Deterministic Dynamic Deadlock Detection and Recovery

HARI K. PYLA and SRINIDHI VARADARAJAN, Virginia Tech

A core problem with concurrent programming using threads is the potential for deadlocks. Even well-written codes that spend an inordinate amount of effort in deadlock avoidance cannot always avoid deadlocks, particularly when the order of lock acquisitions is not known a priori. Furthermore, arbitrarily composing lock based codes may result in deadlock - one of the primary motivations for transactional memory. In this paper, we present a language independent runtime system (Sammati, *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. We implemented our runtime 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. Our performance evaluation of the Sammati's runtime with unmodified SPLASH, Phoenix and synthetic benchmark suites shows that it is capable of performing deadlock detection and recovery even when subject to extremely fine-grain locking (Barnes (640K locks/sec), FMM (265K locks/sec), and Water (70K locks/sec)). Additionally, we present compile time extensions to reduce the runtime overhead of Sammati. We find that Sammati is scalable, with speedup comparable to baseline execution with modest memory overhead for most applications.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.3.4 [**Programming Languages**]: Processors—*Run-time Environments*; D.3.4 [**Programming Languages**]: Processors—*Debuggers*

General Terms: Algorithms, Design, Languages, Measurement, Performance and Reliability

Additional Key Words and Phrases: Deadlock detection and recovery, concurrent programming, concurrency bugs, and runtime systems

ACM Reference Format:
Hari K. Pyla and Srinidhi Varadarajan 2012. Deterministic Dynamic Deadlock Detection and Recovery ACM Trans. Program. Lang. Syst. V, N, Article A (January YYYY), 44 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Making parallel programming easier is one of the most important challenges facing computer science researchers today. A primary motivation for the strong interest in concurrency is the emerging dominance of many-core architectures. Unfortunately, thread based concurrent programming is highly challenging due to the non-deterministic nature of thread execution, which complicates the detection and isolation of concurrency bugs [Lu et al. 2008] [Zhang et al. 2010].

Large software systems are developed by several hundreds to thousands of programmers, often spanning multiple teams, making it hard to enforce and maintain a strict coding discipline required to avoid concurrency bugs [Jula et al. 2008]. Additionally,

A preliminary version of this article was presented at the 19th ACM SIGARCH Conference on Parallel Architectures and Compilation Techniques (PACT'10).
Author's addresses: Hari K. Pyla, Department of Computer Science, Virginia Tech, Email: harip@vt.edu; Srinidhi Varadarajan, Department of Computer Science, Virginia Tech, Email: srinidhi@vt.edu.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

while several billions of dollars are spent each year by the software industry to ensure software quality and perform testing, unfortunately, exercising all possible paths and thread interleavings is still practically impossible. In practice, merely 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 bug manifested in the program code. Recently, when addressing a deadlock in Mozilla, the developers introduced another deadlock which took them them several months to a year to fix [Volos et al. 2012]. In a different Mozilla bug, the programmers introduced a data-race to address the actual deadlock [Volos et al. 2012]. Additionally, concurrency bugs are hard to reproduce and sometimes such bugs are hard to fix requiring a major software redesign. Furthermore, the patches developed to fix a bug are themselves error-prone (70% of the time in their first release) and they introduce new bugs [Jin et al. 2011]. On an average, a concurrency bug fix takes about 3 fixes (patches) before it is actually fixed [Lu et al. 2008; Jin et al. 2011]. Unless we find mechanisms to enable a large number of programmers, representing a wide array of applications, to use these parallel shared memory platforms effectively, the potential of many-core will go unrealized.

In practice, most concurrency bugs in multithreaded applications arise due to *data races* that occur due to improperly guarded accesses to memory and *deadlocks* that arise due to lack of canonical discipline and circular dependencies among locks. While bugs due to data races can be resolved by the use of appropriate synchronization, deadlocks require fairly complex deadlock avoidance techniques, which fail when the order of lock acquisitions is not known a priori [Pyla and Varadarajan 2010]. Fine-grain locking significantly exacerbates deadlock avoidance issues, to the point that it is generally eschewed in favor of simpler less performant locking models that are deadlock free. Furthermore, due to the potential for deadlocks, programmers cannot arbitrarily compose lock based codes without knowing the internal locking structure. Hence, rendering lock-based codes *non-composable*.

In this article we present **Sammati** (*agreement* in Sanskrit), a runtime system that is capable of *transparently* and *deterministically* detecting and recovering from deadlocks in multithreaded applications without requiring any annotations to application source code or recompiling/relinking phases. Since a large percentage of applications are based on weakly typed languages such as C and C++ that permit arbitrary pointer accesses, we eschewed a language based approach in favor of a pure runtime. We implemented Sammati as a pre-loadable library that overloads the standard POSIX threads (pthreads) interface and makes the acquisition of mutual exclusion locks a deadlock free operation. While this goal is in principle similar to transactional memory systems, there is a critical difference. Sammati preserves the mutual exclusion semantics (and more importantly its limitations) of existing lock based codes and does not provide any mechanisms to optimistically execute critical sections concurrently as in transactional memory.

We presented an initial prototype of Sammati in a conference paper [Pyla and Varadarajan 2010]. In this article, we describe several elements of the runtime system including privatization, semantics for propagating the privatized updates and deadlock detection and recovery algorithms. We perform a detailed study on the tradeoffs between the design and performance of the key components of the runtime system and present a comprehensive discussion on the engineering aspects and empirical analysis of the runtime. We present several optimizations to the runtime system and provide compile time extensions to reduce its runtime overhead, and improve its overall scalability. Additionally, we extend Sammati to address some of its limitations. The initial prototype [Pyla and Varadarajan 2010] of Sammati did not support applications that employed asynchronous synchronization (e.g., condition variables) within critical

sections. Furthermore, the original prototype of Sammati was unable to recover from deadlocks if a) the critical section involved, performed a disk I/O, and b) critical sections updated memory in the thread-local storage (TLS) region of the virtual address space. In this study we extend Sammati to address several of its limitations. Our objective in this study is to design a system that is capable of detecting and recovering from deadlocks in lock based codes written in type unsafe languages at a performance level required for wide-spread adoption.

The rest of the article is organized as follows. Section 2 presents an overview Sammati. Section 3 describes the design and architectural aspects of Sammati. Section 4 presents a comprehensive description of the implementation details of the runtime system and compile time extensions. Section 5 presents a detailed performance evaluation of Sammati's runtime system using the SPLASH [SPLASH-2 2012], Phoenix [Ranger et al. 2007], and synthetic benchmark suites. Section 6 discusses the limitations of a pure runtime approach and Section 7 presents compile time extensions to overcome the limitations of a pure runtime. Section 8 discusses the related work in the area of deadlock detection and recovery. Section 9 presents the directions for future work and Section 10 provides our conclusions.

2. OVERVIEW

In the remainder of this section, we present a brief overview of Sammati, outline its key elements and illustrate how Sammati detects and recovers from deadlocks.

Sammati operates by (a) associating memory updates with one or more locks guarding the updates and (b) 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 conceptually simple, this 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, finally (e) perform deadlock detection and recovery that deterministically eliminates deadlocks without either deadlocking itself or require an outside (external) agent.

Isolating memory updates: In this article we explore two approaches to detect and isolate memory updates within critical sections. First, in a pure runtime approach Sammati implements threads as processes, an idea first proposed by Emery Berger in Grace [Berger et al. 2009]. Since processes have distinct address spaces, by employing virtual memory page protection, we can detect the memory updates made within a critical section. Second, in contexts where the runtime cost of address space protection is unacceptable, we explore compile time analysis using the LLVM [Lattner and Adve 2004] compiler infrastructure to detect updates within critical sections. Sammati associates the updates with critical section(s) and then privatizes these updates from being visible to other concurrently executing threads until the release of a critical section. Section 3.1 presents a detailed description of these techniques.

Propagating memory updates: At the successful release of all locks protecting a critical section, the privatized memory updates within the critical section must to be made visible i.e., reconciled with other threads in the shared program address space. Providing isolation and propagating updates in the presence of nested locks, presents several additional challenges. Hence, in the context of nested locks we need to define

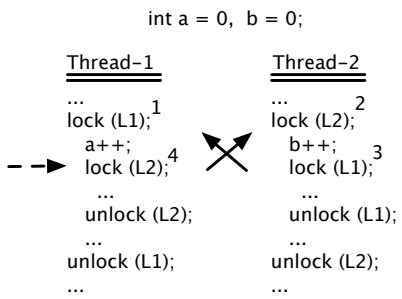


Fig. 1. Simple multithreaded program prone to deadlock.

semantics i.e., a set of visibility rules that preserve existing lock semantics, while still ensuring program correctness. Section 3.2 presents the semantics of propagating updates.

To commit the updates to the program address space (shared among threads), Sammati must first determine the exact set of bytes modified within a critical section. To accomplish this, Sammati computes a bit wise exclusive-or difference between the contents at the privatized version of the page and its copy (twin) that was saved prior to making any modifications to the page within the critical section. The resulting difference precisely identifies the modified bits within a critical section. This technique is inspired from distributed shared memory systems [Carter et al. 1991; Keleher et al. 1994]. The runtime system then updates the contents of the virtual address with the exclusive-or between the bytes it modified and the contents at the virtual address, thus making its updates visible to other threads. Section 4.6 presents the implementation details of propagation of memory updates.

Detecting deadlocks: Based on our observation that a thread may wait (block) on at most one lock, a *waits-for* graph is sufficient to detect deadlocks. Such a property of lock-based codes significantly simplifies deadlock detection, since the perspective of each thread, all deadlocks are single cycled deadlocks. We propose and implement an efficient deadlock detection algorithm with a time complexity of upper bound of O(n), where n is the number of threads. Section 3.3 discusses conditions necessary for deadlocks and presents the deadlock detection algorithm.

Recovering from deadlocks: On the acquisition of every lock, Sammati runs a single cycle deadlock detection algorithm. If a deadlock is detected, its deadlock elimination algorithm breaks the cycle by selecting a victim, rolls it back to the acquisition of the offending lock, and discards any memory updates. Since the containment ensures that memory updates from a critical section are not visible outside the critical section until a successful release, we simply restart the critical section to recover from the deadlock. Section 3.4 presents deadlock recovery algorithm.

Putting everything together: We now present a simple multithreaded program susceptible to deadlock (shown in Figure 1) to illustrate how memory isolation and privatization, deterministic deadlock detection and recovery algorithms of Sammati is capable of resolving the deadlock. Since thread interleaving is non-deterministic, Thread-1 may wait on lock L2 that is owned by Thread-2 and Thread-2 may wait on lock L1 held by Thread-1 (e.g., Thread-1 $\rightarrow$ Thread-2 $\parallel$ Thread-2 $\rightarrow$ Thread-1) resulting in a deadlock between the two threads. Initially, both the threads view the same global state i.e., $a = 0$ and $b = 0$. Sammati isolates the updates to variables a and b by the threads using address space privatization technique described above. Hence the update to a by thread-1 is not visible to thread-2 and similarly update to variable b to 1 by thread-2 is not made visible to thread-1. If a deadlock is detected, the runtime

system rolls back the updates of the offending lock (L1 in case of thread-1 and L2 in case of thread-2) and releases the lock to resolve the deadlock. In the absence of a deadlock, the updates to variable *a* by thread-1 will be made visible to other concurrently executing threads such as thread-2 on release of L1. Likewise the updates to *b* by thread-2 are made visible to thread-1 on release of L2.

3. 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.

3.1. 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, we need an execution model that *provides the ability to selectively isolate and share state between execution contexts.*

To implement containment for threaded codes, we employ a technique first proposed in Grace [Berger et al. 2009]. The key observation is that privatization can be implemented efficiently and transparently in a runtime environment if each thread had 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

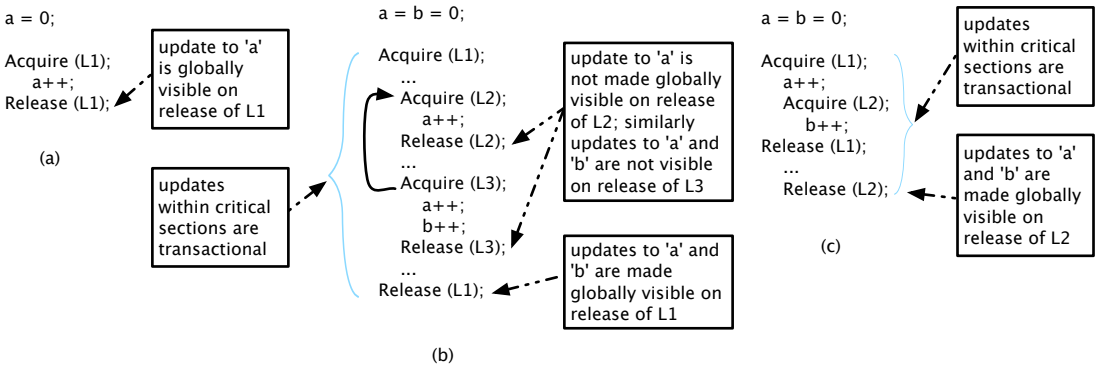


Fig. 2. Visibility rules for memory updates within a lock context.

that are semantically equivalent of 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 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 paper, we refer to these control flow constructs as *cords* to distinguish their properties from regular threads that operate within a single virtual address space.

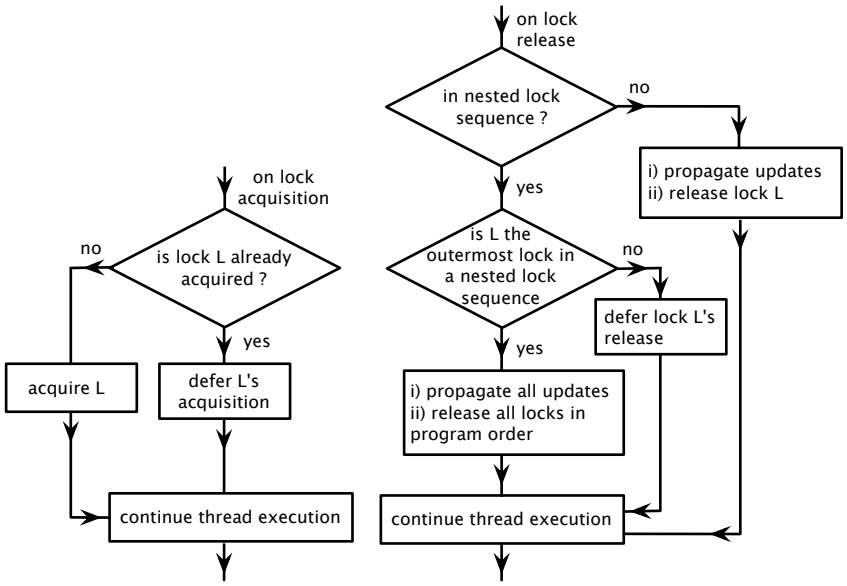
3.2. Semantics for Propagating Updates

Propagating the privatized memory updates made within critical section(s) while ensuring program correctness presents several challenges. In this section we describe the challenges using simple examples and we describe a set of *visibility rules* that define when memory updates made within a critical section are visible outside the critical section.

In the presence of a single lock (shown in Figure 2(a)) around a critical section, Sammati propagates the updates on the release of lock, e.g., L1 in Figure 2(a). However, nested locks are more complex. Consider the nested lock sequence shown in Figure 2(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(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. In Figure 3 we illustrate these 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.*



(a) Semantics of lock acquisitions (b) Semantics of lock releases

Fig. 3. Semantics of lock acquisitions (Figure 3 (a)), releases and propagation of memory updates (Figure 3 (b)).

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(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.

3.2.1. Subtleties of Visibility Rules. Our proposed semantics for propagating memory updates presents certain side-effects. In Figure 4 we present a few examples based on Blundell's et. al.,'s [Colin Blundell and Martin 2005; Colin Blundell and Milo 2006] work that exposes the subtleties of transactional memory and atomicity semantics. Consider the example shown in Figure 4 (a), Threads 1 and 2 run to completion in the absence of Sammati's privatization (discussed in Section 3.1). Recall privatization is used for containment of memory updates to facilitate recovery in the event of a deadlock. However, in the presence of privatization, since update to variable 'b' by thread-1 is not made visible until the release of the critical section L1, thread-2 is unaware of the update and finds the value of 'b' to be always false. Similarly update to 'a' by thread-2 is also not visible to thread-1, resulting in a deadlock between threads 1 and 2. Consequently, converting lock semantics to transactional regions could result in deadlocks [Colin Blundell and Martin 2005; Colin Blundell and Milo 2006; Menon et al. 2008; Wang and Wu 2010]. In Figure 4 (b) we present a similar example as Figure 4 (a) except that the ad-hoc synchronization by Thread-1 is not protected by a critical section.

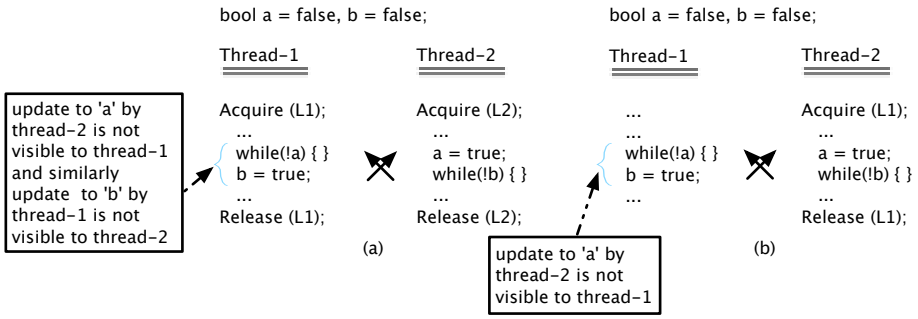


Fig. 4. Side-effects of privatization and transactional semantics could result in a deadlock.

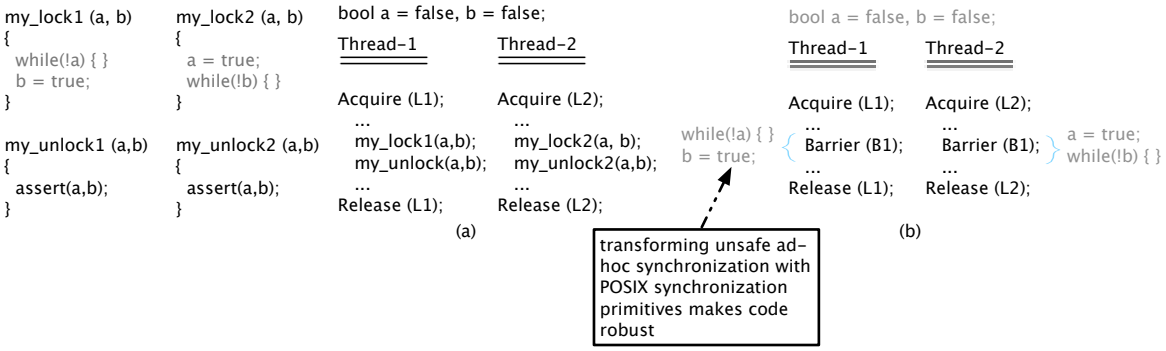


Fig. 5. Simple transformation makes example described in Figures 4 (a) and (b) circumvents the subtleties of Sammati's visibility rules and enables Sammati to successfully execute the program.

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 [Xiong et al. 2010]. Furthermore, such codes assume and rely on certain memory consistency guarantees to propagate the updates, for instance, update to '*a*' by thread-2 may not be propagated to other threads (e.g., thread-1) potentially running on other cores unless the program includes instructions to flush the memory updates immediately after the update. Consequently, the example code in Figures 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 Sammati.

Providing support for mixed locking regimes i.e., employing ad-hoc synchronization within critical sections conflicts with any runtime's system that supports transparent (containment through privatization without modifying source code) deadlock recovery. Consequently, Sammati is incapable of running such codes while ensuring the guarantee of a transparent deadlock recovery. In the presence of such codes to facilitate recovery, we recommend minor modifications to the program by replacing the ad-hoc synchronization with traditional POSIX spin-locks, condition variables and signaling, and barriers (shown in Figure 5 (b)).

In the event that the ad-hoc synchronization is an absolute necessity, the programmer can provide Sammati with a wrapper around ad-hoc synchronization in the form of a user-defined lock as shown in Figure 5 (a).

3.3. 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 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), a *waits-for* graph is sufficient to detect deadlocks. Such a property of lock based codes, significantly simplifies deadlock detection, since from a 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 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 and 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 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 at most wait on 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

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
```

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 by 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
```

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.

3.3.1. Minor Optimization. Sammati employs a conservative approach to detect deadlocks by running its deadlock detection algorithm at every lock acquisition in the program. Additionally, 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.

In this extension of Sammati, we implement a more relaxed approach while guaranteeing the same determinism in detecting and recovering from a deadlock. A necessary condition for a cycle to arise is that a thread has to *hold one or more locks* and be waiting on one or more locks. 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.

3.4. 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 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 by 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 6. A cord C acquires a lock L1, followed by lock L2 and updates a

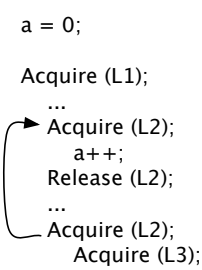


Fig. 6. In this example, if the acquisition of L3 results in a deadlock and the victim was L2, Sammati's deadlock recovery rolls back to the earliest acquisition of L2.

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.

3.4.1. Garbage Collection and 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* garbage collection 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.

3.4.2. Irrevocable disk and network I/O. While Sammati transparently eliminates memory side effects, non-idempotent operations such as memory management and I/O within a critical section pose additional challenges for recovery from concurrency bugs. Handling arbitrary non-idempotent I/O within the context of a restartable critical section is an open problem. Consider a critical section that contains a network write operation requesting a remote procedure invocation, followed by a read operation to get the results. To ensure restartability, if the write operation is delayed till the end of the critical section, the subsequent read operation will cause a self-deadlock. This problem exists in transactional memory systems as well and is quite serious, since it precludes atomic transactions from containing arbitrary non-idempotent network I/O. While techniques have been used to speculate on the read, to our knowledge, there is no known exact solution that does not require both ends of the network communication to be involved in the recovery. Sammati does not support recovery in the presence of network I/O.

To support disk I/O, Sammati overloads all disk I/O primitives including *read*, *write*, *fread*, *fwrite* etc., among others and it performs ordinary (non-lock context) I/O operations as they would otherwise since they cannot result in deadlocks. For instance threads 1 and 2 shown in Figure 7 (a) perform I/O concurrently. Similarly, Sammati

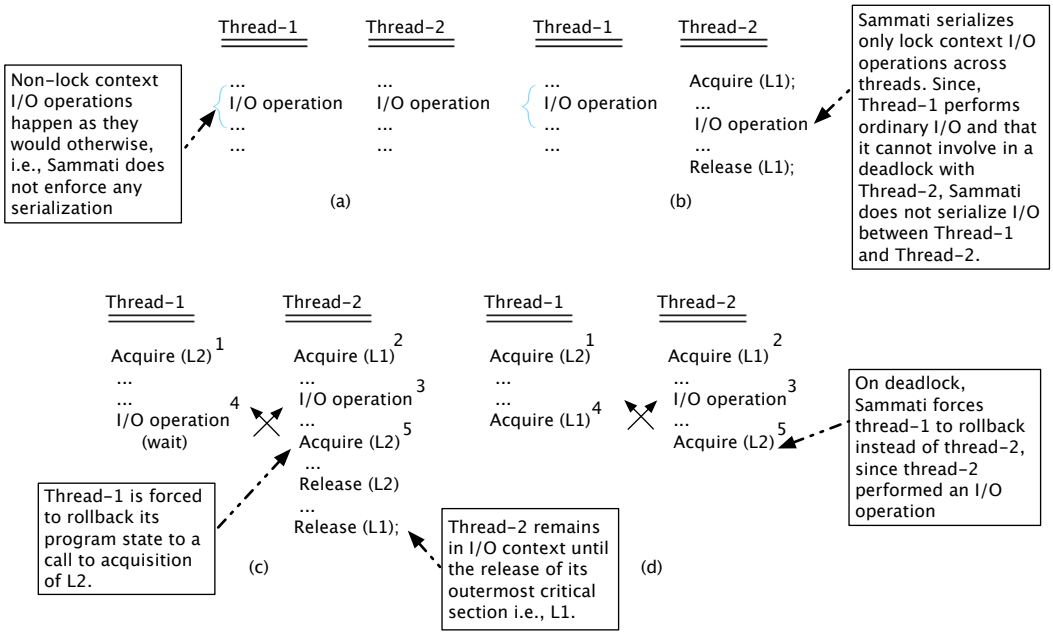


Fig. 7. Support for I/O.

does not serialize an I/O operation that happen between a lock context and an ordinary region. For instance, as shown in Figure 7 (b), Thread-1 which does not acquire any locks cannot result in a deadlock with Thread-2, hence, Sammati can safely perform I/O operations of threads 1 and 2 concurrently.

Sammati employs a conservative strategy to support I/O operations and prevents multiple threads from performing I/O within critical sections. If a deadlock arises between two threads that both performed I/O then it would be impossible to rollback either of them. Hence, Sammati allows only one thread to perform an I/O operation within a lock context. We refer to such a thread being in an I/O context. A thread remains in an I/O context until it releases its outmost critical section. Sammati does not allow any other concurrently executing thread to perform an I/O operation within a lock context until a thread exits its I/O context.

Consider the example shown in Figure 7 (c). Thread-2 first (in program order) performs I/O operation in a lock context (after acquiring L1). Hence, it enters the I/O context and remains in I/O context until it releases L1 (outermost lock), consequently, when Thread-1 attempts to perform an I/O operation it will wait until Thread-2 relinquishes its I/O context. Such a serialization could result in an implicit deadlock. For instance, Thread-2 on acquisition of L2, detects a deadlock between Thread-1 and Thread-2 since Thread-1 is waiting for thread-2 to exit the I/O context while Thread-2 will wait for Thread-1 to release L1. To recover from deadlock, Sammati forces the thread that did not perform an I/O operation, i.e., Thread-1 to rollback to acquisition of L2. This ensures Thread-2 to make progress. In Figure 7 (d) we present an example of an explicit deadlock involving Threads 1 and 2. Sammati forces Thread-1 (non-I/O context) to rollback instead of Thread-2, thus biasing the deadlock recovery algorithm to selecting victims that have not issued I/O.

4. PURE RUNTIME APPROACH

In this section we describe the key elements and implementation details of Sammati's runtime system.

4.1. Overview

We implemented Sammati's runtime as a shared library and that is pre-loaded by the dynamic linker (ld) before executing the binary. Sammati implements most of the POSIX threads interface, including thread creation, destruction, mutual exclusion locks, barriers and, condition variables.

4.2. 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 3.1) Sammati creates an illusion of a shared address space among processes. Sammati overloads POSIX thread create (*pthread_create*) call to create a cord.

4.2.1. 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 and 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.

4.2.2. 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 [Doug Lea 2012] dlmalloc 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 described in Section 4.2.1. 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 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.*

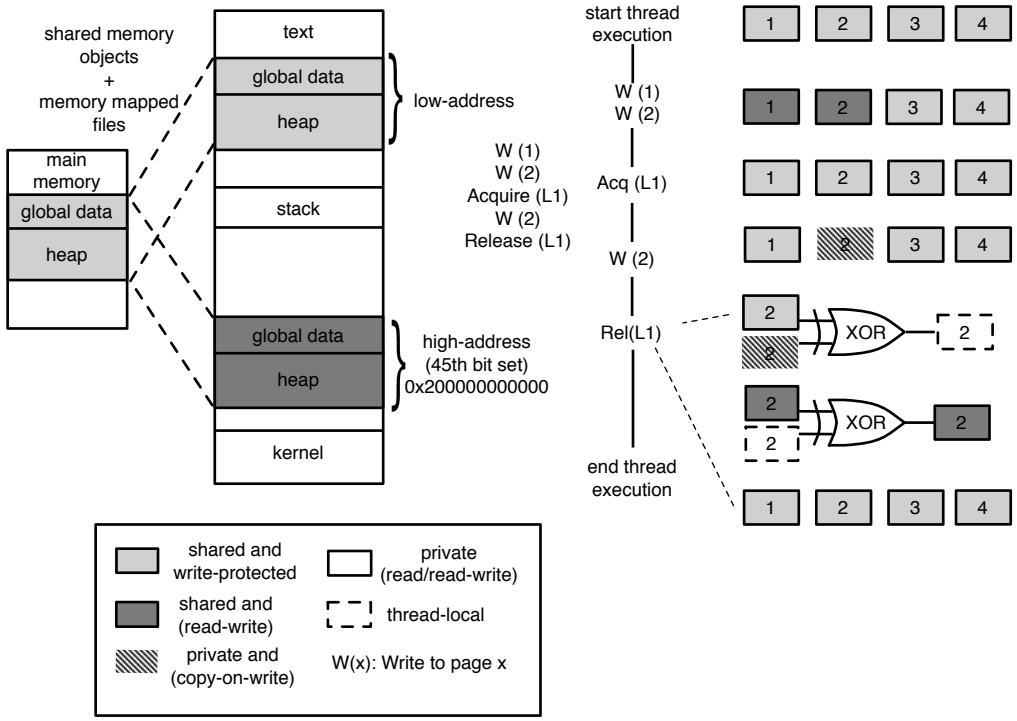


Fig. 8. (left) Illustrates the virtual memory address (VMA) layout of each process (cord). Sammati provides memory isolation by transforming threads to processes and uses shared memory objects and memory mapped files to share global variables and the process heap among cords. (right) Illustrates how the VMA is manipulated by Sammati with a simple example explained in text.

4.2.3. 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.

4.2.4. 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 8 illustrates the virtual address space layout of a cord.

4.3. 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 lock context until all the locks it acquired previously are released.

4.3.1. 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. We extend Sammati's implementation to include 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:* As opposed to variant-1, in this approach, 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 the (a) total size of the memory footprint and (b) the total number of locks acquired and finally (c) the number of updates performed within a critical section. The choice between the variants is dependent on these characteristics of the applications.

4.4. Isolating Memory Updates

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 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.

4.4.1. Weak Atomicity. Intuitively, Sammati implements a lazy privatization scheme that defers privatization to the instant when the first memory update occurs. We note that such a lazy privatization results in *weak atomicity* [Colin Blundell and Milo 2006]. While a conservative privatization of the entire address space at the acquisition of the first lock in a nested lock sequence results in *strong atomicity* – this was in fact our original solution, the measured runtime costs of this approach were far too high for applications with fine-grain locks. Since standard lock semantics of mutual exclusion locks do not require strong atomicity, we chose to implement the more efficient, lazy privatization approach and its resultant weak atomicity.

4.5. Preserving Synchronization Semantics

Sammati's runtime preserves the synchronization semantics of multi-threaded codes among cords (recall, implemented as processes) by transforming all

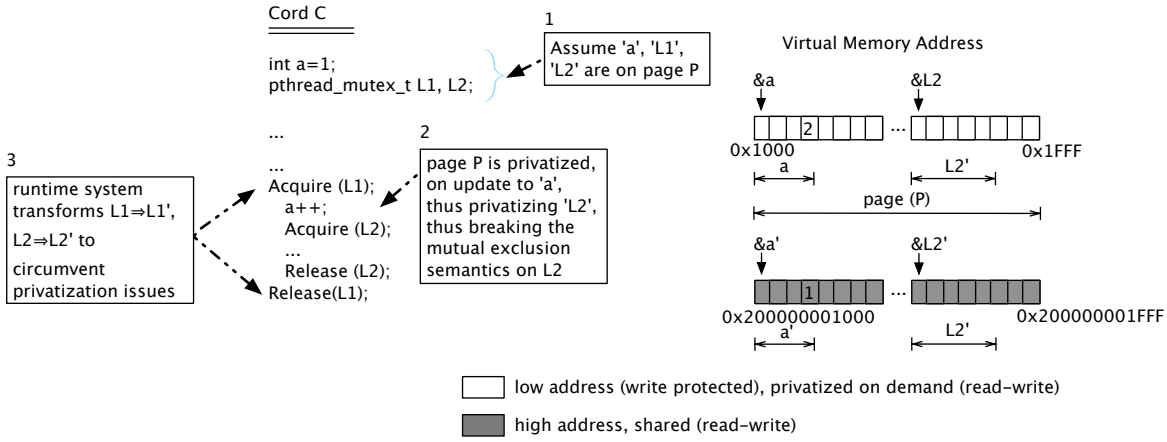


Fig. 9. Subtle issues with privatization

mutex exclusion locks (pthread_mutex_t), barriers (pthread_barrier_t), condition variables (pthread_cond_t) within the program to process-shared (PTHREAD_PROCESS_SHARED) locks, which enables their use among cords.

Recall that in 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 3.2).

A subtle side effect of memory isolation through privatization (discussed in Section 4.4) 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 (globals–statically declared or heap–dynamically allocated). For instance consider the example illustrated in Figure 9 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 heap (described in Section 4.2) 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 8). 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 9), 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.

4.6. 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 (TreadMarks [Amza et al. 1996]) 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 8) 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 4.8.

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 8 where a thread has four shared pages when it started 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 difference to the page in the shared high-address space.

4.7. 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 condition wait or semaphore semantics, Sammati overloads the wait primitive and treats it as a commit point, i.e., we perform inclusion as if the enclosing lock were being unlocked (including making all updates within the critical section visible) and then call the underlying POSIX condition wait to perform the operation. We employ a similar strategy to handle the signal primitive if it is protected by a lock i.e., called within a lock context. When another thread in the application signals the condi-

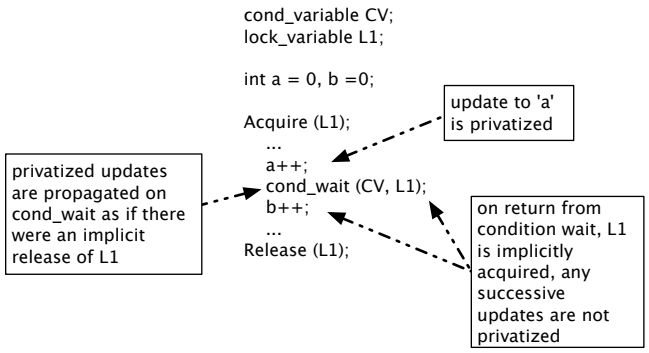


Fig. 10. Condition Variables

tion, a thread waiting on it atomically reacquires the lock, since it is simply using the underlying condition wait call. On resuming execution from a condition wait, a thread (cord) treats the resumption as a new lock acquisition (shown in Figure 10), however it does not privatize any updates (variable *b* in Figure 10).

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 wait/signal operation. Such semantics conflict with the privatization mechanism (discussed in Section 3.1) required to facilitate recovery, consequently Sammati while it is capable of detecting deadlocks in the presence of condition variables in nested locks, it is incapable of recovering deadlocks if they involve rolling back asynchronous events such as condition wait/signals.

4.8. 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 4.6) 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 happened 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 caused the conict.

5. EXPERIMENTAL EVALUATION

We evaluated Sammati's runtime performance using two POSIX threaded benchmark suites (SPLASH [SPLASH-2 2012], Phoenix [Ranger et al. 2007]), several synthetic benchmark suites. The SPLASH suite (described in Table I) 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

Table I. 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

Table II. 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

a few seconds to avoid the statistical noise from scheduler and (b) applications that compiled and run on a 64-bit machine. The Phoenix suite (described in Table II) 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 literature [Berger et al. 2009; Joshi et al. 2009; Pyla and Varadarajan 2010].

We chose the SPLASH and PHOENIX benchmarks for several reasons. First, their performance has been well studied in 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 for instance, Barnes (640K locks/sec), FMM (265K locks/sec) and Water (70K locks/sec) is extremely high. 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 tests Sammati to the extremes since the lock rate is one of the primary determinant 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.

5.1. 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, second, we ran the vanilla Pthread application and finally, we evaluated an approach based on Sammati's compile time instrumentation and runtime. 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 the each application using the UNIX *time* command.

5.2. Performance Analysis

We classify the 13 benchmarks from SPLASH and PHOENIX suites into 4 sets based on the lock acquisition rate of each application. We present a summary of this classi-

Table III. 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

fication in Table III. 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.

5.2.1. Runtime Overhead. Recall that we presented two approaches (a.k.a variants, described in Section 4.3) to detect memory updates performed within critical section. 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, privatizes 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 maps the page back to shared state and finally un write-protects the entire shared address space.

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

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 ordinary region. Similar to varaint-1, Sammati tracks the access faults to determine the pages modified in a lock context and creates a copy of the page, privatizes 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.

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

5.2.2. Set #1. Figure 11 illustrates the performance of Sammati and Pthreads for applications in set-1. As shown in Table IV, applications such as Barnes, FMM and Water acquire a reasonably large number of locks ($\approx 10^6$) and they have extremely high lock acquisition rate. For instance, Barnes acquires 640K locks/sec and 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.

Table IV. 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 V. 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 VI. 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

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 the address space protection. Barnes write-protects over 3×10^{10} pages. The high costs of address space protection prevent FMM and Water to make any meaningful progress resulting in a 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 V.

Sammati write-protects and un-write-protects an order magnitude (10^4) fewer pages under variant-2. Hence, variant-2 (shown in Figure 11) 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

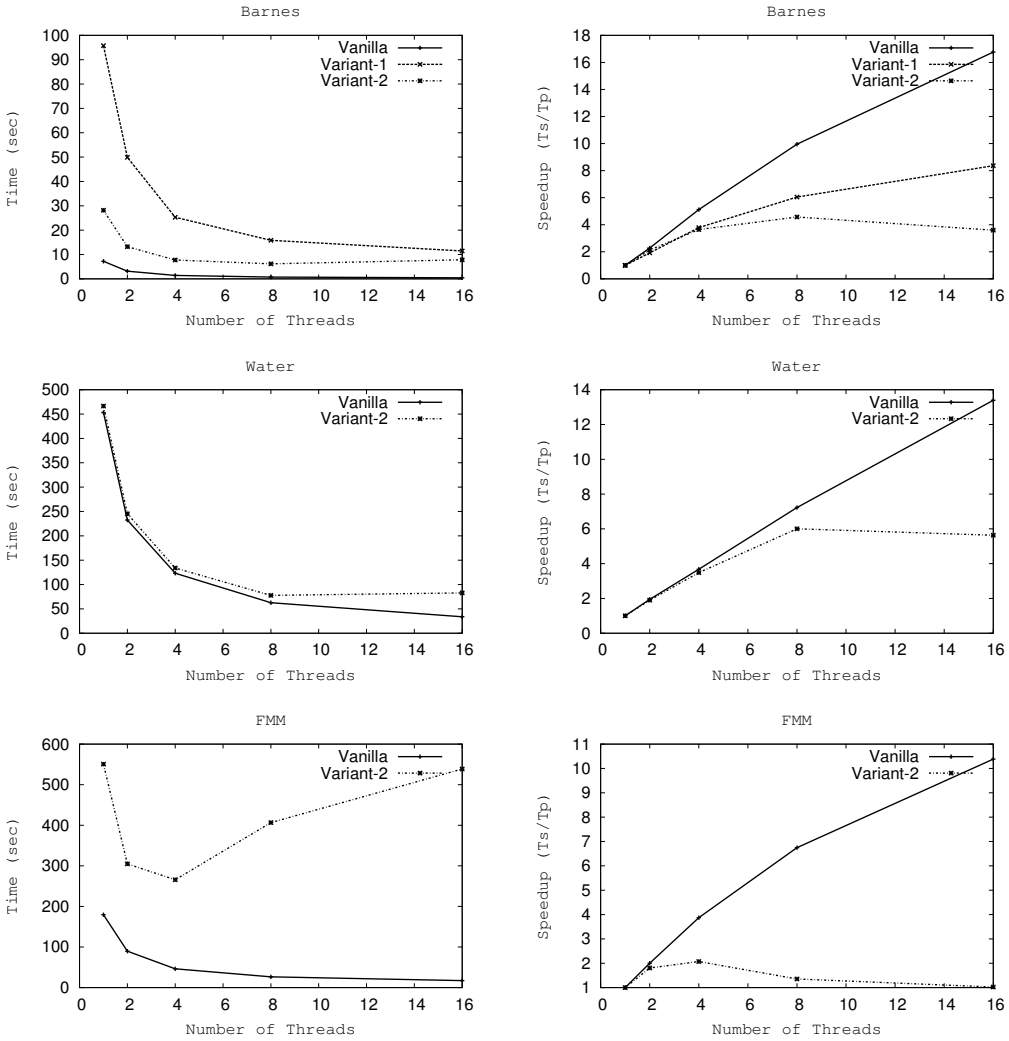


Fig. 11. Performance of applications from Set-1 (extremely high lock acquisition rate, typically 50K/sec - 700K/sec) with Sammati on a 16 core system.

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 is small (shown in Table VI), 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 IV) is identical to number of pages map shared (shown in Tables V and VI). This indicates that the overhead due to write-set in both the variants of Sammati 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

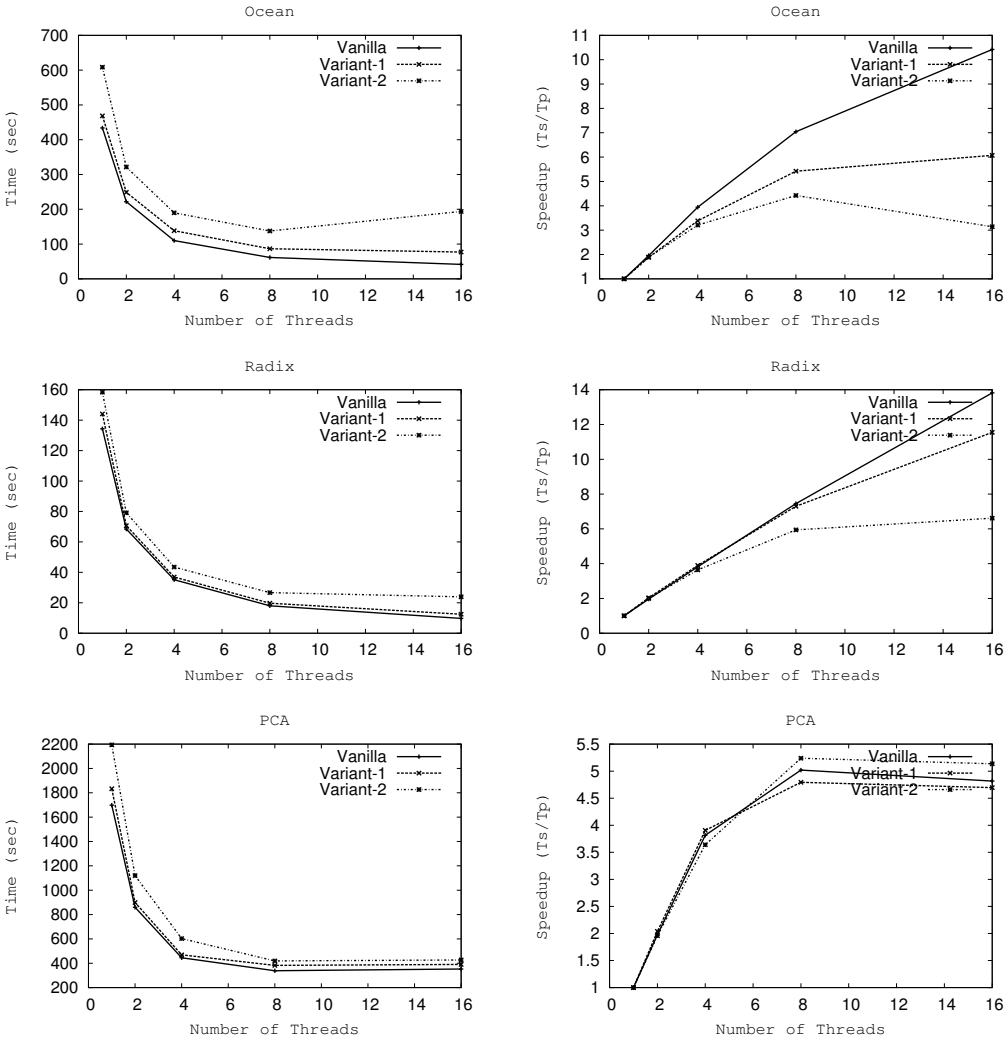


Fig. 12. Performance of applications from Set-2 (moderate lock acquisition rate, typically 10 - 50K/sec) with Sammati on a 16 core system.

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.

5.2.3. Set #2. Figure 12 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 X), and PCA acquires $10K$ locks. The lock acquisition rate (shown in Table VII) 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

Table VII. 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

Table VIII. 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 IX. 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 X. Profile of POSIX condition variables (signals/waits) in Radix.

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

very little data (≈ 500 pages for 16 threads) under lock context, however it incurs a noticeable reduction in speedup due to the cost of address space protection and unprotection. Ocean accesses a significant amount of data ($\approx 3.7 \times 10^7$ 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 an 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 a reduction in speedup.

Radix uses condition signals (shown in Table X) and 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 higher number of 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 VIII and IX. Consequently, both variants of Sammati scale well.

5.2.4. Set #3. The performance of applications in set-3 as shown in Figure 13 is comparable to native pthread execution. LU-CP and LU-NCP have relatively low number of ordinary accesses and lock context accesses (shown in Tables XII and XIII), thus, significantly reducing the overall cost of address space protection. Additionally, these applications also acquire few locks and have a significantly low lock acquisition rate (shown in Table XI) compared to applications from set1 and set2, thus resulting in trivial 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. Sammati write-protects and un-write-protects approximately 2 and 3 orders of magnitude more pages than LU-CP and LU-NCP in variants 1 and 2 respectively.

5.2.5. Set #4. Figure 14 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 4). The results show that performance of

Table XI. 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 XII. 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 XIII. 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

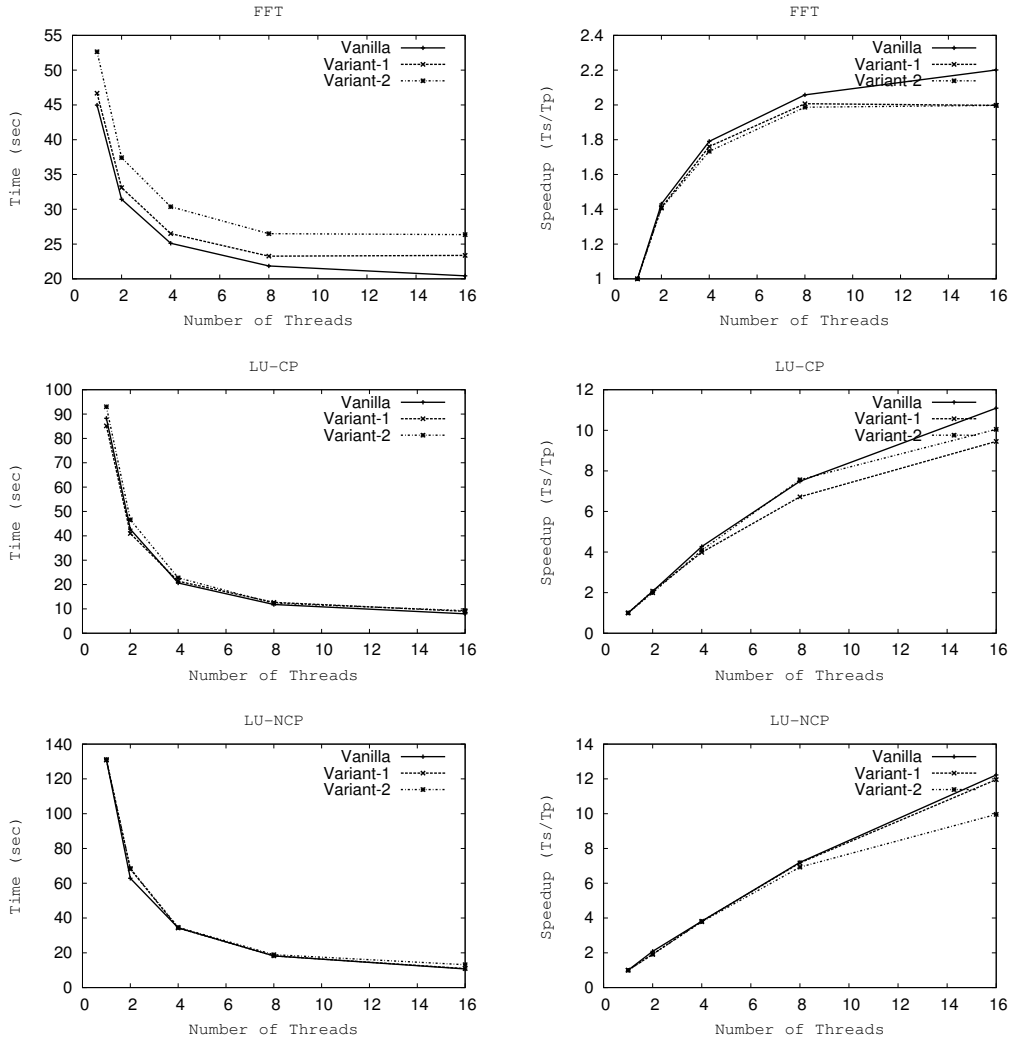


Fig. 13. Performance of applications from Set-3 (low lock acquisition rate, typically 0.5 - 10/sec) with Sammati on a 16 core system.

Sammati's cords incurs no performance overhead and Sammati's performance is comparable to native thread execution.

5.2.6. 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 discussed in Section 4.4. This overhead is caused by the Sammati's runtime maintaining twin copies of the page, which are then used to com-

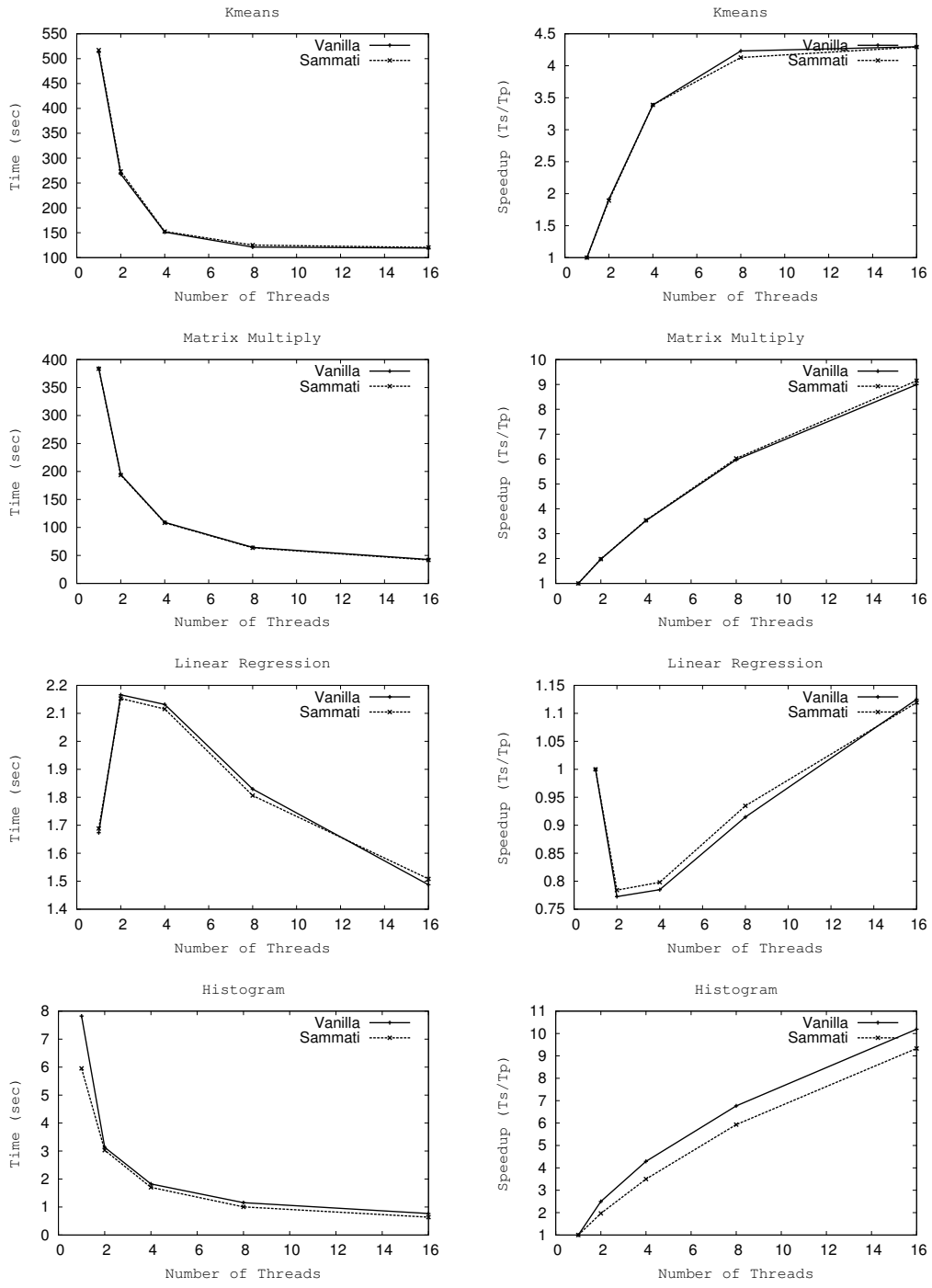


Fig. 14. Performance of applications from Set-4 (extremely low lock acquisition rate, typically 0 - 0.5/sec) with Sammati on a 16 core system.

pute the XOR differences during inclusion. Note that the twin pages are freed at the end of the critical section, i.e., hence this memory overhead is transient. The memory footprint of the 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 IV, VII and XI.

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 and the sum of the maximum number of twin pages for all cords was ≈ 16 pages. Barnes, FMM, and Water of set-1 modified 308, 49, 32 pages respectively for 16 cords. Consecutively, Barnes incurred the highest memory overhead of 1.203 MB ($308 \times 4K$) of memory, and FMM incurred a memory overhead of 196 K ($49 \times 4K$) and Water incurred an overhead of 128 K ($32 \times 4K$).

5.3. Deadlock Detection and Recovery

We created a synthetic benchmark suite that contained programs that are 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 [Pyla and Varadarajan 2010; Berger et al. 2009; Joshi et al. 2009; Joshi et al. 2010; Xiong et al. 2010]. In Figure 19 we present a few examples from our synthetic benchmark suite. In the first example, two threads (shown in Figure 19(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 resulted in a deterministic deadlock among the two threads. In Figure 19(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.

5.4. Summary

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, and finally 70K/sec for Water). 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 is almost identical to the performance of native pthreads applications and incurs no overhead. We show that is possible to efficiently design and implement a runtime system that is capable of transparently detecting and recovering from deadlocks.

6. LIMITATIONS OF PURE RUNTIME APPROACH

While Sammati's pure runtime approach can deterministically and transparently detecting deadlocks, it has limitations on recovering from deadlocks involving Thread-Local Storage (TLS) data *transparently* and *efficiently* while ensuring *portability*.

6.0.1. Thread-Local Storage (TLS). The original prototype of Sammati [Pyla and Varadarajan 2010] does not recover TLS data in the event of a deadlock. We briefly dis-

cuss the semantics of TLS data (denoted by *__thread* in type information of a variable) to explain how we extend Sammati to include support for TLS. 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 respectively in the object file (.o). The ld 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.

6.0.2. Address Space Protection Overhead. The high costs of address space protection and un-protection may preclude certain applications that have reasonably large memory footprint and lock rate to benefit from Sammati.

To address these limitations of a pure runtime approach, we extend Sammati to perform compile time analysis. The idea here is that by enforcing the following two design constraints a) availability of program source and b) recompilation and relinking of program source, which are otherwise relaxed in a pure runtime approach, we can perform program analysis and compile time instrumentation to avoid the high runtime costs of address space protection and provide a robust deadlock recovery.

7. COMPILE TIME EXTENSIONS

7.1. Overview

In order to mitigate the high runtime costs of virtual address space protection (two variants described in Section 4.3), Sammati leverages support from the LLVM [Lattner and Adve 2004] compiler infrastructure and instruments the store instructions in the application at compile time. This enables Sammati's runtime to track the program updates without the need to employ address space protection. We note that this extension breaks our initial goal to avoid application recompilation or the availability of program source, however we still preserve transparency by avoiding the programmer to make any modifications to application. The compile time extensions merely involve an additional step during the application compilation.

7.2. Instrumentation

Sammati's compile time infrastructure transforms the program source code to LLVM's intermediate representation (IR). Each *store* instruction in the program is instrumented with a hook (an inline function) as shown in Figure 15. The hook passes information to the runtime system about the target address (known precisely only at runtime) and, its length. The hook at runtime prior to performing the store instruction, checks if a cord is in a lock context. If so, it checks if the store is being performed on a stack then Sammati does not privatize the VMA page, instead it simply performs the store. Else, the store is being performed elsewhere in the address space (global data, heap , TLS) hence, Sammati creates a copy of the page (twin as discussed in Section 4.4), privatizes the page and performs the store. Sammati does not instrument the *load* instructions and they are executed as they would otherwise. To preserve program correctness and maintain efficiency, we generate highly optimized IR using (-O3)

LLVM's Intermediate Representation (IR) of Program Source

```
1000   %85 = load double* %84, align 8
1001   %86 = getelementptr inbounds double* %73, i64 %75
1002   store double %85, double* %86, align 8
```

Sammati's Compile Time Instrumentation of LLVM IR

```
1000   %95 = load double* %94, align 8
1001   %96 = getelementptr inbounds double* %83, i64 %85
1002   %97 = ptrtoint double* %96 to i64
1003   %98 = call i64 @store_double_instrument_fn(double %95, i64 %97, i32 8) nounwind
```

Inline expansion of Sammati's Instrumentation functions

```
1000   %110 = load double* %109, align 8
1001   %111 = getelementptr inbounds double* %98, i64 %100
1002   %112 = ptrtoint double* %111 to i64
1003   %113 = load i8* @lock_context, align 1     //begin inline expansion
1004   %114 = icmp eq i8 %113, 1
1005   br i1 %114, label %bb3.i14, label %bb.i13
1006
1007 bb.i13:                                           ; preds = %bb14
1008   %115 = inttoptr i64 %112 to double*
1009   store double %110, double* %115, align 8
1010   br label %store_double_instrument_fn.exit15
1011
1012 bb3.i14:                                          ; preds = %bb14
1013   %116 = call i64 @handle_lock_context_store_double(double %110, i64 %112, i32 8) nounwind
1014   br label %store_double_instrument_fn.exit15
1015
1016 store_double_instrument_fn.exit15:                ; preds = %bb.i13, %bb3.i14
1017   %.pre262 = load %struct.Global_Private** @gp, align 8
1018   br label %bb15
```

Fig. 15. Sammati's Compile time extension instruments store instructions by transforming program source to LLVM's intermediate representation. Sammati replaces each store instruction with a corresponding function call depending on the type (int, float, double) information. After instrumentation Sammati expands the inlined instrumentation function.

flag prior to instrumentation and we do not perform any optimizations on the IR after instrumentation.

7.3. Performance Analysis

We follow the classification described in Section 5.2 and summarized in Table III to analyze the performance of Sammati with compile time extensions. For each benchmark we measured several important characteristics including the pages privatized in lock context, the total number of store instructions instrumented, number of ordinary stores, number of lock context stores performed on global data and also on the stack.

7.3.1. Performance Overhead. Sammati instruments all the store instructions in a program and at runtime privatizes (at page granularity) all the lock context stores. Hence, the runtime overhead due to privatization and containment, propagation of memory updates, and finally the cost of un-privatizing memory updates is precisely identical to that in a pure runtime approach (variant-1 and variant-2).

Table XIV. Instrumentation profile of applications in Set-1

Benchmark	Threads	Total Locks	Lock Context pages (privatized)	Stores			
				Total instrumented	Ordinary	Lock Context	
						Global	Stack
Barnes	1	275216	842869	849330207	845624472	3575065	130670
	2	275227	842834	849331111	845625361	3575080	130670
	4	275293	843086	849332990	845627223	3575097	130670
	8	275366	842585	849335390	845629583	3575137	130670
	16	275507	842880	849339457	845633636	3575151	130670
FMM	1	4517437	4728860	4996393727	4887930878	108462849	0
	2	4521239	4736799	5003784172	4895309923	108474249	0
	4	4543901	4782105	5008168681	4899626461	108542220	0
	8	4559540	4813782	5010535972	4901946871	108589101	0
	16	4586949	4868541	5012392514	4903721375	108671139	0
Water	1	266568	270728	57240113261	57237716763	2396498	0
	2	533071	541391	57244921720	57240129309	4792411	0
	4	799902	812382	57252142804	57244954142	7188662	0
	8	1333564	1354364	57266585749	57254604585	11981164	0
	16	2400888	2438328	57295471121	57273904953	21566168	0

$$Overhead_{Sammati} \propto (\ cost \ of \ store \ instrumentation + \ write - set \times [\ cost \ of \ privatization \ + \ cost \ of \ creating \ copy + \ cost \ of \ propagating \ updates \ + \ cost \ of \ un-privatizing \ ])$$

7.3.2. Set #1. Figure 16 illustrates the performance of Sammati's runtime with compile time extensions for applications in set-1. As shown in Table XIV, Barnes, FMM and Water perform a reasonably large number of store instructions. For instance, Barnes performs approximately 8.49×10^8 stores, FMM performs 5×10^9 stores and Water performs approximately 5.7×10^{10} stores.

99.563% of stores performed by Barnes with 16 threads, are ordinary accesses and only a fraction (0.436%) of the total stores are performed within a lock context. Consequently, resulting in a noticeable overhead due to instrumentation. Variant-2 write-protects and un-write-protects approximately 3.5×10^5 pages (shown in Table VI) and incurs 1×10^6 access faults for Barnes. The instrumentation cost is almost identical the cost of address space protection, and we do not notice a significant benefit of instrumentation over variant-2.

FMM performs 97.83% ordinary stores and 2.16% stores in a lock context. Moreover, FMM write-protects and also un-write-protects approximately two order of magnitude more pages then Barnes. Hence, we find that instrumentation performs significantly better than variant-2. Additionally, compile time instrumentation allows Sammati to scale well compared to variant-2 due to the absence of address space protection and un-protection costs.

Water performs approximately 5.7×10^{10} stores and 99.96% of them are ordinary stores, thus we find a significant overhead due to instrumentation compared to native thread execution. Sammati's variant-2 incurs approximately (4.2×10^7) access faults. Since there are fewer access faults (recall tracked at the granularity of a $4K$ page), the overhead of instrumentation exceeds the overhead of tracking access faults in Water. Hence we find that variant-2 outperforms instrumentation.

7.3.3. Set #2. Figure 17 illustrates the performance of Sammati's runtime with compile time extensions for applications in set-2. Ocean, Radix, PCA perform relatively large number of store instructions (shown in Table XV). Ocean performs approximately 2.4×10^9 stores, Radix performs 2.6×10^9 stores and PCA performs approximately 1.0×10^8 stores and all the three applications perform over 99.99% of such stores outside a lock context (a.k.a ordinary stores).

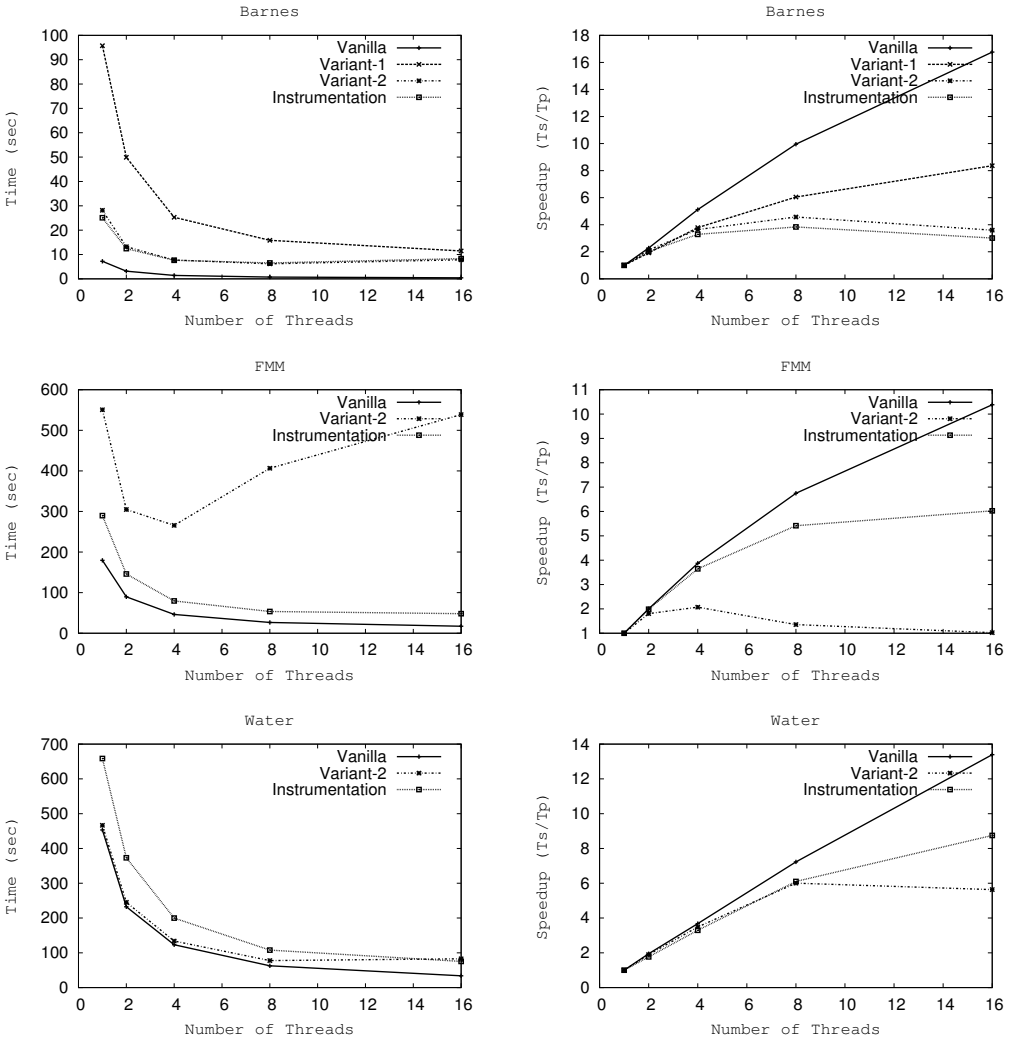


Fig. 16. Performance of applications from Set-1 (extremely high lock acquisition rate, typically 50K/sec - 700K/sec) with Sammati's runtime and compile time extensions on a 16 core system.

For Ocean, variant-2 protects and write-protects approximately 7.1×10^7 pages and incurs approximately, 3.6×10^7 access faults. Thus, variant-2 incurs significant overhead compared to instrumentation. Additionally, since variant-1 write-protects and un-write-protects a reasonably large number (approximately 2×10^{10}) pages, instrumentation outperforms variant-1.

Variant-2 for Radix incurs approximately 2×10^6 access faults and write-protects and un-write-protects approximately 3.6×10^6 pages, as a result, variant-2 incurs a noticeable overhead. Variant-1 incurs additional overhead than instrumentation and variant-2 due to its high address space protection cost (approximately 5.7×10^8 pages).

The performance overhead due to instrumentation is negligible in PCA and its performance is comparable to native thread execution. Instrumentation outperforms variant-1 due to the cost of address space protection ($\approx 1.9 \times 10^9$ pages). Variant-2 in-

Table XV. Instrumentation profile of applications in Set-2

Benchmark	Threads	Total Locks	Lock Context pages (privatized)	Stores			
				Total instrumented	Ordinary	Lock Context	
						Global	Stack
Ocean	1	173	173	24201025953	24201025780	173	0
	2	346	238	24205041443	24205041205	238	0
	4	692	286	24209295907	24209295621	286	0
	8	1384	475	24217329657	24217329182	475	0
	16	2768	530	24225844016	24225843486	530	0
Radix	1	4	4	2684373026	2684373022	4	0
	2	17	16	2684400690	2684400674	16	0
	4	43	32	2684462162	2684462130	32	0
	8	95	65	2684594322	2684594257	65	0
	16	199	132	2684874002	2684873870	132	0
PCA	1	10001	10001	100020001	100010000	10001	0
	2	10002	10002	100020002	100010000	10002	0
	4	10004	10004	100020004	100010000	10004	0
	8	10008	10008	100020008	100010000	10008	0
	16	10016	10016	100020016	100010000	10016	0

Table XVI. Instrumentation profile of applications in Set-3

Benchmark	Threads	Total Locks	Lock Context pages (privatized)	Stores			
				Total instrumented	Ordinary	Lock Context	
						Global	Stack
FFT	1	1	1	4559224862	4559224861	1	0
	2	2	2	4559224864	4559224862	2	0
	4	4	4	4559224868	4559224864	4	0
	8	8	8	4559224876	4559224868	8	0
	16	16	16	4559224892	4559224876	16	0
LU-CP	1	1	1	22940251171	22940251170	1	0
	2	2	2	22940251180	22940251178	2	0
	4	4	4	22940251198	22940251194	4	0
	8	8	8	22940251234	22940251226	8	0
	16	16	16	22940251306	22940251290	16	0
LU-NCP	1	1	1	22940054557	22940054556	1	0
	2	2	2	22940054562	22940054560	2	0
	4	4	4	22940054572	22940054568	4	0
	8	8	8	22940054592	22940054584	8	0
	16	16	16	22940054632	22940054616	16	0

curs over 3 orders of magnitude more access faults compared to variant-1, consequently variant-2 incurs a noticeable overhead compared to variant-1

7.3.4. Set #3. Figure 18 illustrates the performance of Sammati's runtime with compile time extensions for applications in set-3. All the applications in set-3 perform almost 100% of writes outside a lock context as shown in Table XVI.

FFT performs approximately 4.5×10^9 stores. Sammati write-protects and un-write-protects approximately 1.2×10^7 pages and 1.5×10^6 pages in variant-1 and variant-2 respectively. Additionally, variant-2 incurs an order magnitude (1.0×10^6) more access faults compared to variant-1. Hence, variant-2 incurs the most overhead compared to variant-1 and instrumentation. Due to the relatively large number of store instructions, the benefit of instrumentation is marginal in FFT compared to variant-2.

Both LU-CP and LU-NCP perform a large number ($\approx 2.29 \times 10^{10}$) of stores and they protect significantly fewer (5×10^5 in variants 1 and 2) pages, consequently, instrumentation incurs more overhead than the cost of protecting the address space in LU-CP and LU-NCP.

7.3.5. Memory Overhead. Sammati performs privatization and isolation at runtime, the memory overhead (discussed in Section 5.2.6) is invariant of compile time instru-

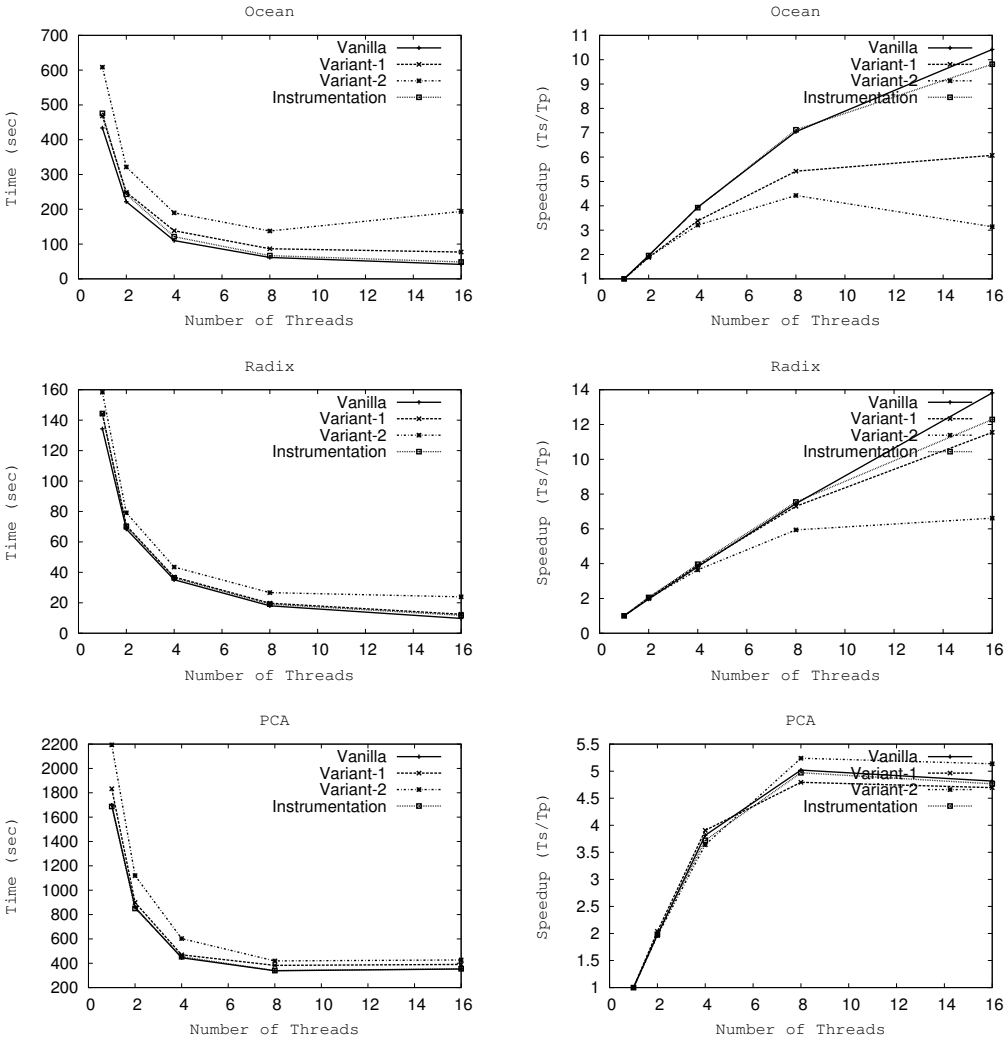


Fig. 17. Performance of applications from Set-2 (moderate lock acquisition rate, typically 10/sec - 50K/sec) with Sammati's runtime and compile time extensions on a 16 core system.

mentation and the memory overhead is precisely identical to Sammati's pure runtime approach.

7.4. Summary

Sammati's compile time extensions significantly reduced the runtime overhead of Sammati for some applications including FMM, Ocean, and PCA. The compile time extensions did have a negative impact on the performance of some applications (LU-CP, LU-NCP, and, FFT) where the cost of address space protection was lower than instrumenting the entire program. While there is certainly no clear winner between a pure runtime approach and a runtime approach with compile time extensions, we note that on the whole the performance of Sammati is impressive.

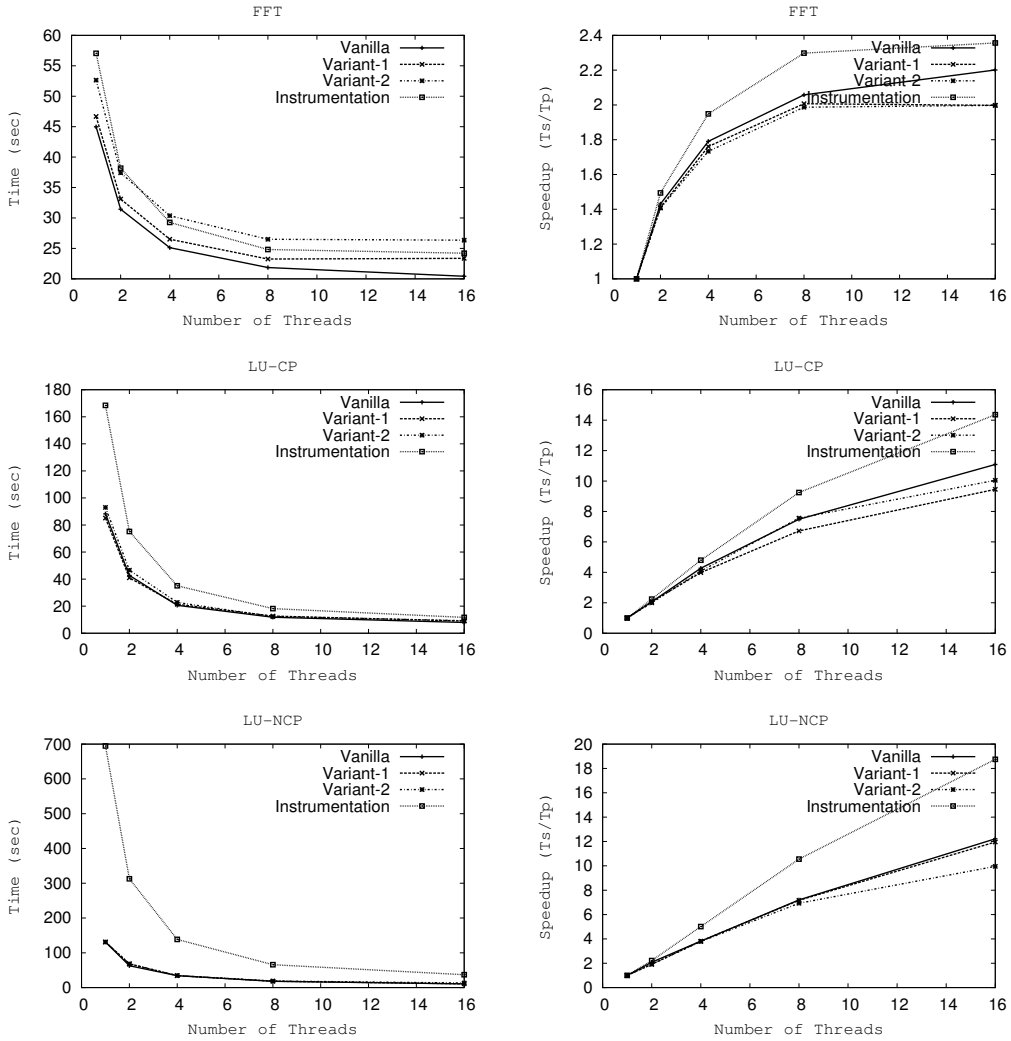


Fig. 18. Performance of applications from Set-3 (low lock acquisition rate, typically 0.5/sec - 10/sec) with Sammati's runtime and compile time extensions on a 16 core system.

8. RELATED WORK

This article is related to research in the areas of concurrency bug detection. 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 4.8). In this section we briefly discuss literature related to deadlock detection and recovery, and data races.

8.1. Deadlock Detection and Recovery

8.1.1. Static Analysis. Several systems [Boyapati et al. 2002; Engler and Ashcraft 2003; Flanagan et al. 2002; Sun Microsystems 2012; Naik et al. 2009; Savage et al. 1997; Shanbhag 2008; Williams et al. 2005; Praun 2004; Valgrind 2012; Gerakios et al. 2011] based on program analysis were proposed to determine deadlocks. While pro-

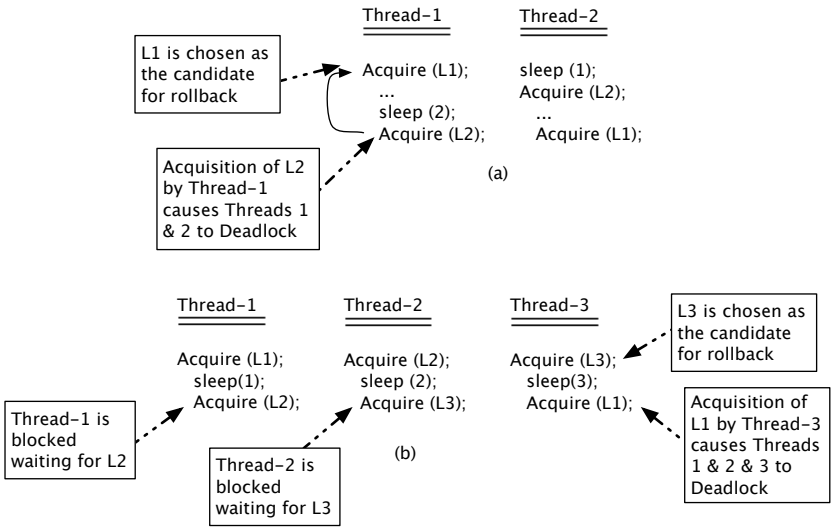


Fig. 19. (a) illustrates a simple deadlock between two threads due to cyclic lock acquisition. (b) depicts a more complex example of deadlock involving more than two threads.

gram analysis can identify certain deadlocks based on information obtained at compile time, 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 –deadlocks are detected and eliminated without effecting the expected program semantics.

8.1.2. Dynamic Analysis. Dynamic analysis tools [Li et al. 2005; Agarwal and Stoller 2006; Bensalem and Havelund 2005; Harrow 2000; Havelund 2000; Joshi et al. 2009; Jula et al. 2008; Qin et al. 2005; Wang et al. 2008] detect deadlocks at runtime. We discuss each in detail.

Li et. al., [Li et al. 2005] 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 detect deadlocks also 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 hence, leading 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 to detect deadlocks. Additionally, Sammati is capable of recovering from deadlocks unlike Pulse.

Harrow [Harrow 2000] 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., [Bensalem and Havelund 2005; Bensalem et al. 2006] proposed a runtime verification algorithm to detect potential deadlocks in applications written in Java. In their approach they instrument the java bytecode to collect 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 [Agarwal and Stoller 2006] 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., [Joshi et al. 2009] 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 Goodlock [Havelund 2000] algorithm to detect potential deadlocks in multi-threaded applications.

Jula et. al., [Jula et al. 2008] proposed Dimmunix to enable applications develop immunity to deadlocks. Dimmunix captures the signatures of deadlocks as they occur in program 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 avoid deadlocks by maintaining the state information of each deadlock pattern that occurs at runtime and aims to prevent such future occurrences through deadlock prediction.

Qin et. al., [Qin et al. 2005] proposed Rx, a technique to recover programs for 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 and 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., [Wang et al. 2008; Wang et al. 2009] 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.

Most recently, Gerakios et. al., [Gerakios et al. 2011] proposed a deadlock avoidance technique. In their approach, they employ program analysis to collect the order of lock acquisitions and releases in a program. 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., [Berger et al. 2009] 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.

Joshi et. al., [Joshi et al. 2010] proposed CheckMate to detect a broad range of deadlocks resulting from locks, condition variables, and other forms of synchronization primitives. CheckMate collects the 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 consequently it is prone is false positives and false negatives.

Several techniques [Musuvathi and Qadeer 2007; Musuvathi et al. 2008; Sen 2008; Burckhardt et al. 2010; Edelstein et al. 2008] 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.

8.2. Transactional Memory (TM)

Transactional memory [Herlihy and Moss 1993; Shavit and Touitou 1995] 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 implementation. However, interactions between transactions and non-transactional code is still ill-defined. Blundell et.al., [Colin Blundell and Milo 2006] 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 races, program errors or even deadlock. Defining the semantics of the memory model in the interaction between transactional and non transactional code is an on going area of research [Hudson et al. 2006; Shpeisman et al. 2007; Spear et al. 2007].

Most TM systems are based on language support with special programming language constructs [Harris and Fraser 2003; Ni 2008; Yoo et al. 2008] or API [Saha et al. 2006] to provide TM semantics. Alternatively, some TMs rely on special memory allocation primitives [Hudson et al. 2006] and wrappers [Ni 2008] to support transactional memory semantics. Fine-grain privatization of updates within transactions is typically achieved by instrumenting load and store operations, which can result in significant impact on application speedup [Cascaval et al. 2008]. Sammati may be viewed as a pessimistic STM without optimistic concurrency.

9. ONGOING AND FUTURE WORK

We are currently working on improving the performance of Sammati through compile time analysis and instrumentation. Sammati's overhead primarily stems from the protection and privatization of the virtual address space. We believe that we can reduce this runtime overhead by employing program analysis to accurately determine the write-set (i.e., data modified) within a lock even in the presence of nested and conditional lock acquisition and release sequences. There are several challenges in the details of this work. First we need a mechanism to identify locks and their scope in the program. Second, we need to accurately determine the write-set (i.e., data modified) within a lock. In situations where program analysis cannot determine control flow, the Sammati runtime can act as the fail-safe to provide deterministic deadlock detection and recovery. Third, we need to isolate the memory updates within locks to facilitate recovery on deadlock. We need a lightweight memory shadowing mechanism to accomplish isolation. Additionally, the ordering and integrity of the load and store instructions must be preserved in order to maintain program correctness. We plan on leveraging the LLVM compiler infrastructure to implement some of our proposed techniques.

During deadlock recovery, an arbitrary lock that is part of the cyclic dependency is chosen as the victim and the associated critical section is rolled back to a point prior to lock acquisition. Instead of selecting an arbitrary lock as the victim, we present three different metrics that can be used to bias lock recovery. First, to support non-idempotent operations, we bias the victim selection against critical sections that are non-restartable. Second, to support thread priority, we propose to bias the victim selection such that locks owned by lower priority threads are selected as victims. Finally, we propose to track execution progress (in CPU cycles) and bias the victim selection such that the probability of selection of a lock as a victim is inversely proportional to the CPU time already consumed by the associated critical section.

10. CONCLUSION

In this paper 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 and architecture of Sammati and we provided several extensions to address its shortcomings and to reduce its performance overhead.

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 and its performance is comparable to native Pthreads for some applications with modest memory overhead.

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 work will impact the future of concurrent programming and assist in improving the productivity of application developers.

REFERENCES

AGARWAL, R. AND STOLLER, S. D. 2006. 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*. ACM, 51–60.

AMZA, C., COX, A., DWARKADAS, S., KELEHER, P., LU, H., RAJAMONY, R., YU, W., AND ZWAENEPOEL, W. 1996. Treadmarks: shared memory computing on networks of workstations. *Computer 29,* 2, 18 –28.

BENSALEM, S., FERNANDEZ, J.-C., HAVELUND, K., AND MOUNIER, L. 2006. Confirmation of deadlock potentials detected by runtime analysis. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*. PADTAD '06. ACM, New York, NY, USA, 41–50.

BENSALEM, S. AND HAVELUND, K. 2005. Scalable deadlock analysis of multi-threaded programs. In *PADTAD '05: Proceedings of the Parallel and Distributed Systems: Testing and Debugging*. Vol. 1. Springer-Verlag.

BERGER, E. D., YANG, T., LIU, T., AND NOVARK, G. 2009. 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*. ACM, 81–96.

BOYAPATI, C., LEE, R., AND RINARD, M. 2002. 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*. ACM, 211–230.

BURCKHARDT, S., KOTHARI, P., MUSUVATHI, M., AND NAGARAKATTE, S. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. *SIGARCH Computer Architecture News 38,* 1, 167–178.

CARTER, J. B., BENNETT, J. K., AND ZWAENEPOEL, W. 1991. Implementation and performance of munin. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*. SOSP '91. ACM, New York, NY, USA, 152–164.

CASCAVAL, C., BLUNDELL, C., MICHAEL, M., CAIN, H. W., WU, P., CHIRAS, S., AND CHATTERJEE, S. 2008. Software Transactional Memory: Why Is It Only a Research Toy? *ACM Queue 6,* 5, 46–58.

COLIN BLUNDELL, E. C. L. AND MARTIN, M. M. K. 2005. Deconstructing transactional semantics: The subtleties of atomicity. *WDDD '05: Proceedings of the 4th workshop on duplicating, deconstructing and debunking*, 48–55.

COLIN BLUNDELL, E. C. L. AND MILO, M. 2006. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters 5,* 2, 17.

DOUG LEA. 2012. A memory allocator. `http://g.oswego.edu/dl/html/malloc.html`.

EDELSTEIN, O., FARCHI, E., GOLDIN, E., NIR, Y., RATSABY, G., AND UR, S. 2008. Framework for testing multi-threaded java programs. In *Concurrency and Computation: Practice and Experience*. Vol. 15. USENIX, 485–499.

ENGLER, D. AND ASHCRAFT, K. 2003. RacerX: effective, static detection of race conditions and deadlocks. *SIGOPS Operating Systems Review 37,* 5, 237–252.

FLANAGAN, C., LEINO, K. R. M., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. 2002. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. ACM, 234–245.

GERAKIOS, P., PAPASPYROU, N., AND SAGONAS, K. 2011. 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. ACM, New York, NY, USA, 15–28.

GERAKIOS, P., PAPASPYROU, N., SAGONAS, K., AND VEKRIS, P. 2011. Dynamic deadlock avoidance in systems code using statically inferred effects. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*. PLOS '11. ACM, New York, NY, USA, 5:1–5:5.

HARRIS, T. AND FRASER, K. 2003. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*. Vol. 38. ACM, 388–402.

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

HAVELUND, K. 2000. 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*. Springer-Verlag, 245–264.

HERLIHY, M. AND MOSS, J. E. B. 1993. Transactional memory: architectural support for lock-free data structures. *SIGARCH Computer Architecture News 21,* 2, 289–300.

HUDSON, R. L., SAHA, B., ADL-TABATABAI, A.-R., AND HERTZBERG, B. C. 2006. McRT-Malloc: a scalable transactional memory allocator. In *ISMM '06: Proceedings of the 5th International Symposium on Memory Management*. ACM, 74–83.

JIN, G., SONG, L., ZHANG, W., LU, S., AND LIBLIT, B. 2011. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. PLDI '11. ACM, New York, NY, USA, 389–400.

JOSHI, P., NAIK, M., SEN, K., AND GAY, D. 2010. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. FSE '10. ACM, New York, NY, USA, 327–336.

JOSHI, P., PARK, C.-S., SEN, K., AND NAIK, M. 2009. 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*. ACM, 110–120.

JULA, H., TRALAMAZZA, D., ZAMFIR, C., AND CANDEA, G. 2008. Deadlock immunity: Enabling systems to defend against deadlocks. In *In Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX.

KELEHER, P., COX, A. L., DWARKADAS, S., AND ZWAENEPOEL, W. 1994. 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*. USENIX Association, Berkeley, CA, USA, 10–10.

LATTNER, C. AND ADVE, V. 2004. 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. 75–.

LI, T., ELLIS, C. S., LEBECK, A. R., AND SORIN, D. J. 2005. Pulse: a dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the annual conference on USENIX Annual Technical Conference*. ATEC '05. USENIX Association, Berkeley, CA, USA, 3–3.

LU, S., PARK, S., SEO, E., AND ZHOU, Y. 2008. 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*. ACM, 329–339.

MENON, V., BALENSIEFER, S., SHPEISMAN, T., ADL-TABATABAI, A.-R., HUDSON, R. L., SAHA, B., AND WELC, A. 2008. Single global lock semantics in a weakly atomic stm. *SIGPLAN Notices. 43*, 15–26.

MUSUVATHI, M. AND QADEER, S. 2007. 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. ACM, New York, NY, USA, 446–455.

MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. 2008. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. OSDI'08. USENIX Association, Berkeley, CA, USA, 267–280.

NAIK, M., PARK, C.-S., SEN, K., AND GAY, D. 2009. Effective static deadlock detection. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*. IEEE Computer Society, 386–396.

NI, Y. E. A. 2008. 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*. ACM, 195–212.

PRAUN, C. V. 2004. Detecting synchronization defects in multi-threaded object-oriented programs. In *PhD Thesis*.

PYLA, H. K. AND VARADARAJAN, S. 2010. Avoiding deadlock avoidance. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. PACT '10. ACM, New York, NY, USA, 75–86.

QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. 2005. 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*. ACM, 235–248.

RANGER, C., RAGHURAMAN, R., PENMETSA, A., BRADSKI, G., AND KOZYRAKIS, C. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE Computer Society, 13–24.

SAHA, B., ADL-TABATABAI, A.-R., HUDSON, R. L., MINH, C. C., AND HERTZBERG, B. 2006. 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*. ACM, 187–197.

SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. 1997. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems 15,* 4, 391–411.

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

SHANBHAG, V. K. 2008. Deadlock-detection in java-library using static-analysis. *Asia-Pacific Software Engineering Conference 0*, 361–368.

SHAVIT, N. AND TOUITOU, D. 1995. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of Distributed Computing*. ACM, 204–213.

SHPEISMAN, T., MENON, V., ADL-TABATABAI, A.-R., BALENSIEFER, S., GROSSMAN, D., HUDSON, R. L., MOORE, K. F., AND SAHA, B. 2007. Enforcing isolation and ordering in STM. In *PLDI '07: Proceedings of ACM SIGPLAN 2007 conference on Programming Language Design and Implementation*. Vol. 42. ACM, 78–88.

SPEAR, M. F., MARATHE, V. J., DALESSANDRO, L., AND SCOTT, M. L. 2007. Privatization techniques for software transactional memory. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of Distributed Computing*. ACM, 338–339.

SPLASH-2. 2012. SPLASH-2 benchmark suite. `http://www.capsl.udel.edu/splash`.

SUN MICROSYSTEMS. 2012. Lock_Lint - Static Data Race and Deadlock Detection Tool for C. `http://developers.sun.com/solaris/articles/locklint.html`.

VALGRIND. 2012. Helgrind: a thread error detector. `http://valgrind.org/docs/manual/hg-manual.html`.

VOLOS, H., TACK, A. J., SWIFT, M. M., AND LU, S. 2012. Applying transactional memory to concurrency bugs. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '12. ACM, New York, NY, USA, 211–222.

WANG, C. AND WU, Y. 2010. From lock to correct and efficient software transactional memory. In *Proceedings of the 2010 Workshop on Interaction between Compilers and Computer Architecture*. INTERACT-14. ACM, New York, NY, USA, 8:1–8:8.

WANG, Y., KELLY, T., KUDLUR, M., LAFORTUNE, S., AND MAHLKE, S. 2008. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *In Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX.

WANG, Y., LAFORTUNE, S., KELLY, T., KUDLUR, M., AND MAHLKE, S. 2009. 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. ACM, New York, NY, USA, 252–263.

WILLIAMS, A., THIES, W., AND ERNST, M. D. 2005. Static deadlock detection for java libraries. In *ECOOP 2005 - Object-Oriented Programming*. 602–629.

XIONG, W., PARK, S., ZHANG, J., ZHOU, Y., AND MA, Z. 2010. Ad hoc synchronization considered harmful. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. OSDI'10. 1–8.

YOO, R. M., NI, Y., WELC, A., SAHA, B., ADL-TABATABAI, A.-R., AND LEE, H.-H. S. 2008. 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*. ACM, 265–274.

ZHANG, W., SUN, C., AND LU, S. 2010. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS XV:Proceedings of the 15th International conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA, 179–192.