speculative flow in the concurrent continuation is exactly identical to its parent

Table 4.1: Programming constructs exposed by Anumita for leveraging speculation. For brevity, C++ and Fortran interfaces are omitted.

Programming Constructs	Description
speculation_t* **init_speculation**(void)	Initialize a speculative composition.
int **begin_speculation**(speculation_t *spec_context, int num_spec, size_t mem)	Begins a speculative composition. Arguments are composition context, number of speculations and size of memory to allocate for storing intermediate results used in evaluating speculations.
int **get_rank**(speculation_t *spec_context)	Gets the rank of the calling speculative flow.
int **get_size**(speculation_t *spec_context)	Gets the number of speculative flows in a composition.
int **commit_speculation**(speculation_t *spec_context)	Attempts to commit the state of the calling speculative flow.
int **abort_speculation**(speculation_t *spec_context)	Aborts the calling speculative flow.
int **cancel_speculation**(speculation_t *spec_context, int rank)	Cancels (terminates) the speculation flow with a rank of 'rank'.
int **evaluate_speculation**(speculation_t *spec_context, evaluate_t *evaluation_fn, void *ptr)	Used to invoke an evaluation of the intermediate results of the calling speculative flow. Intermediate results are passed through 'ptr'.
void * **get_ir_memory**(speculation_t *spec_context)	Returns a pointer to the calling speculative flow's memory region used to store intermediate results for evaluation.
int (*evaluation_fn)(speculation_t *spec_context, void *ptr)	Signature of the user defined evaluation function.

flow in that it shares the same view of memory, but is isolated from other concurrent speculative flows. In order to distinguish speculative flows from each other, we associate each speculative flow with a unique rank. This notion of rank is identical to ranks in MPI and thread number in OpenMP. A speculation may query its rank (0 to n-1) in order to map a particular unit of work to itself. The parent flow then enters an evaluation context, where it waits (de-scheduled) for evaluation requests from its speculative flows.

To implement an interface for evaluation, the call to `begin_speculation` takes an argument that specifies the size of a memory region that is used for communication between speculative flows and the parent evaluation context. Each speculative flow receives a distinct memory region of the specified size; this region is shared between a speculative flow and the parent evaluation context. Periodically, a speculative flow can request an evaluation using the `evaluate_speculation` call, passing the parent intermediate results using the shared memory region. This call synchronously transfers control to the evaluation context (i.e., the idled parent flow), which executes the evaluation function and returns a status indicating whether the speculation calling the evaluation should

```
int num_specs = 3, rank;

/* begin a speculation composition */
# pragma speculate (spec_context, num_specs, 0)
{

 /* get rank for a speculation */
 rank = omp_get_thread_num();

 switch (rank)
 {
    case 0:
      /* code for speculation */
      vns (colors, graph);
      break;

    case 1:
      /* code for speculation */
      sa (colors, graph);
      break;

    case 2:
      /* code for speculation */
      tabu (colors, graph);
      break;

    default:
      /* invalid rank */
 }

 /* implicit commit of speculation composition */
}
```

Figure 4.4: Composing speculations in OpenMP using the OpenMP extensions built on top of the programming constructs exposed by Anumita. Anumita's source-source translator expands the *speculate* pragma to begin-commit constructs.

continue or abort execution. The evaluation context may also use the intermediate results to cancel other speculations based on the results of the current evaluation, for instance, when the progress of one surrogate is significantly better than another within the same composition. In essence, the evaluation mechanism enables pruning of surrogates based on a user-defined notion of result quality.

On completing execution, a surrogate terminates the speculative region by calling `commit_speculation`. The first call to `commit_speculation` succeeds, canceling all other speculations in the composition and propagating its execution context to the parent flow, which then resumes execution at the

point of commit. Selecting by time to solution (fastest surrogate wins) is trivially implemented by not specifying an evaluation function, as shown in Figure 4.2. In this case the first surrogate to commit would succeed and cancel its siblings. For completeness, the API also supports an `abort_speculation` call that can be used by a surrogate to terminate itself if it detects that it is not making progress or has reached some failure mode. We also provide a `cancel_speculation` call that can be used by any surrogate to terminate any other surrogate. This can be useful, for example, in a case where a subset of surrogates can be pruned from the composition when one member of that subset meets some condition.

Many scientific applications use OpenMP directives for shared memory programming rather than the underlying POSIX threads interface. To support such applications, we provide extensions to OpenMP in the form of a new OpenMP pragma that provides a natural interface to speculation. Figure 4.4 illustrates the OpenMP syntax for creating a composition. The *speculate* pragma is scoped between an open and close brace ({ and }), with an implicit `commit_speculation` at the end of the speculate pragma. In traditional OpenMP programming, name space isolation is achieved through explicit variable scoping (e.g., private, shared, etc.). To simplify programming, the Anumita runtime automatically isolates speculative flows without requiring explicit private scoping.

4.2.3 Overhead

Anumita achieves low runtime overhead since speculative flows are isolated and mispredictions cause the memory updates of the failed speculation to be discarded as opposed to rollback recovery. The memory overhead is proportional to the write-set of all the speculative flows, which is typically much smaller than the read set. Given N speculative flows with the write-set of each flow being W pages, the memory overhead is $\mathrm{O}(NW)$.

4.3 Implementation

The Anumita implementation consists of a shared library that exposes our API and a runtime system. The OpenMP interfaces are implemented using source-to-source translation. To ensure ease of deployment, Anumita is implemented completely in user-space with no modifications to the operating system. The rest of this section describes the Anumita runtime in detail.

4.3.1 Shared Address Space

In the POSIX threads model, each thread has a distinct stack and threads of a process share their address space. In contrast, distinct processes are fully isolated from each other and execute in separate virtual address spaces. Neither of these models satisfies the isolation and selective state sharing requirements imposed by Anumita. Intuitively, we need an execution model that *provides the ability to selectively share state between execution contexts.*

To create the notion of a shared address space among processes, we implemented the *cords* abstraction first proposed in Sammati [75] (The cords abstraction was introduced in Chapter 2, but is included again here for completeness). The constructor in our runtime (a shared library) traverses through the link map of the application (ELF binary) at runtime and identifies the global data (*.bss* and *.data*) sections, i.e., the zero initialized and uninitialized data and non-zero initialized data, respectively. The runtime then unmaps these sections from the loaded binary image in memory, maps them from a SYSV memory mapped shared memory file and reinitializes these sections to the original values.

This mapping to a shared memory file is done by the *main* process before its execution begins at *main*. Speculative flows are then instantiated as processes (we use the *clone()* system call in Linux to ensure that file mappings are shared as well) and a copy of the address space of the parent is created for each instantiation of a speculation. Consequently, the speculations inherit the shared global data mapping. Hence any modifications made by a process to global data are immediately visible to all processes. Such a technique guarantees that all the processes have the same view of

global data, similar to a threads model. In essence, this technique creates a set of processes that are semantically identical to threads, but operate in distinct virtual address spaces. By controlling the binding to the shared memory mapping, data can be selectively isolated or shared based on the requirements of the speculation model.

To implement a shared heap, we modified Doug Lea's dlmalloc [27] allocator to operate over shared memory mappings so that the allocated memory is visible to all processes. Our runtime system provides global heap allocation by sharing memory management metadata among processes using the same shared memory backing mechanism used for .data and .bss sections. Hence any process can allocate memory that is visible and usable by other processes. If a process accesses memory that is not mapped in its address space, it results in a segmentation violation (a map error). Our runtime system handles this segmentation violation by consulting memory management metadata to check if the reference is to a valid memory address allocated by a different process. If so, it maps the shared memory file associated with the memory, thereby making it available. Note that such an access fault only occurs on the first access to a memory region allocated by a different process, and is conceptually similar to lazy memory allocation within an operating system.

To ensure program correctness, speculative flows within the same composition should appear as a concurrent continuation of the parent execution context. To achieve this, our runtime system ensures that the base address of the stack in a speculative flow is identical to that of the parent speculation. The default size of a stack is 8MB. When composing a speculation, the runtime saves only the stack frame of the parent speculation (not the entire 8MB) and each speculation within a composition uses a copy of this stack frame for execution. Each speculative flow is now identical to its parent flow, thereby creating a concurrent continuation.

Since each speculative flow is implemented as a process, it is important to note that in UNIX process semantics, each process is created with its own copy of the data segment of the shared libraries. Consequently, by default the runtime is not shared among speculation flows. To circumvent this problem and to maintain a shared and consistent view of the runtime, each newly created process automatically executes an initialization routine that maps the shared state of the Anumita runtime.

4.3.2 Speculative Composition

To mitigate the costs of creating and terminating speculative flows, the runtime instantiates a configurable pool of speculative flows in the Anumita library constructor before the *main* process begins its execution. Additional speculative flows are created as necessary. This pool of speculative flows is initially idle (blocked), waiting for work from the *main* process. On the termination of a speculative flow, it is returned back to the pool. In principle this is similar to a worker thread pool used to mitigate the performance impact of thread creation.

To instantiate a speculation, the parent flow first saves its current stack frame and execution context (*setjmp*) before waking the specified set of speculative flows from the pool. Upon waking, each speculation adjusts its execution context (*longjmp*), restores its stack to that of the parent flow and isolates its shared virtual memory address (VMA) before starting execution. The speculations begin their execution as a concurrent continuation of the `begin_speculation` construct. The parent flow then enters an evaluation context and waits for messages from the members of the speculative composition. The parent flow may be woken up under three scenarios.

First, if a speculative flow completes its assigned task it executes a `commit_speculation`. A call to `commit_speculation` is mutually exclusive to prevent race conditions on commits from multiple speculations. The first speculation to invoke commit is designated the winner. The winning speculation saves its current execution context and its stack frame so as to allow its parent to continue from the commit point. Additionally, the winning speculation attaches (*ptrace*) itself to the remaining sibling speculations and alters their instruction pointer to point to a cleanup routine. In the cleanup routine, it performs an inclusion (propagation of privatized updates) of the shared virtual memory address (VMA) and frees any dynamically allocated memory it allocated before returning to the pool. The winning speculation then commits its changes, wakes up its parent with a "winning speculation" message and joins the worker pool. Upon waking up, the parent flow adjusts its execution context and stack and returns from `commit_speculation` to continue its execution.

Second, if a speculative flow requests an evaluation, the parent flow executes the user defined eval-

uation function and returns a boolean value to indicate either a success or a cancellation. The speculative flow then either continues or aborts its execution based on the boolean value. Additionally, the parent flow can steer the computations of speculative flows. Anumita also implements a flexible approach to allow the parent flow to store the intermediate results of speculative flows for evaluation. In order to accomplish this, a speculative flow may access memory using `get_ir_memory`. This region of memory is shared between the parent flow and the speculative flow and it is unique to each speculative flow. This obviates the need for any synchronization among speculative flows to update their intermediate results.

Finally, in the event that all the speculations in a composition abort, the last speculation to abort (in program order within a composition) signals the parent flow to terminate the program, since no surrogate satisfied the expected quality criterion.

4.3.3 Nested Speculative Compositions

Implementing support for nested speculations presents additional challenges. Recall that in order to contain updates within a speculative flow, the pages modified in a speculative flow are privatized. Hence, if a speculative flow in turn creates a new composition, then it should propagate all its "privatized updates" to the speculative flows in the new composition. This has to be achieved without committing the updates, since the parent of the nested speculation may not be the winner in its composition. Conversely, the updates by the speculations in a nested composition should be propagated only to its parent flow to ensure program correctness.

To resolve this, in the case of nested speculations, the runtime creates new speculative flows during the call to `begin_speculation` instead of using a worker pool entry. This creates a current copy of the parent flow and includes privatized updates. Since the parent is blocked upon composing a speculation, the lazy copy-on-write semantics provided by the operating system efficiently creates isolated private address spaces for the nested speculations.

4.3.4 Containment

In order to determine the write-set and and contain (privatize) the updates of a speculation, Anumita employs page level protection and privatization of the shared VMA. Each speculative flow initially write-protects (PROT_READ) its shared VMA. Read accesses to shared data do not produce any faults and execute normally. However, if a speculation attempts to write to a shared page (global data, heap), the runtime handles the access fault (SEGV_ACCERR) by remapping the faulting page from the shared memory in private (MAP_PRIVATE) mode. The runtime maintains a list of pages that were modified by each speculation within a composition. The permissions of the page are then reset to read-write so that the speculation can continue its execution.

This privatization provides containment of updates by a speculation. Such a lazy privatization scheme that defers privatization until the instant memory update happens is sufficient to ensure program correctness in our speculation model. We do not present a formal proof of correctness, however, the intuition behind the proof is that once a winning speculation commits, none of the remaining speculations will be allowed to commit, thus precluding Write-after-Read or Write-after-Write hazards. Hence, we chose the lazy privatization approach over a conservative approach, which privatizes the entire VMA.

The runtime does not track accesses of local variables on the stack. Since a copy of the parent stack frame is passed to each speculation, the stacks do not need to be write-protected. Instead the parent's stack frame is updated with contents of the stack frame from the winning speculation. Such a strategy works for programs that contain pointers to stack-allocated data.

4.3.5 Inclusion

When a speculation composition culminates with a winning surrogate, the updates of the winning speculation (contained in the privatized data) must be propagated and made visible to the parent flow and any other non speculative flows in the program.

In order to perform this *inclusion of updates*, we implement the shadow addressing technique similar

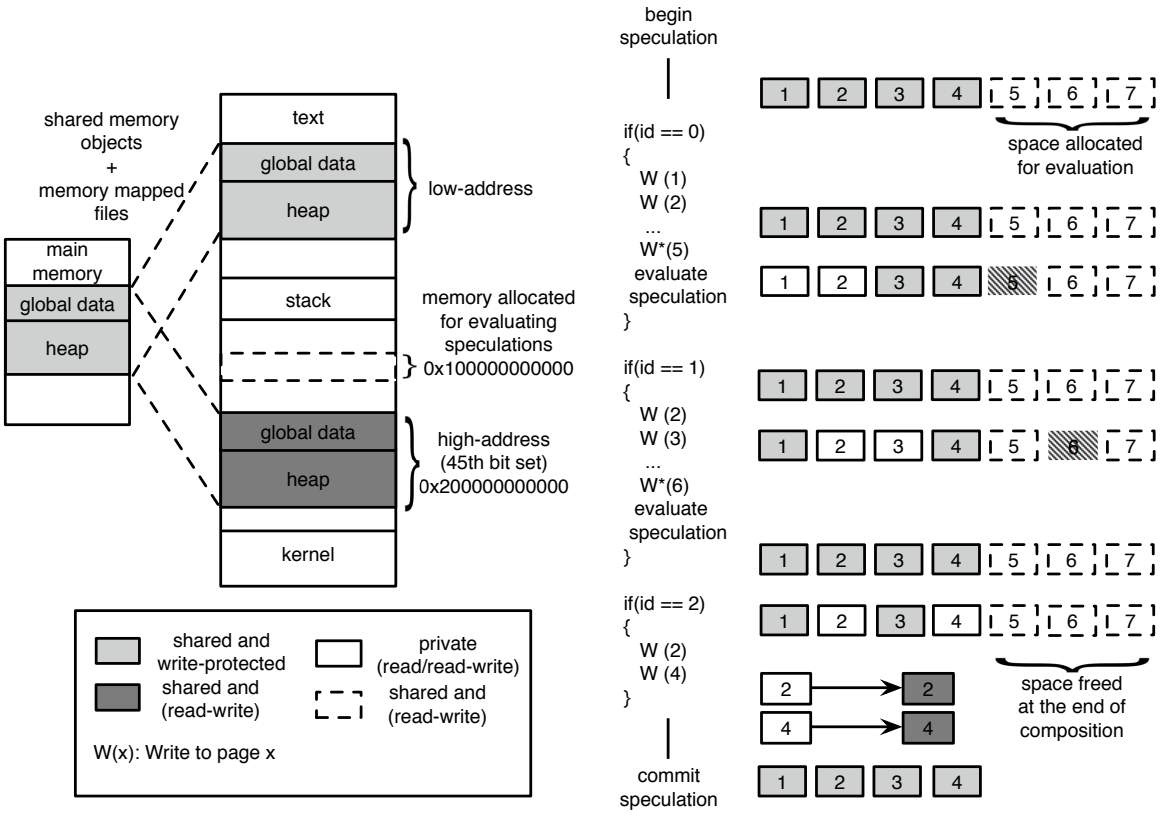


Figure 4.5: (left) Illustrates the virtual memory address (VMA) layout of each process. The runtime provides memory isolation by using shared memory objects and memory mapped files to share global state among processes. (right) Illustrates how the VMA is manipulated by the runtime with a simple example.

to [75] that leverages the large virtual memory address (VMA) provided by 64-bit operating systems. Recall that the runtime system maps globals and the heap (described in Section 4.3.1) using shared memory objects and memory mapped files. Using the same shared memory objects and memory mapped file, the runtime creates an identical secondary *shadow* mapping of the global data sections and heap at a *high address* (45th bit set) in the VMA of each speculation flow. The program is unaware of this mapping, and performs its accesses (reads/writes) at the original low address space. The high address space shadow is always shared among speculative flows and is never privatized. Hence, any updates to it are propagated across all flows. In effect, the high address mapping creates a *shadow* address space for all shared program data and modifications (unless privatized) are visible in both address spaces (shown in Figure 4.5).

To perform this inclusion the runtime employs two distinct strategies depending on the depth of the speculations (nested or otherwise). In a single level speculation, where a flow creates a composition, updates from the winning speculation must be made visible to the parent flow. To achieve this, the runtime copies all the pages in the write set of the winning speculation to the high address shadow region of the VMA (shown in Figure 4.5), which automatically propagates the updates to the parent flow, due to the shared memory bindings. The runtime then reverts all privatized mappings within the speculative flows (cleanup) and returns them to the pool.

In a nested speculation, the runtime creates a new shared memory mapping equal to the write set of the winning speculation and it copies the write set of the winning speculation to the newly created mapping. The parent flow then copies the write-set from this new mapping into its address space to perform inclusion.

4.3.6 Example

We present a simple example to illustrate how the runtime manipulates the VMA of each speculation flow while providing isolation, privatization and inclusion. Consider the scenario as shown in Figure 4.5 where a control flow creates a composition involving three speculations. Initially, all the shared pages (1, 2, 3, 4) of the speculative flows are write-protected. When the flow with rank

0 attempts to write to pages (1, 2), the pages are privatized (lazy privatization). Similarly, the runtime system privatizes the updates of speculations with ranks 1 and 2, which write to pages (2, 3) and (2, 4) respectively. If the speculation with rank 2 wins the composition, the runtime commits the write-set to the shared high-address space to propagate the updates.

Additionally, in Figure 4.5 we illustrate how the speculative flows may request evaluation of their progress. At the beginning of a speculative composition, a program may choose to request space for storing partial results. In the above example, pages (5, 6, 7) are allocated by the runtime system to store the partial results. Each speculation may use the `get_ir_memory()` to obtain the address of the region used to store its partial results. The speculative flow at rank 0 writes its partial results to page 5, before requesting evaluation. Following the same procedure, speculative flow 1 writes to page 6 before requesting evaluation from its parent flow. The parent flow can access these memory locations and execute the evaluation function to determine the relative quality and/or progress of the speculative flows.

Recall, that while Anumita supports multi-threaded applications, caution should be exercised in leveraging speculative parallelism in a multi-threaded environment. As discussed in Section 4.2.1, if a surrogate requires concurrency with other non-speculative threads or other concurrently executing speculative threads then it is not a candidate for speculation, since such a surrogate is based on concurrency rather than speculation. Hence, in the presence of data dependencies we expect explicit serialization in composing speculations in order to ensure program correctness.

A complication arises when two distinct threads of a process independently instantiate speculative compositions that execute concurrently. This is depicted in Figure 4.1, where threads 0 and 2 enter distinct compositions. While the compositions may be distinct, the granularity of our protection mechanism is at the level of an operating system page, which can cause false-sharing if updates to distinct bytes from distinct compositions reside on the same operating system page. The artifact of this problem is that in the case of concurrent speculations, the contents on a page subject to false sharing will reflect the updates from the last speculation in program order to successfully commit without reflecting any of the updates from concurrent commits. For example in Figure 4.1, if the

monte-carlo method updated page 1, which was also updated (albeit at different locations) by the *adaptive* method, the final contents of page 1 will reflect the updates from *adaptive* and none of the updates from *monte-carlo*. In effect, updates to pages subject to false sharing are mutually exclusive, which is clearly incorrect.

To solve this problem, we need an efficient mechanism to propagate updates from a winning speculative flow to all concurrent speculations. This is achieved by computing and propagating XOR differences. To see how this works, consider the example shown in Figure 4.1. When *monte-carlo* wins the speculative composition in thread 2, prior to performing inclusion the runtime determines if there are concurrent speculative compositions in other threads. If so, for each page in the write-set of the first winning flow (*monte-carlo*), the runtime computes an XOR of the privatized page with its counterpart in the high address space. Recall that prior to inclusion, the page in the write-set of *monte-carlo* is privatized and includes updates from the flow, whereas its counterpart in the shadow address space contains the original contents of the page. The XOR difference thus yields the exact set of bits that were updated by *monte-carlo* method.

The runtime then pushes this XOR difference to all concurrently executing flows (*adaptive*, *conservative*, *extrapolation*) and performs inclusion of updates from *monte-carlo* as before. The concurrent flows apply the differences by computing the XOR of the difference they received with their privatized copy of the page (if one exists due to false sharing). Intuitively this mechanism updates the privatized contents of a concurrent speculation with the latest updates from a winning speculation in another composition. In the example above, when *adaptive* finally wins its composition, its write set already contains the updates from *monte-carlo* and hence the final resulting update of a falsely shared page from *adaptive* correctly contains the cumulative updates from winning speculations. To minimize time and space overhead, the XOR difference is only computed on the pages in the write set that are subject to false sharing (typically small), which is determined by computing the intersection of the write sets of the winning flow (*monte-carlo*) and all other concurrently executing flows (*adaptive*, *conservative*, *extrapolation*) .

4.3.7 Support for OpenMP

To support OpenMP, we provide a simple source to source translator that expands the *#pragma speculate (...)*{....} directive to begin and commit constructs. Our translator parses only the speculate pragma leaving the rest of the OpenMP code intact. This approach does not require any modifications to existing OpenMP compilers and/or OpenMP runtime libraries.

Our runtime system overrides mutual exclusion locks, barriers and condition variables of the POSIX thread interface and a few OpenMP library routines in order to provide a clean interface to OpenMP. We overload the `omp_get_thread_num` call in OpenMP to return the speculation rank from `get_rank`. The Anumita runtime automatically detects if an OpenMP program is in a speculative context and selectively overloads OpenMP calls, which fall back to their original OpenMP runtime when execution is outside a speculative composition. Finally, our OpenMP subsystem implements a simple static analyzer to perform lexical scoping of a speculative composition. This can be used to check for logical errors such as a call to commit before beginning a speculation.

4.4 Experimental Evaluation

We evaluated the performance of the Anumati runtime over three applications: a multi-algorithmic PDE solving framework [83], a graph (vertex) coloring problem [57] and a suite of sorting algorithms [98].

We ran each benchmark under two scenarios. The first scenario uses Anumita to speculatively execute multiple algorithms concurrently. This was done by modifying approximately 8-10 lines of source code in the above benchmarks. Since Anumita guarantees isolation, these modifications were short and required little to no understanding of the algorithms themselves. In the other scenario we ran the vanilla benchmark executing each algorithm individually. All experiments were performed on a 16 core shared memory machine (NUMA) running Linux 2.6.31-14 with 64GB of RAM. The system contains four 2 GHz Quad-core AMD Opteron processors.

4.4.1 PDE Solver

One approach for dealing with the unpredictable input-dependent performance of PDE solvers is a 'poly-algorithmic' strategy, where multiple algorithms are tried in parallel, with the one finishing first declared the winner, e.g., [6, 10]. This approach is robust, essentially guaranteeing a solution, and is easily implemented using our framework.

We consider the scalar linear elliptic equation [78]

$$-\nabla^2 u + \frac{\alpha}{(\beta + x + y)^2} u_x + \frac{\alpha}{(\beta + x + y)^2} u_y = f(x, y),$$

with Dirichlet boundary conditions on the unit square, where $\beta > 0$. Discretized with centered finite differences, the resulting linear system of algebraic equations is increasingly ill-conditioned for large α and small β. Krylov linear solvers have difficulty as this problem approaches the singular case, i.e., as α/β^2 grows. What is not so clear is how quickly the performance degrades, and how much preconditioning can help. To simplify the case study, we fix β at 0.01 and vary α.

Discretizing the problem using a uniform grid with spacing $h = 1/300$ results in a linear system of dimension 89401. We consider three iterative methods and one direct method for solving this system of equations:

1. GMRES(`kdim=20`) with ILUTP(`droptol=.001`)
2. GMRES(`kdim=50`) with ILUTP(`droptol=.0001`)
3. GMRES(`kdim=100`) with ILUTP(`droptol=.00001`)
4. Band Gaussian Elimination

Here `kdim` is the GMRES restart parameter, ILUTP is the "incomplete LU with threshold pivoting" preconditioner [86, Chap. 10], and `droptol` controls the number of nonzeros kept in the ILU preconditioner. Increasing `kdim` or decreasing `droptol` increases the computational cost per iteration of the iterative method, but should also increase the residual reduction per iteration. Hence, one

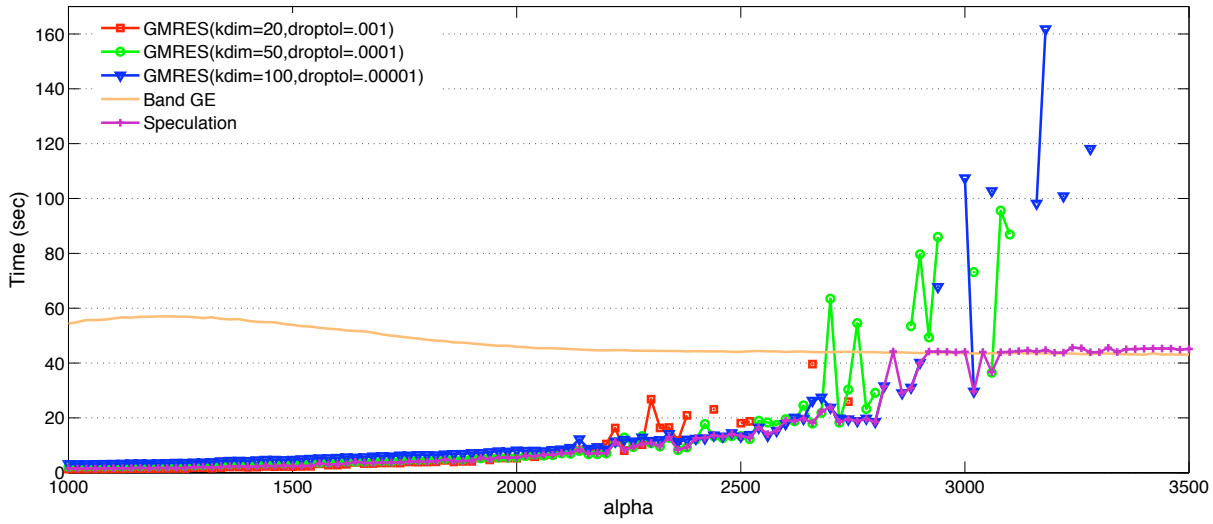


Figure 4.6: Time to solution for individual PDE solvers and speculation based version using Anumita. Cases that fail to converge in 1000 iterations are not shown. The results show that Anumita has relatively small overhead, allowing the speculation based program to consistently achieve performance comparable to the fastest individual method for each problem.

can think of methods one to four as being ordered form "fast but brittle" to "slow but sure." Our PDE-solving framework for these experiments is ELLPACK [83], with the GMRES implementation from SPARSKIT [85].

Figure 4.6 shows the performance of the four methods and speculation for varying α. For small α the results are consistent, with Method 1 consistently fastest. As α grows, however, the performance of the iterative methods vary dramatically, with each method taking turns being the most efficient. In many cases the GMRES iteration fails to converge (i.e., iterations exceeding 1000 are not shown in the figure). Eventually, for large enough α, Band GE is the only method that succeeds.

The write set of the PDE solver is 157156 pages ($\approx$ 614MB) of data. The overhead of speculation shrinks steadily as the problem difficulty grows, with overheads of no more than 5% for large α. This is to be expected since the time to solve sparse linear systems grows faster as a function of problem dimension than the data set size, which largely determines the overhead. However, in cases with small ($<$ 10 sec) runtime, the overhead due to speculation is noticeable (up to 16%). This is due to initial thread creation and start up costs which are otherwise amortized over the runtime of

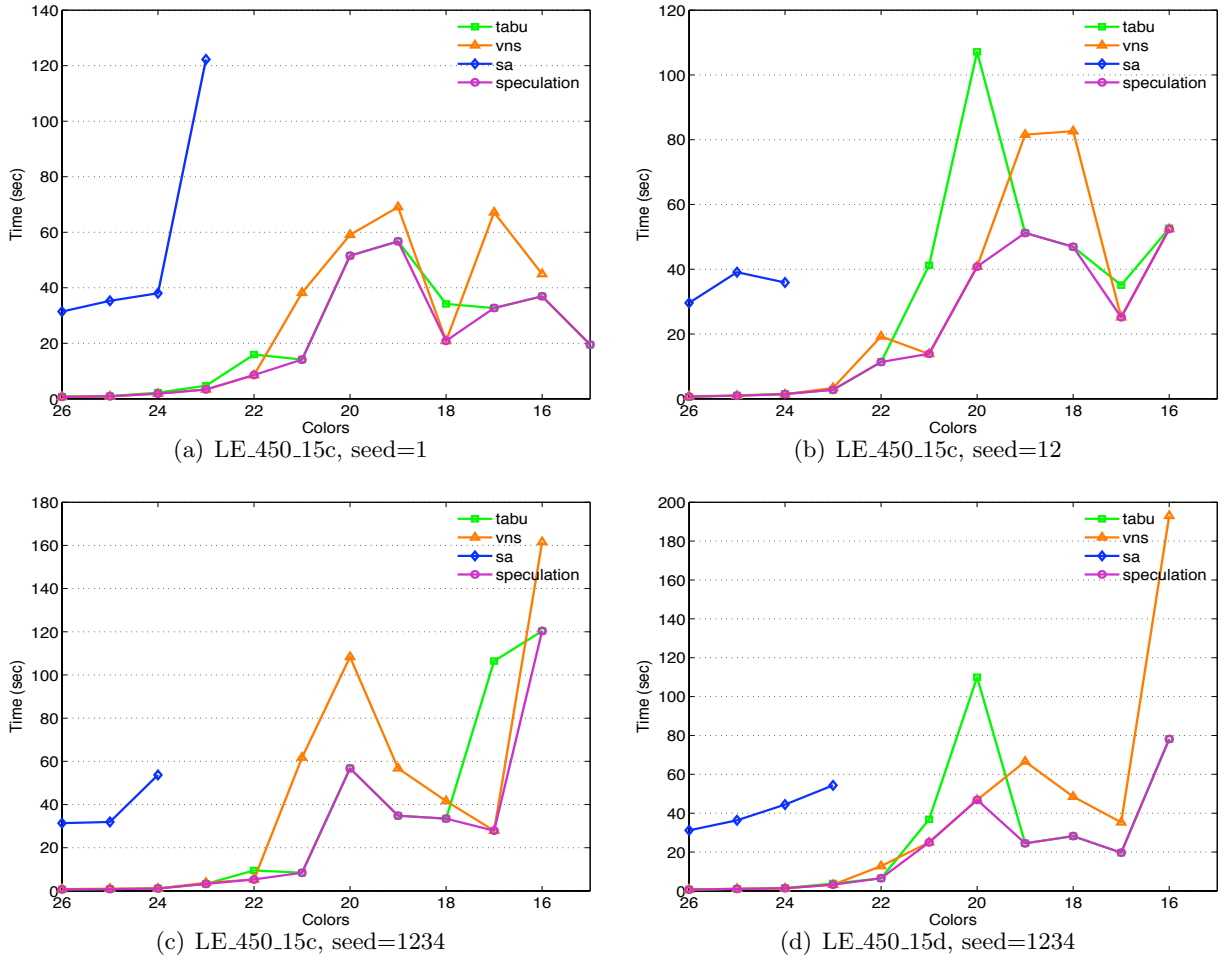
Table 4.2: Number of failing cases (out of 125) for each PDE solver, and speedup of speculative approach relative to each method.

		Speedup		
Method	Fail	Min	Max	Median
1	51	0.84	2.47	0.94
2	27	0.94	2.89	1.18
3	23	0.94	3.62	1.52
4	0	0.95	36.19	5.01

a larger run.

The results show that speculative execution provides clear benefits over any single static selection of PDE solver. Table 4.2 summarizes the performance of the four methods relative to the speculatively executed case. Statically choosing any one of the GMRES methods (Methods 1-3) causes a serious robustness problem, as many of the problems fail completely. Even for the cases where GMRES succeeds, we see that the speculative approach yields noticeable improvements. For the problems where method 1 succeeds, it is faster than speculation more than half the time (median speedup = 0.94). Compared to methods 2-4 speculation is significantly faster in the majority of cases. In essence, speculation dynamically chooses the best algorithm for a given problem, with minimal overhead.

It must be pointed out that the speculative code uses four computational cores, while the standalone cases each use only one core. In the case where we only have sequential implementations of a given surrogate, speculation gives us a convenient way to do useful work on multiple cores, moving more quickly on average to a solution. However, given parallel implementations of each of the four methods, an alternative to speculation is to choose one method to run (in parallel) on the four cores. However, this strategy still suffers from the risk of a method failing, in which case one or more additional methods would have to be tried. In addition, it is well-known that sparse linear solvers do not exhibit ideal strong scaling, i.e., parallel performance for a fixed problem does not scale well to high core counts. By contrast, running each surrogate on a core is embarrassingly parallel; each core is doing completely independent work. Given hundreds of cores, the optimal strategy is likely to be to use speculation at the highest level, with each surrogate running in



(a) LE_450_15c, seed=1

(b) LE_450_15c, seed=12

(c) LE_450_15c, seed=1234

(d) LE_450_15d, seed=1234

Figure 4.7: The performance of Graphcol benchmark using two DIMACS data sets LE_450_15c (subfigures (a) through (c)) and LE_450_15d (subfigure (d)).

parallel on some subset of the cores. Choosing the number of cores to assign to each surrogate should depend on the problem and the scalability of each method on that problem, and is beyond the scope of this discussion.

4.4.2 Graph Coloring Problem

In graph theory, vertex coloring is the problem of finding the smallest set of colors needed to color a graph $G = (V, E)$ such that no two vertices $v_i, v_j \in V$ with the same color share an edge e. The graphcol [57] benchmark implements three surrogate heuristics for coloring the vertices of a graph:

simulated annealing, tabu search and variable neighborhood search. The benchmark initializes the graph by randomly coloring the vertices with a specified set of colors and each heuristic algorithm iteratively recolors the graph within the coloring constraints. We used the DIMACS [25] data sets for the graph coloring benchmark, which are widely used in evaluating algorithms and serve as the testbed for DIMACS implementation challenges. Each data set (graph) has a fixed number of colors that it can use to color a graph. We experimented with over 80 DIMACS data sets using different seeds (for initial colors) and show the results from representative runs.

In Figure 4.4.1 we present the results of the graph coloring benchmark using two DIMACS data sets. The results show several interesting characteristics. First, certain heuristics do not converge and cannot guarantee a solution. For instance, simulated annealing (*sa*) cannot color the graph beyond a certain number of colors. Second, the choice of the input seed, which decides the initial random coloring, creates significant performance variations among the heuristics (Figures 4.4.1 (a) through (c)) *even when the graph is identical*. Third, when the seed is constant, there is performance variation among the data sets, which represent different graphs as shown in Figures 4.4.1 (c) and (d). In the presence of such strong input dependence across multiple input parameters, it is difficult even for a domain expert to predict the best algorithm a priori.

Using Anumita it is possible to obtain the best solution among multiple heuristics. We found that in some cases where *sa* failed to arrive at a solution (unable to color the graph using specified number of colors), the use of speculation guaranteed not only a solution but also one that is nearly as fast the fastest alternative. Since the write set is relatively small at around 50-100 pages, the overhead of speculation is negligible. Anumita's speedup, across all the data sets (in Figure 4.4.1), ranges from 0.954 (vns with 26 colors in Figure 4.4.1 (b)) in the worst case, when the static selection is the best surrogate, to 7.326 (vns with 21 colors in Figure 4.4.1 (d)), when the static selection is the worst surrogate. We omit the results from the *sa* method in calculating speedup since *sa* consistently performs worse than the other algorithms on these data sets.

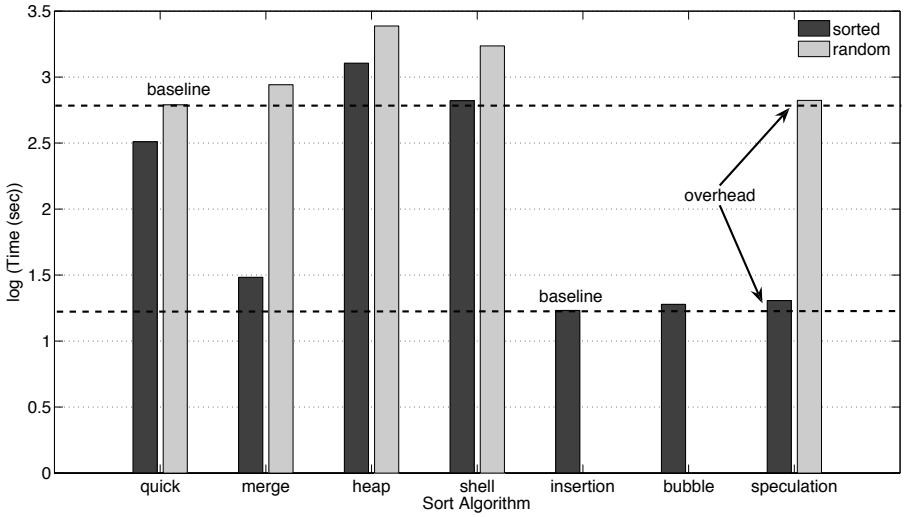


Figure 4.8: Performance of Anumita over a suite of sorting algorithms.

4.4.3 Sorting Algorithms

Since the overhead of speculation in our runtime is proportional to the write set of an application, we chose sort as our third benchmark since it can be configured to have an arbitrarily large memory footprint. Sort is relatively easy to understand, and yet there are a wide variety of sorting algorithms with varying performance characteristics depending on the size of the data, sortedness and cache behavior [5]. Our suite of sorting algorithms includes C implementations of quick sort, heap sort, shell sort, insertion sort, merge sort and bubble sort.

The time to completion of the sorting algorithms is based on several cardinal properties including the input size, their values (sorted or unsorted) and algorithmic complexity. In this set of experiments we fixed the input size and used two sets of input data — completely sorted and completely random, each of size 8 GB. Each sorting algorithm is implemented as a separate routine. The input data is generated using a random number generator. After sorting the data the benchmark verifies that the data is properly sorted. We measured the runtime of each sorting algorithm and excluded the initialization and verification phases. Using Anumita, we speculatively executed all six sorting algorithms concurrently.

In Figure 4.8 we present the results of the sort benchmark. Results for insertion sort and bubble sort for random data were omitted since their runtime exceeds 24 hours. The results show that insertion sort is the fastest for sorted data and quick sort performs the best on completely random data, which is expected. Despite the large write set of 8 GB per speculation, a total of 6x8GB for the entire speculative composition, Anumita is at least the second fastest of all the alternatives considered and is nearly as fast as the fastest alternative.

The worst case overhead of speculation on sorted data relative to the best algorithm (insertion sort) is 15.78% (3.2 sec), which stems from the map faults handled by the runtime system. The worst case overhead of speculation compared to the fastest algorithm on the random data is 8.72% (50.34 sec over 616 secs). This overhead stems from privatization, isolation and inclusion of the large 8 GB data set. Anumita achieves a speedup ranging from 0.84 (quick sort/random data) to 62.95 (heap sort/sorted data).

4.4.4 Memory Overhead

Anumita's memory overhead depends on the number of speculative flows within a composition and their corresponding write sets. For each speculative flow, Anumita's employs privatization at the granularity of a page, consequently, each speculative flow will incur 1x memory overhead for every unique page it modified. The total write set of the PDE solver is 614MB, hence the total memory consumption is 1228MB. For the Graph coloring problem the write set is around 50 to 100 pages. Hence its memory consumption ranged between 100 and 200 pages. The sorting algorithm had a total write set of 8GB hence the total memory footprint using Anumita is 16GB.

4.4.5 Energy Overhead

The primary focus of Anumita is to improve run time performance. Reducing energy consumption runs counter to this goal. However, in this section, we demonstrate that adopting coarse-grain speculation to exploit parallelism on multi-core systems does not come with a large energy consumption

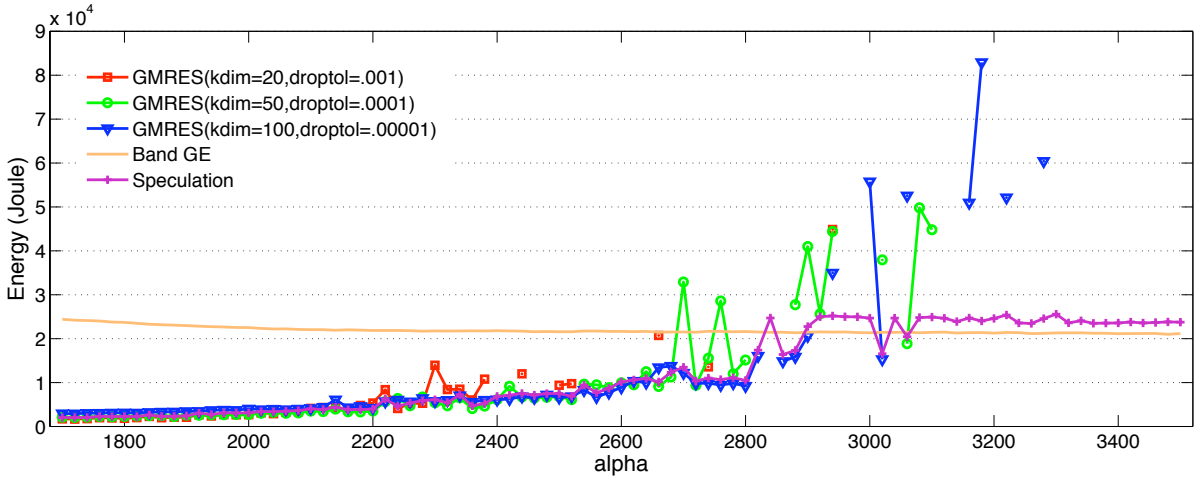


Figure 4.9: Energy consumption of PDE solver using surrogates in Anumita. The results show that Anumita has relatively low energy overhead.

penalty, and in fact can reduce total energy consumption in many cases.

Energy consumption in modern multi-core processors is not proportional to CPU utilization. An idle core consumes nearly 50% of the energy of a fully loaded core [67]. There is a significant body of research in the architectures community on making the energy consumption proportional to offered load, which is motivated by energy consumption of large data centers that run at an average utilization of $7 - 10\%$.

To measure the energy overhead of coarse-grain speculative execution using Anumita, we connected a Wattsup Pro wattmeter to the AC input of the 16 core system running the benchmark. This device measures the total input power to the entire system. We performed the power measurement using the SPEC Power daemon (ptd), which samples the input power averaged over 1 sec intervals for the entire run time of the application. We calculated energy consumption as the product of the total runtime and the average power. We measured energy consumption under two scenarios: a) each algorithm run individually and b) speculatively execute multiple algorithms using Anumita.

Figure 4.9 presents the energy consumption of the PDE solver. We report the results for α values greater than 1700 for the PDE solver, since they have a runtime of at least a few seconds (required to make any meaningful power measurements). Comparing the most energy efficient algorithm at

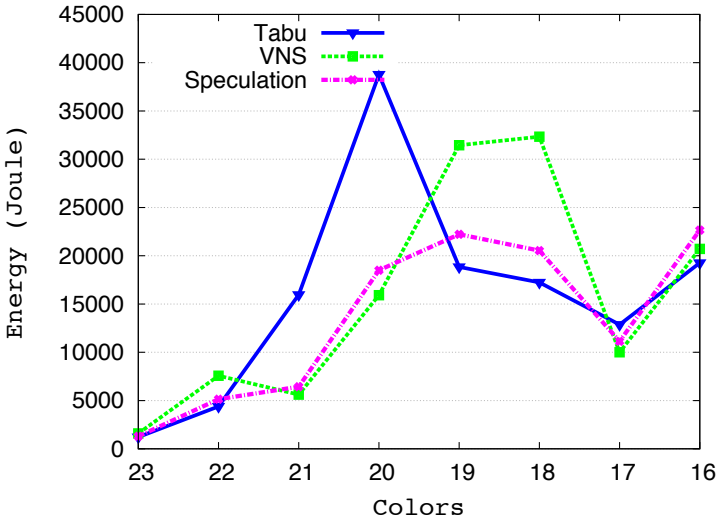


Figure 4.10: Energy consumption of the graph coloring benchmark for the LE_450_15c data set with a seed of 12.

each α with the corresponding speculative execution, we found that the overall energy overhead of speculation ranged between 7.72% and 19.21%. It is comforting to see that, even in the presence of running four surrogates concurrently, Anumita incurred a maximum energy overhead of 19.21% compared to the most energy efficient algorithm.

More importantly, since the most energy efficient algorithm for a given problem is not known a priori, we again see a large robustness advantage for speculation — this time with respect to energy consumption. With a static choice of algorithm there is substantial risk that a method will fail (necessitating the use of another method) or take much longer than the best method, all of which consume more energy than the speculatively executed approach.

Figure 4.10 shows the energy consumption for two vertex coloring algorithmic surrogates (*tabu*, *vns*) and speculation using Anumita. In this case, Anumita speculates over three algorithms (*sa*, *tabu* and *vns*) even though one of them consistently fails to color the graph. Comparing the most energy efficient algorithm at each color with the corresponding speculation, we found that the overall energy overhead due to speculation ranged between 6.08% and 16.04%.

The total energy consumed to color the graph is actually lower for speculation when compared

to a static choice of either algorithmic surrogate. This is because energy is the product of power and time, and since neither algorithm is consistently better (strong input dependence), speculation results in lower time to completion of an entire test case which translates to lower energy consumption. Speculation using Anumita takes 252 seconds for the problem set with a total energy consumption of 107904 joules. *Tabu* takes a total of 321 seconds and consumes a total energy of 128531 joules for the problem set. In contrast the best static choice of the surrogate (*vns*) runs in 314 seconds and consumes 125194 joules. Speculation here is 24.6% faster in time and consumes 16.02% less energy, a result that is positive in both aspects.

4.4.6 Summary of Results

Anumita provides resilience to failure of optimistic algorithmic surrogates. In both graph coloring and PDE solvers, not all algorithmic surrogates successfully run to completion. In the absence of a system such as Anumita, the alternative is to run the best known algorithmic surrogate and if it fails, retry with a fail-safe algorithm that is known to succeed. While this works for the PDE solving example with Band Gaussian Elimination being the fail-safe, there is no clear equivalent for graph coloring, with each surrogate failing at different combinations of graph geometry and initial coloring. With modest energy overhead and sometimes savings, Anumita can significantly improve the performance of otherwise hard to parallelize applications.

4.5 Related Work

We categorize existing software based speculative execution models [26, 47, 82, 50, 49, 48, 70, 79, 96, 99, 20] into two categories depending on the granularity at which they perform speculation — loops or user defined regions of code. Loop level models [82, 50, 49, 48, 79, 96, 99, 20] achieve parallelism in sequential programs by employing speculative execution within loops. While such models transparently parallelize sequential applications without requiring any effort from the programmer, their scope is limited to loops. In contrast, the second category of speculative exe-

cution models [9, 26, 47, 70, 101] allow the programmer to specify regions of code to be evaluated speculatively. We restrict our discussion to these models throughout the rest of this section.

Berger et al. [9] proposed the Grace framework to speculatively execute fork-join based multi-threaded applications. Grace uses processes for state separation with virtual memory protection and employs page-level versioning to detect mis-speculations. Grace focuses on eliminating concurrency bugs through sequential composition of threads.

Ding et al. [26] proposed behavior oriented parallelization (BOP). BOP aims to leverage input dependent course grained parallelism by allowing the programmer to annotate regions of code, denoted by *possibly parallel regions* (PPR). BOP uses a lead process to execute the program non-speculatively and uses processes to execute the possibly parallel regions. When the lead process reaches a PPR, it forks a speculation and continues the execution until it reaches the end of the PPR. The forked process then jumps to the end of the PPR region and in turn acts as lead process and continues to fork speculations. This process is repeated until all the PPRs in the program are covered. BOP's PPR execution model is identical to pipelining. The lead process at the start of the pipeline waits for the speculation it forked to complete and then checks for conflicts before committing the results of the speculation. This process is recursively performed by all the speculation processes that assumed the role of the lead process. BOP employes page-based protection of shared data by allocating each shared variable in a separate page and uses a value-based checking algorithm to validate speculations.

In another study, Kelsey et al. [47] proposed the Fast Track execution model, which allows unsafe optimization of sequential code. It executes sequential (normal tracks) and speculative variants (fast tracks) of the code in parallel and compares the results of both these tracks to validate speculations. Their model achieves speedup by overlapping the normal tracks and by starting the next normal track in program order as soon as the previous fast track is completed. Fast Track performs source transformation to convert all global variables to use dynamic memory allocation so its runtime can track accesses to global variables. Additionally, Fast Track employes a memory-safety checking tool to insert memory checks while instrumenting the program. Finally, Fast Track

provides the programmer with configurations that tradeoff program correctness against performance gains. In contrast, Anumita provides transparent name space isolation and it does not require any annotations to the variables in a program. Additionally, Anumita does not rely on program instrumentation.

Prabhu et al. [70] proposed a programming language for speculative execution. Their model uses value speculation to predict the values of data dependancies between coupled interactions based on a user specified predictor. Their work defines a safety condition called rollback freedom and is combined with static analysis techniques to determine the safety of speculations. They implemented their constructs as a C# library. The domains where value speculation is applicable are orthogonal to our work.

Trachsel and Gross [101, 100] present an approach called competitive parallel execution (CPE) to leverage multi-core systems for sequential programs. In their approach they execute different variants of a single threaded program competitively in parallel on a multicore system. The variants are either hand generated surrogates or automatically generated by selecting different optimization strategies during compilation. The program's execution is divided into phases and the variants compete with each other in a phase. The variant that finishes first (temporal order) determines the execution time of that phase, thereby reducing the overall execution time. In contrast, Anumita is capable of supporting both sequential and parallel applications and provides expressive evaluation criterion (temporal and qualitative) to evaluate speculations.

Praun et al. [104] propose a programming model called implicit parallelism with ordered transactions (IPOT) for exploiting speculative parallelism in sequential or explicitly parallel programming models. The authors implement an emulator using the PIN instrumentation tool to collect memory traces and emulate their proposed speculation model. In their work, they propose and define various attributes to variables to enable privatization at compile time and avoid conflicts among speculations. In contrast, as mentioned previously, Anumita does not require annotations to variables or rely on binary instrumentation. Instead, Anumita provides isolation to shared data at runtime.

In another study, Cledat et al. [21] proposed opportunistic computing, a technique to increase the performance of applications depending on responsiveness constraints. In their model multiple instances of a single program are generated by varying input parameters to the program, which then compete with each other. In contrast, Anumita is designed to support speculation at arbitrary granularity as opposed to the entire program.

Ansel et al. [5] proposed the PetaBricks programming language and compiler infrastructure. PetaBricks provides language constructs to specify multiple implementations of algorithms in solving a problem. The PetaBricks compiler automatically tunes the program based on profiling and generates an optimized hybrid as a part of the compile process. In contrast, our approach performs coarse-grain speculation at runtime and is hence better suited for scenarios where performance is highly input data dependent.

Additionally, certain compiler directed approaches [11, 40, 41, 56, 58, 53] provide support for speculative execution and operate at the granularity of loops. Such approaches rely on program instrumentation [41], use hardware counters for profiling [41] or binary instrumentation to collect traces [56, 58] in order to optimize loops. In contrast to such systems, Anumita is implemented as a language independent runtime system. The main goal of Anumita is to simplify the notion of speculative execution.

Finally, a nondeterministic programming language (e.g., Prolog, Lisp) allows the programmer to specify various alternatives for program flow. The choice among the alternatives is not directly specified by the programmer, however the program at runtime decides to choose between the alternatives [1]. Several techniques such as backtracking and reinforcement learning are commonly employed in choosing a particular alternative. It is unclear if it is the responsibility of the programmer to ensure and correct the side-effects of the alternatives. Anumita represents a concurrent implementation of the non-deterministic choice operator. The contribution here is to introduce this notion and an efficient implementation to imperative programming.

4.6 Future Work

We are continuing to improve Anumita. We are presently working on extending support for disk IO among speculative surrogates. While Anumita simplifies the subtleties of coarse-grain speculative parallelism by providing simple sequential semantics, the programmer must identify the scope for speculation. We plan to automate this aspect of our system. Currently there is an ongoing effort [13, 14, 2] to extend C++ to include threading models. We propose that *speculation should also be a natural extension of the imperative languages and the speculation model should be a natural extension to threading models.* We plan to investigate extending language support for speculation.

4.7 Summary

In this chapter we presented Anumita, a language independent runtime system to achieve coarse-grain speculative parallelism in hard to parallelize and/or highly input dependent applications. We proposed and implemented programming constructs and extensions to the OpenMP programming model to achieve speedup in such applications without sacrificing performance, portability and usability. We have shown that speculative execution at coarse granularities (e.g., code-blocks, methods, algorithms) offers a promising programming model for exploiting parallelism on modern architectures. Our experimental results from a performance evaluation of Anumita show that it is (a) robust in the presence of performance variations or failure and (b) achieves significant speedup over statically chosen alternatives with modest overhead.

Chapter 5

Conclusion

Multi and many core architectures have become ubiquitous. Their wide prevalence has brought issues related to concurrency and parallelism to the forefront of everyday computing. This dissertation describes contributions in two major areas that seek to address the challenges and opportunities of multi/many core parallelism.

Safe Concurrent Execution

Many applications are amenable to traditional parallelization techniques and can benefit from multi/many core architectures but are susceptible to the subtleties of concurrent programming — concurrency bugs. One of the most difficult challenges facing computer science researchers is how to provide safe and efficient mechanisms to enable a large number of programmers, representing a wide range of applications, to use these multi-core architectures effectively. In practice the most recurrent concurrency bugs are data races and deadlocks. Due to the potential of deadlocks, lock based codes have been deemed non-composable — a fundamental problem in lock based code. This dissertation presents practical techniques to transparently and deterministically detect and eliminate deadlocks, enabling a next generation of safe composable lock based codes.

Transparently eliminating deadlocks and related programming errors is a particularly difficult prob-

lem when targeting multi-threaded applications written in type-unsafe languages such as C and C++. The challenges include the following.

- Determining and exploring all possible thread interleavings is often impractical and gets worse in the presence of function pointers. Additionally, an application may depend on several external libraries which should also be verified for deadlock freedom. Detecting deadlocks precisely at compile time in type-unsafe languages is not feasible. It is important that deadlock detection does not incur any false positives or false negatives. Often the imprecision of program analysis deters these tools from being used in production systems. Programmers are often under pressure to meet product deadlines, expecting them to sift through false positives hoping to find potential concurrency bugs is not practical. Expecting programmers to use heavy weight runtime techniques which could potentially slowdown the application's performance is also unrealistic. Hence, *efficiently*, *transparently*, and *deterministically* detecting deadlocks is extremely hard.

- In practice uncovering a concurrency bug does not necessarily mean that it can be fixed easily. To properly fix the concurrency bug, the programmer must identify the root cause of the problem rather than simply observing how the bug manifests itself in the program execution. Concurrency bugs are often hard to reproduce due to the non-deterministic nature of thread execution and properly fixing a concurrency bug may require a major software redesign. Hence transparently eliminating concurrency bugs (particularly deadlocks) without requiring programmer assistance or modifications to application source code is a daunting task.

- To recover from a deadlock transparently, we need efficient techniques that (a) associate the memory updates with locks, (b) isolate and contain the memory updates from being visible to other concurrently executing threads, while (c) preserving the synchronization semantics, (d) propagating the memory updates on the release of a critical section, and (e) recovering I/O within critical sections and offering rollback aware memory management. Type-unsafe languages allow arbitrary pointer accesses and dynamic memory allocation, consequently, providing isolation during compilation is not a practical option. We need powerful runtime

support and all the compile time analysis that we can leverage to successfully accomplish these objectives.

- To simplify lock based programming and achieve safe and efficient concurrent execution we need to detect and eliminate other programming errors affecting lock based codes such as asymmetric data-races, priority inversion, order violations, live-locks, asynchronous signaling, and locks and related performance issues in addition to deadlocks.
- To make a meaningful impact, in the context of emerging multi-threaded applications with increasing size (code base), and complexity, we need scalable, competent, and practical techniques for concurrency bug detection and/or recovery.

This dissertation is devoted to addressing these challenges. In this dissertation, we carefully studied and analyzed these issues in detail. We have systematically broken down these problems into smaller pieces and presented techniques to address them.

We presented an execution model that provides the ability to selectively share and isolate state between execution contexts — a platform for containment based deadlock recovery. We presented techniques that transparently and efficiently detect, isolate, and privatize memory updates. We presented semantics for propagating memory updates, and preserving program correctness while still permitting a containment based deadlock recovery. We addressed the challenges of privatization and presented techniques to address potential side-effects. We presented an efficient algorithm to detect deadlocks without the presence of an external agent. We presented techniques to provide granular deadlock recovery — rolling back the program state to only the offending lock as opposed to rolling the program state all the way to the start of the outermost critical section.

The culmination of these techniques and algorithms is **a pure runtime approach** (discussed in Chapter 2) for transparent deadlock detection and recovery in POSIX threaded applications written in type-unsafe languages such as C and C++. The techniques we have implemented enable such applications to run unmodified and these techniques do not require any modifications to the compiler or the operating system. We discussed the design, architecture, and limitations of

our approach. We performed a comprehensive performance analysis of our proposed approach. Our results indicate that our approach performs reasonably well even in the presence of fine-grain locking.

Leveraging some of the lessons learnt from this work (Chapter 2), we presented techniques to address its limitations, and broaden our impact. We presented reachability and flow analysis techniques, and opportunistic optimizations to derive the scope of critical sections. We presented an efficient runtime shadowing technique to provide isolation of memory updates. We presented transparent recovery techniques for deadlocks, priority inversion, and live-locks that perform rollback aware memory management, and I/O within critical sections. We discussed techniques for detecting asymmetric write-write data-races, priority inversion, order violations, live-locks, and asynchronous signaling.

Collectively, these techniques present **a program analysis and runtime approach** (discussed in Chapter 3) that can be used on production systems to eliminate deadlocks, and concurrency bugs affecting lock based codes. Neither of our approaches (Chapters 2 and 3) encounter any false positives or false negatives in detecting deadlocks. The program analysis and runtime approach also does not require any modifications to the application source-code, compiler, or the operating system. We presented the design, implementation, experimental evaluation, and limitations. Our experimental results indicate that our approach achieves its core goal of deadlock elimination at a performance level that enables widespread adoption. Our approach scales very well and supports applications that acquire billions of locks with lock rates of over hundreds of millions of locks/sec.

We believe that by providing usable and efficient approaches for **safe concurrent execution** of threaded codes, we provide a critical tool to programmers designing, implementing, and debugging complex applications for emerging many-core platforms. More broadly, this research work will impact and assist in improving the productivity of application developers.

Safe Concurrent Programming

While some applications are amenable to traditional parallelization techniques, we also have a large body of applications that are intrinsically unsuitable for mainstream parallelization techniques due to execution order dependencies (control and data) and sensitivity to input data. These applications do not scale well, leaving several cores idle. Therefore, an important dilemma facing computer science researchers is how to realize the potential of many core architectures and improve the performance of such hard-to-parallelize applications. A second main thrust of this dissertation addresses these challenges. Our observation here is that speculative execution at coarse granularities (e.g., code-blocks, methods, algorithms) offers a promising alternative for exploiting parallelism in many hard-to-parallelize applications on multicore architectures. This dissertation presents a simple programming framework to leverage **coarse-grain speculative parallelism** (discussed in Chapter 4) in otherwise hard-to-parallelize applications.

Realizing a safe concurrent programming environment relying on speculative parallelism while delivering all the desired elements (portability, scalability, usability, and efficiency) required to achieve wide-spread adoption is a difficult task, to say the least. The challenges include the following.

- Writing correct concurrent programs is extremely challenging for reasons described above. Hence, we do not want to burden the programmers with the subtleties of concurrent programming and the low level details of threading primitives to create speculative control flows, manage rollbacks, and perform recovery actions in the event of mis-speculations. Additionally, we do want to burden the programmer with explicitly having to manage name-space isolation among speculations. We want the programmer to focus on the problem at hand — achieving performance by leveraging course grain speculative execution. Hence, we need an efficient runtime system to handle these aspects transparently on behalf of the programmer, and robust programming semantics that preclude concurrency bugs.

- To support a wide variety of problem-solving or algorithmic strategies, we need expressive evaluation criterion for speculative execution that go beyond time to solution to include

arbitrary quality of solution. Additionally, we do not want to deprive already parallelizable (the ones capable of leveraging traditional techniques for parallelism) applications also from benefitting from speculative parallelism. Hence, we need simple (yet powerful) and expressive semantics to efficiently exploit coarse-grain speculative parallelism.

- To make a broader impact, in the context of high-performance computing applications, where the OpenMP threading model is widely prevalent, we need to make speculation a first class parallelization technique. We need well-defined semantics and extensions that naturally extend speculation into an OpenMP context. We should accomplish this without having to confine to a particular language, compiler and/or its runtime libraries, i.e., the speculation framework should be compatible with existing compiler infrastructures.

This dissertation focusses on addressing these challenges. In this dissertation we analyzed these issues in detail and presented techniques to address them.

We presented an execution model for speculative parallelism that provides the ability to selectively share and isolate state between execution contexts — a platform for composing speculations, and providing transparent name-space isolation. We presented techniques to transparently and efficiently (a) create, instantiate, and destroy speculative control flows, (b) perform transparent name-space isolation, (c) provide isolation, track data accesses for each speculation, (d) commit the memory updates of successful speculations, and (e) recover from memory side-effects of any mispredictions. We presented a powerful programming model with expressive semantics, and expressive evaluation criteria (temporal and qualitative) to exploit coarse-grain speculative parallelism. We also presented pragmas for composing speculations in the OpenMP threading model. To the best of our knowledge, our approach is the first to provide support for exploiting coarse-grain speculative parallelism in OpenMP based applications. Furthermore, we were able to accomplish this without sacrificing portability — we require no modifications to the complier.

The culmination of these techniques is a **safe concurrent programming** framework (discussed in Chapter 4) to improve performance of hard-to-parallelize and/or highly input dependent ap-

plications via coarse-grain speculative parallelism. Our performance evaluation using real world applications shows that our framework (a) is robust in the presence of performance variations or failure, and (b) achieves significant speedup over statically chosen alternatives with modest runtime overhead. Speculative execution at coarse granularities (e.g., code-blocks, methods, algorithms) is a promising alternative for exploiting parallelism on multi and many core architectures.

Looking Ahead

We have discussed interesting research directions throughout this dissertation and presented the scope for future work in Sections 3.7 and 4.6. We have mentioned several ways in which the systems we have designed and built (principally Sammati, Serenity, and Anumita) can be extended and applied. Although these extensions will broaden the scope and impact of our contributions, we believe that even more ambitious approaches to exploiting concurrency should be investigated. Presently, in type-unsafe languages such as C and C++, threads are not a part of the language. Consequently, concurrency control is also not a part of the language. As a result, the compiler is unaware that it is generating concurrent code and provides limited assistance to the programmer in developing robust parallel code [13]. We believe that simple language/compiler extensions can go a long way in improving the overall process of developing and debugging concurrent code. For instance, program variables can be either local or global depending on the scope of their declaration. In such a model, the entire global data is exposed to potential data races since threads share the virtual address space. The onus is now on the programmer to carefully reason about the necessary serialization required to prevent concurrent accesses to global data and avoid potential data races.

The fundamental problem here is — declaring data as global and later (during the development process) making sure that all concurrent accesses to that data are protected is a two-fold process. We believe that while writing concurrent code there is a disconnect between these two related processes. If programmers inadvertently forget to protect data (that ought to have been synchronized) at some program point, then it could result in a data race. Data race detection tools typically assume that any access to the global data could result in a potential data race. In essence, neither the language

nor its tool chain provides a safety net to the programmer. The programmer is left to either sift through the numerous false positives or run the code in good faith and attempt to debug and fix the problems as they arise.

Simple program annotations in the form of language extensions or variable attributes can enable the programmer to write robust concurrent code. These extensions allow the programmer to explicitly tag data that should be serialized. Unserialized or improperly serialized accesses to such data can then be caught by the compiler, resulting in a compilation error. Despite these measures bugs will occur — as Alan Perlis said, "*There are two ways to write error-free programs; only the third one works.*" [69]. The rest of the tool chain (e.g., runtime) should detect these errors and could potentially treat them as runtime exceptions (e.g., conflict exceptions [19] [55]).

Ultimately, programmers must be well equipped with a powerful tool chain that (a) prevents most concurrency bugs from happening in the first place, and (b) handles the remaining bugs appropriately at runtime. Looking forward, to effectively manage concurrency and enable programmers to develop robust parallel code we need support from all layers of the systems software stack.

Bibliography

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[2] Sarita V. Adve and Hans-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *Communications of the ACM*, 53:90–101, August 2010.

[3] Rahul Agarwal and Scott D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *PADTAD '06: Proceedings of the 2006 workshop on Parallel and Distributed Systems: Testing and Debugging*, pages 51–60. ACM, 2006.

[4] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, Honghui Lu, R. Rajamony, Weimin Yu, and W. Zwaenepoel. Treadmarks: shared memory computing on networks of workstations. *Computer*, 29(2):18 –28, feb 1996.

[5] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM.

[6] R. Barrett, M. Berry, J. Dongarra, V. Eijkhout, and C. Romine. Algorithmic bombardment for the iterative solution of linear systems: A poly-iterative approach. *Jnl. of Computational & Appl. Math.*, 74:91–110, 1996.

[7] Saddek Bensalem, Jean-Claude Fernandez, Klaus Havelund, and Laurent Mounier. Confirmation of deadlock potentials detected by runtime analysis. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, PADTAD '06, pages 41–50, New York, NY, USA, 2006. ACM.

[8] Saddek Bensalem and Klaus Havelund. Scalable deadlock analysis of multi-threaded programs. In *PADTAD '05: Proceedings of the Parallel and Distributed Systems: Testing and Debugging*, volume 1. Springer-Verlag., 2005.

[9] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: safe multithreaded programming for C/C++. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications*, pages 81–96. ACM, 2009.

[10] S. Bhowmick, L. C. McInnes, B. Norris, and P. Raghavan. The role of multi-method linear solvers in pde-based simulations. In *ICCSA (1)*, pages 828–839, 2003.

[11] Anasua Bhowmik and Manoj Franklin. A general compiler framework for speculative multithreading. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 99–108, New York, NY, USA, 2002. ACM.

[12] Christian Bienia and Kai Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[13] Hans-J. Boehm. Threads Cannot be Implemented As a Library. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '05, pages 261–268, New York, NY, USA, 2005. ACM.

[14] Boehm, Hans-J. and Adve, Sarita V. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI 2008, pages 68–78, New York, NY, USA, 2008. ACM.

[15] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230. ACM, 2002.

[16] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. *SIGARCH Computer Architecture News*, 38(1):167–178, 2010.

[17] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP '91, pages 152–164, New York, NY, USA, 1991. ACM.

[18] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *ACM Queue*, 6(5):46–58, 2008.

[19] Luis Ceze, Joseph Devietti, Brandon Lucia, and Shaz Qadeer. A case for system support for concurrency exceptions. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 6–6, Berkeley, CA, USA, 2009. USENIX Association.

[20] Tian Chen, Min Feng, and Rajiv Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *PLDI '10: Proceedings of ACM SIGPLAN 2010 conference on Programming Language Design and Implementation*, volume 45, pages 62–73, New York, NY, USA, 2010. ACM.

[21] Romain Cledat, Tushar Kumar, Jaswanth Sreeram, and Santosh Pande. Opportunistic Computing: A New Paradigm for Scalable Realism on Many-Cores. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 5–5, Berkeley, CA, USA, 2009. USENIX Association.

[22] E Christopher Lewis Colin Blundell and Martin Milo. Deconstructing Transactional Semantics: The Subtleties of Atomicity. *In WDDD '05: Proc. 4th workshop on duplicating, deconstructing and debunking*, pages 48–55, 2005.

[23] E Christopher Lewis Colin Blundell and Martin Milo. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2):17, 2006.

[24] Deadlock bug with mutex. SQLite-3.3.3. `http://www.sqlite.org/src/info/a6c30be214`, Aug 16 2012.

[25] DIMACS. Discrete Mathematics and Theoretical Computer Science, A National Science Foundation Science and Technology Center. `http://dimacs.rutgers.edu/`, January 2013.

[26] Chen Ding, Xipeng Shen, Kirk Kelsey, Chris Tice, Ruke Huang, and Chengliang Zhang. Software behavior oriented parallelization. In *PLDI '07: Proceedings of ACM SIGPLAN 2007 conference on Programming Language Design and Implementation*, volume 42, pages 223–234, New York, NY, USA, 2007. ACM.

[27] Doug Lea. A memory allocator. `http://g.oswego.edu/dl/html/malloc.html`, January 2012.

[28] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded java programs. In *Concurrency and Computation: Practice and Experience*, volume 15, pages 485–499. USENIX, 2008.

[29] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. *SIGOPS Operating Systems Review*, 37(5):237–252, 2003.

[30] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245. ACM, 2002.

[31] Prodromos Gerakios, Nikolaos Papaspyrou, Konstantinos Sagonas, and Panagiotis Vekris. Dynamic deadlock avoidance in systems code using statically inferred effects. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, PLOS '11, pages 5:1–5:5, New York, NY, USA, 2011. ACM.

[32] Prodromos Gerakios, Nikolaos Papaspyrou, and Kostis Sagonas. A type and effect system for deadlock avoidance in low-level languages. In *Proceedings of the 7th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '11, pages 15–28, New York, NY, USA, 2011. ACM.

[33] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, volume 38, pages 388–402. ACM, 2003.

[34] Jerry J. Harrow. Runtime checking of multithreaded applications with visual threads. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 331–342. Springer-Verlag, 2000.

[35] Klaus Havelund. Using runtime analysis to guide model checking of java programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 245–264. Springer-Verlag, 2000.

[36] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Computer Architecture News*, 21(2):289–300, 1993.

[37] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. McRT-Malloc: a scalable transactional memory allocator. In *ISMM '06: Proceedings of the 5th International Symposium on Memory Management*, pages 74–83. ACM, 2006.

[38] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.

[39] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 389–400, New York, NY, USA, 2011. ACM.

[40] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *PLDI '04: Proceedings of ACM SIGPLAN 2004 conference on Programming Language Design and Implementation*, volume 39, pages 59–70, New York, NY, USA, 2004. ACM.

[41] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 205–214, New York, NY, USA, 2007. ACM.

[42] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 327–336, New York, NY, USA, 2010. ACM.

[43] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 110–120. ACM, 2009.

[44] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 110–120. ACM, 2009.

[45] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *In Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2008.

[46] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association.

[47] Kirk Kelsey, Tongxin Bai, Chen Ding, and Chengliang Zhang. Fast Track: A Software System for Speculative Program Optimization. In *CGO '09: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 157–168, Washington, DC, USA, 2009. IEEE Computer Society.

[48] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How much Parallelism is There in Irregular Applications? In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 3–14, New York, NY, USA, 2009. ACM.

[49] Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Optimistic Parallelism Benefits from Data Partitioning. In *ASPLOS XIII: Proceedings of the 13th International conference on Architectural Support for Programming Languages and Operating Systems*, volume 36, pages 233–243, New York, NY, USA, 2008. ACM.

[50] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic Parallelism Requires Abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 211–222, New York, NY, USA, 2007. ACM.

[51] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, 2004.

[52] Tong Li, Carla S. Ellis, Alvin R. Lebeck, and Daniel J. Sorin. Pulse: a dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 3–3, 2005.

[53] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: a TLS compiler that exploits program structure. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 158–167, New York, NY, USA, 2006. ACM.

[54] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS XIII: Proceedings of the 13th International conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–339. ACM, 2008.

[55] Brandon Lucia, Luis Ceze, Karin Strauss, Shaz Qadeer, and Hans-J. Boehm. Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 210–221, New York, NY, USA, 2010. ACM.

[56] Yangchun Luo, Venkatesan Packirisamy, Wei-Chung Hsu, Antonia Zhai, Nikhil Mungre, and Ankit Tarkas. Dynamic performance tuning for speculative threads. In *ISCA '09: Proceedings of the 22nd annual International Symposium on Computer Architecture*, volume 37, pages 462–473, New York, NY, USA, 2009. ACM.

[57] Marco Pagliari. Graphcol: Graph Coloring Heuristic Tool. `http://www.cs.sunysb.edu/~algorith/implement/graphcol/implement.shtml`, January 2013.

[58] Pedro Marcuello and Antonio González. Thread-Spawning Schemes for Speculative Multithreading. In *HPCA '02: Proceedings of the 8th International Symposium on High-*

Performance Computer Architecture, page 55, Washington, DC, USA, 2002. IEEE Computer Society.

[59] Paul E. McKenney, Maged M. Michael, and Jonathan Walpole. Why the grass may not be greener on the other side: a comparison of locking vs. transactional memory. In *Proceedings of the 4th workshop on Programming languages and operating systems*, PLOS '07, pages 6:1–6:5, New York, NY, USA, 2007. ACM.

[60] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Bratin Saha, and Adam Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 314–325, New York, NY, USA, 2008. ACM.

[61] Multithreaded grep. Sun Microsystems, Multithreaded Programming Guide. `http://docs.oracle.com/cd/E19455-01/806-5257/6je9h034c/index.html`, April 13 2011.

[62] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM.

[63] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[64] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 386–396. IEEE Computer Society, 2009.

[65] Yang et al. Ni. Design and implementation of transactional constructs for C/C++. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems Languages and Applications*, pages 195–212. ACM, 2008.

[66] Open Source Sendmail and Milters. Sendmail Inc. `http://www.sendmail.com/sm/open_source/`, April 13 2011.

[67] Patterson, David A. and Hennessy, John L. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008.

[68] PBZIP2. Parallel BZIP2 (PBZIP2), Data Compression Software. `http://compression.ca/pbzip2/`, April 13 2011.

[69] Alan J. Perlis. Special feature: Epigrams on programming. *SIGPLAN Not.*, 17(9):7–13, September 1982.

[70] Prakash Prabhu, Ganesan Ramalingam, and Kapil Vaswani. Safe Programmable Speculative Parallelism. In *PLDI '10: Proceedings of ACM SIGPLAN 2010 conference on Programming Language Design and Implementation*, volume 45, pages 50–61, New York, NY, USA, 2010. ACM.

[71] C. Von Praun. Detecting synchronization defects in multi-threaded object-oriented programs. In *PhD Thesis*, 2004.

[72] Hari K. Pyla. Coarse-Grain Speculation for Emerging Processors. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '11, pages 217–218, New York, NY, USA, 2011. ACM.

[73] Hari K. Pyla. Composing Locks by Decomposing Deadlocks. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '11, pages 67–70, New York, NY, USA, 2011. ACM.

[74] Hari K. Pyla, Calvin Ribbens, and Srinidhi Varadarajan. Exploiting Coarse-Grain Speculative Parallelism. In *Proceedings of the 2011 ACM international conference on Object oriented*

programming systems languages and applications, OOPSLA '11, pages 555–574, New York, NY, USA, 2011. ACM.

[75] Hari K. Pyla and Srinidhi Varadarajan. Avoiding Deadlock Avoidance. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 75–86, 2010.

[76] Hari K. Pyla and Srinidhi Varadarajan. Transparent Runtime Deadlock Elimination. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 477–478, New York, NY, USA, 2012. ACM.

[77] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP '05: Proceedings of the twentieth ACM Symposium on Operating Systems Principles*, pages 235–248. ACM, 2005.

[78] Naren Ramakrishnan and Calvin J. Ribbens. Mining and visualizing recommendation spaces for elliptic pdes with continuous attributes. *ACM Trans. Math. Softw.*, 26(2):254–273, June 2000.

[79] Arun Raman, Hanjun Kim, Thomas R. Mason, Thomas B. Jablin, and David I. August. Speculative parallelization using software multi-threaded transactions. In *ASPLOS XV: Proceedings of the 15th International conference on Architectural Support for Programming Languages and Operating Systems*, volume 38, pages 65–76, New York, NY, USA, 2010. ACM.

[80] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. IEEE Computer Society, 2007.

[81] Paruj Ratanaworabhan, Martin Burtscher, Darko Kirovski, Benjamin Zorn, Rahul Nagpal, and Karthik Pattabiraman. Detecting and tolerating asymmetric races. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 173–184, New York, NY, USA, 2009. ACM.

[82] Lawrence Rauchwerger and David A. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *IEEE Transactions on Parallel Distributed Systems*, 10(2):160–180, 1999.

[83] J. R. Rice and R. F. Boisvert. *Solving Elliptic Problems Using ELLPACK*. Springer–Verlag, 1985.

[84] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 47–56, New York, NY, USA, 2010. ACM.

[85] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.

[86] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing, Boston, 1996.

[87] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 187–197. ACM, 2006.

[88] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[89] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 11–21, New York, NY, USA, 2008. ACM.

[90] Vivek K. Shanbhag. Deadlock-detection in java-library using static-analysis. *Asia-Pacific Software Engineering Conference*, 0:361–368, 2008.

[91] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of Distributed Computing*, pages 204–213. ACM, 1995.

[92] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. In *PLDI '07: Proceedings of ACM SIGPLAN 2007 conference on Programming Language Design and Implementation*, volume 42, pages 78–88. ACM, 2007.

[93] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of Distributed Computing*, pages 338–339. ACM, 2007.

[94] SPLASH-2. SPLASH-2 benchmark suite. `http://www.capsl.udel.edu/splash`, November 5 2011.

[95] Squid. Squid: Optimizing Web Delivery. `http://www.squid-cache.org/`, April 13 2011.

[96] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[97] Sun Microsystems. Lock_Lint - Static Data Race and Deadlock Detection Tool for C. `http://developers.sun.com/solaris/articles/locklint.html`, April 13 2011.

[98] Thomas Wang. Sorting Algorithm Examples. `http://www.concentric.net/~ttwang/sort/sort.htm`, January 2013.

[99] Chen Tian, Min Feng, Nagarajan Vijay, and Gupta Rajiv. Copy or Discard execution model for speculative parallelization on multicores. In *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 330–341, Washington, DC, USA, 2008. IEEE Computer Society.

[100] Oliver Trachsel and Thomas R. Gross. Supporting Application-Specific Speculation with Competitive Parallel Execution. In *3rd ISCA Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures*, PESPMA'10, 2010.

[101] Oliver Trachsel and Thomas R. Gross. Variant-based competitive Parallel Execution of Sequential Programs. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 197–206, New York, NY, USA, 2010. ACM.

[102] Valgrind. Helgrind: a thread error detector. `http://valgrind.org/docs/manual/hg-manual.html`, Feb 19 2012.

[103] Haris Volos, Andres Jaan Tack, Michael M. Swift, and Shan Lu. Applying transactional memory to concurrency bugs. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 211–222, New York, NY, USA, 2012. ACM.

[104] Christoph von Praun, Luis Ceze, and Calin Caşcaval. Implicit Parallelism with Ordered Transactions. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP 2007, pages 79–89, New York, NY, USA, 2007. ACM.

[105] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *In Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2008.

[106] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott Mahlke. The theory of deadlock avoidance via discrete control. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 252–263, New York, NY, USA, 2009. ACM.

[107] Amy Williams, William Thies, and Michael D. Ernst. Static deadlock detection for java libraries. In *ECOOP 2005 - Object-Oriented Programming*, pages 602–629, 2005.

[108] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8, 2010.

[109] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *SPAA '08: Proceedings of the twentieth annual Symposium on Parallelism in Algorithms and Architectures*, pages 265–274. ACM, 2008.