Avoiding Deadlock Avoidance

Hari K. Pyla and Srinidhi Varadarajan
Center for High-End Computing Systems
Department of Computer Science
Virginia Tech, Blacksburg, United States
{harip, srinidhi}@cs.vt.edu

ABSTRACT

The evolution of processor architectures from single core designs with increasing clock frequencies to multi-core designs with relatively stable clock frequencies has fundamentally altered application design. Since application programmers can no longer rely on clock frequency increases to boost performance, over the last several years, there has been significant emphasis on application level threading to achieve performance gains. 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 called Sammati that provides automatic deadlock detection and recovery for threaded applications that use the POSIX threads (pthreads) interface - the de facto standard for UNIX systems. The runtime is implemented as a pre-loadable library and does not require either the application source code or recompiling/relinking phases, enabling its use for existing applications with arbitrary multi-threading models. Performance evaluation of the runtime with unmodified SPLASH, Phoenix and synthetic benchmark suites shows that it is scalable, with speedup comparable to baseline execution with modest memory overhead.

Categories and Subject Descriptors

D.1 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms

Algorithms, Design, Languages, Measurement, Performance and Reliability

PACT'10, September 11–15, 2010, Vienna, Austria.

Keywords

Deadlock Detection and Recovery, Concurrent Programming and Runtime Systems

1. INTRODUCTION

Over the last several years, the dominant trend in processor architectures has been the move away from increasing clock frequencies of single core processors to adding more processing cores. Unlike clock frequency increases, which transparently increase application performance, the trend to multi-core architectures squarely places the burden on the application to use concurrent programming to achieve performance. Correct concurrent programming is notoriously hard as has been well documented in [19]. The most common concurrency bugs are data races that arise due to unguarded or improperly guarded memory updates and deadlocks that arise due to circular dependencies among locks. While data races can be ameliorated by appropriate synchronization (a non trivial problem in itself), deadlocks require fairly complex deadlock avoidance techniques, which may fail when the order of lock acquisitions is not known a priori. Furthermore, due to the potential for deadlocks, application programmers cannot arbitrarily compose lock based codes without knowing the internal locking structure.

In this paper, we present Sammati (agreement in Sanskrit), a software system that transparently detects and eliminates deadlocks in threaded codes, without requiring any modifications 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. Sammati is implemented 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. Sammati supports arbitrary application level threading models including those that use locks for concurrency control where serial lock elision does not result in a program with the same semantics. Our core goal in this work is to achieve composability of lock based codes. 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 limitations) of existing lock based codes and does not provide any mechanisms to optimistically execute critical sections concurrently as in transactional memory.

Fundamentally, Sammati works by associating memory accesses with locks and privatizing memory updates within

a critical section. Memory updates within a critical section are made visible outside the critical section on the release of the parent lock(s) viz. the containment property. On the acquisition of every lock, Sammati runs a single cycle deadlock detection algorithm. If a deadlock is detected, our deadlock elimination algorithm breaks the cycle by selecting a victim, rolling it back to the acquisition of the offending lock, and discarding any memory updates. Since our 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.

While the core idea behind Sammati is quite simple, there are several challenges in the details. First, we need a transparent mechanism for detecting memory updates within a critical section and privatizing the updates. Second, in the context of nested locks we need to define a set of visibility rules that preserve existing lock semantics, while still permitting containment based deadlock elimination and recovery. Finally, we need a deadlock detection and recovery mechanism that is capable of deterministically eliminating deadlocks without either (a) deadlocking itself or (b) requiring an outside agent.

The rest of this paper is organized as follows. In section 2 we present related work in the area of deadlock detection and recovery. Section 3 describes the components of Sammati including privatization, visibility rules and deadlock detection and recovery algorithms. Section 4 presents our implementation of the Sammati runtime in the Linux operating systems. In section 5, we present the results from a performance evaluation of Sammati on the SPLASH [31] and Phoenix [24] shared memory benchmark suites, which include applications that use a variety of threading models. Finally, we conclude the paper in section 6 with directions for future work.

2. RELATED WORK

2.1 Transactional Memory (TM)

Transactional memory [15, 28] 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., [7] introduce 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 [16, 29, 30].

Most TM systems are based on language support with special programming language constructs [12, 22, 34] or API [25] to provide TM semantics. Alternatively, some TMs rely on special memory allocation primitives [16] and wrappers [22] 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 [6].

2.2 Deadlock Detection and Recovery

Several techniques were proposed to detect deadlocks in concurrent programs. In this section we briefly discuss some of the literature. Static type checking systems [4, 10, 11, 20, 21, 26, 27, 33] use program analysis to determine deadlocks. While static 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, they may generate false positives in identifying deadlocks resulting in spurious recovery actions. On the other hand, dynamic analysis tools [1, 2, 13, 14, 17, 18, 23, 32] detect deadlocks at runtime. For instance, certain runtime tools instrument programs and introduce scheduler [5, 9, 17] noise to increase the chances of deadlock occurrence while testing concurrent programs. Other techniques to detect deadlocks include sequential composition of threads with speculative execution [3], which can perform deadlock recovery for applications written with fork-join parallelism, evaluating runtime traces [1, 2, 14], employing binary instrumentation [13], generating control-flow-graphs [32] of programs. Additionally, certain tools such as Rx [23] and Dimmunix [18] provide recovery support on detecting a deadlock. For instance Rx uses checkpointing to rollback to a recent checkpoint and resume execution in the new environment. Dimmunix maintains a state information of each deadlock pattern that occurs at runtime and aims to prevent such future occurrences through deadlock prediction.

In contrast, Sammati is implemented as a runtime system that uses a deterministic algorithm to detect and eliminate deadlocks with no false positives or negatives. Second, our approach does not rely on source analysis or require any modifications to the source code. Third, it is transparent to the source application - deadlocks are detected and eliminated with no change to the expected program semantics. Fourth, unlike transactional memory systems, Sammati's runtime privatizes at page granularity and does not instrument individual load store operations and thereby largely preserves application speedup. Finally, Sammati performs efficient deadlock recovery without requiring a complete application checkpoint and the associated overhead.

3. DESIGN

The core goal of Sammati is to provide transparent deadlock detection and recovery for threaded codes with arbitrary threading models. In this section, we present the challenges behind and the design of the major components of Sammati.

3.1 Privatization and Containment

In order 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 or 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.

To implement containment for threaded codes, we use a technique first described in [3]. 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 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 Visibility Rules

Visibility rules define when memory updates made within a critical section are visible outside the critical section. This is relatively straightforward for a single lock (shown in Figure 1(a)) around a critical section –updates are visible at the release of the lock. However, nested locks are more complex. Consider the nested lock sequence shown in (Figure 1(b)). If the memory update to x 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 x visible. A secondary issue exists here in associating data

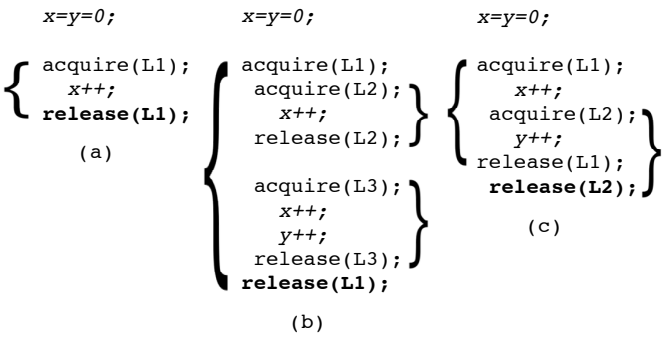

Figure 1: Visibility rules for memory updates within a lock context.

with locks. When a unit is 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 1(c), it is not possible to transparently determine what data should be made visible. The variable x is protected by lock L1, however, the variable y, may be protected by L2 or L1.

To ensure containment of memory updates in the presence of nested locks, Sammati makes memory updates visible on the release of *all* locks. This is a necessary and sufficient condition for Sammati's deadlock elimination and recovery. Furthermore, in order to preserve the semantics of mutual exclusion, we 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. To see why, consider the example shown in Figure 1(b). If lock L2 is released at its original location, but its update to x is privatized (since there is another lock L1 around the same critical section), another thread may acquire lock L2, update x and release L2, creating a 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.

3.3 Deadlock Detection and Recovery

A concurrent multi-threaded program can hold a set of locks (holding set) and simultaneously be waiting on one or more locks (waiting set). In this 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. Multi-cycle deadlock detection is typically implemented as an external process that implements distributed deadlock detection and recovery.

Since threads have been transparently converted to cords in Sammati, the key observation here is that each cord (a single threaded process) may only wait on one lock. This significantly simplifies deadlock detection, since from the perspective of each cord, all deadlocks are single cycle deadlocks. Furthermore, since all cords share a global address space, the deadlock detection algorithm has access to the holding and waiting sets of each cord. Deadlock detection can be hence performed at lock acquisition with the guar-

```
holding hash table ( lock : key, pid : value)
waiting hash table ( pid : key, lock : value)
lock_list : list of locks ordered by program
order acquisition of locks.

void deadlock_free_lock (lock L)
{
 lock S; /* globally shared lock across cords */
 lock W; /* local lock identifier */

 start:
  R := restore point containing the contents
  of stack frame and processor registers.
  set restore point R for lock (L) on rollback.

  if ( returning from a restore point flag) {
   restore the stack.
   free the old stack context and reset
   returning from restore point flag.
  }
  id := my pid;
  acquire lock (S).
  try acquiring lock (L).
  if( lock (L) is acquired successfully ) {
   insert in holding hash table (lock (L), id).
   insert (lock (L), restore point (R)) at tail
   of lock_list.
   release lock (S).
  }
  else {
   insert id in waiting hash table(id, lock (L))
   W := L;

   traverse:
    candidate:= find lock (W) in holding
    hash table.
    if( candidate == id ) {
     /* We have a deadlock !!! */
     recover_from_deadlock (W).
     return to restore point (W).
    }
    else {
     W := lock that candidate is waiting on;
     if( lock (W) is valid ) {
     /*continue traversing the waits for graph*/
      goto traverse;
     }
     else {
      /* There is no deadlock. */
      release lock (S).
      acquire lock (L).
      acquire lock (S).
      delete ids entries from waiting hash table
      if (lock(L) is acquired successfully) {
       insert lock (L) in holding
       hash table (lock (L), id).
       insert (lock (L), restore point (R)) at
       tail of lock_list.
       release lock (S).
      }
      else{
       /* Error in acquiring the Lock (L) */
       release lock (S).
       throw error and terminate program.
      }
     }
    }
  }
   update internal data structures and return.
}
```

Figure 2: Deadlock detection algorithm

```
void recover_from_deadlock (lock (W))
{
  For all entries starting from
  head of lock_list find the
  first occurrence of lock (W)
  and do the following

  discard all the modifications made
  by locks from lock (W) to tail of
  lock_list.

  release all locks including lock (W).
  update internal data structures.
  clear relevant entries from holding hash
  table including entries for lock (W).

  release lock (S).
  return;
}
```

Figure 3: Deadlock recovery algorithm

antee that the deadlock detection algorithm itself cannot be deadlocked.

Figure 2 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 cord can wait on at most one lock, the depth of traversal of this deadlock detection algorithm is equal to the diameter of the waits-for graph. Since we use hash tables to represent the graph, our deadlock detection algorithm has a time complexity upper bound of O(n), where n is the number of cords.

Note that 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 this algorithm and hence the initial non blocking trylock may fail, and yet the holding hash table may not have an entry for the owning cord. This

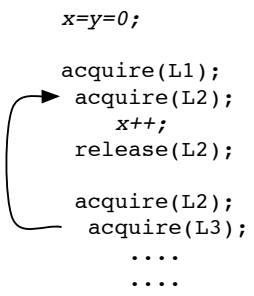

Figure 4: In this example, if the acquisition of L3 results in a deadlock and the victim was L2, Sammati's deadlock recovery rolls back to the earliest acquisition of L2.

condition (an intentional benign race) cannot result in a deadlock. We do not present our formal proof of correctness due to space constraints. The intuition behind the proof 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.

The deadlock detection algorithm presented above detects a deadlock and identifies a lock W as the victim for deadlock recovery. The deadlock recovery algorithm scans the list of locks to find the oldest acquisition of W in program order and uses the associated recovery point 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 back to deadlock free lock acquisition.

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 4. A cord C acquires a lock L1, followed by lock L2 and updates a variable x. 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 x 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 x, creating an illegal write-write conflict with the privatized copy within cord C.

4. IMPLEMENTATION

Sammati is implemented as a shared library and that is pre-loaded by the dynamic linker (ld) before executing the binary. It implements most of the POSIX threads interface, including mutual exclusion locks and barriers. The current implementation does not support synchronization control through condition variables. Condition variables can be supported by a fairly straightforward extension to Sammati's design and is the subject of a future paper. The implementation of Sammati's runtime is available as a public download.

4.1 Shared Address Space

A threaded process has a shared address space, with a distinct stack for each thread. To create the illusion of a shared address space among processes, the constructor in Sammati 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 they inherit the shared global data mapping. Any modifications made by any cord to global data is immediately visible to all cords. As an aside, we note that 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 automatically. 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 before executing its thread start function.

The heap of a process is also shared among all threads of a process. To implement this abstraction, we modified Doug Lea's [8] 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. 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 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.*

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

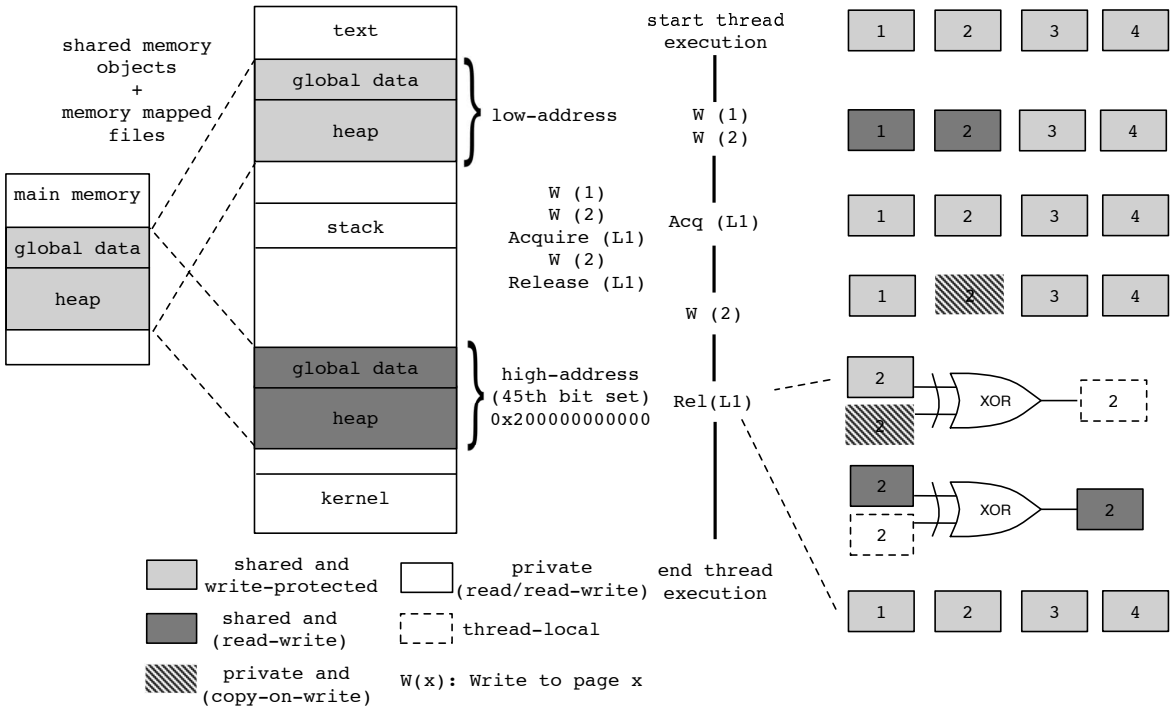

Figure 5: (left) Illustrates the virtual memory address (VMA) layout of each process (cord). Sammati provides memory isolation by transforming threads to processes and uses shared memory objects and memory mapped files to share global variables and the process heap among cords. (right) Illustrates how the VMA is manipulated by Sammati with a simple example explained in text.

4.2 Detecting Memory Updates Within Locks

In order to perform rollback on deadlock recovery successfully, the runtime system must be able to precisely identify write accesses within locks. Hence, 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 (marked in bold in Figure 1).

In order to track accesses, Sammati initially write-protects (PROT_READ) all pages of the shared VMA. Read accesses to shared data do not produce any faults and execute as they would otherwise. However, if a process attempts to write to a shared page (global data, heap), the runtime system handles the access fault (SEGV_ACCERR) and makes a note of the page and the current context of execution. Sammati maintains a unique list of pages that were modified within the scope of each lock. The permissions of the page are then set to read-write so that the cord can continue its execution. On acquiring a lock, only the set of pages that were modified before acquiring a lock are write-protected instead of protecting the entire shared VMA. 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, thereby minimizing the cost of changing memory protection.

4.3 Privatization

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) since all pages in the shared VMA are write protected. Sammati handles the access violation by remapping the faulting page from the shared memory backing in MAP_PRIVATE mode. In this mode, updates from the cord in the lock context are no longer visible to other cords, effectively privatizing the page. Sammati 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. 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. Note that the space overhead of this solution is O(W), where W is the write set (in pages) within a lock context.

Intuitively, Sammati implements a lazy privatization scheme that defers privatization to the instant when the first memory update occurs. Note that lazy privatization results in weak atomicity. While a conservative privatization of the entire address space at the acquisition of the first lock in a nested lock sequence would give us 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 the more efficient lazy privatization approach and its resultant weak atomicity.

4.4 Mutual Exclusion

Sammati provides mutual exclusion among processes by transforming all mutex exclusion locks (pthread_mutex_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 (refer to section 3).

A subtle side effect of memory isolation occurs because locks in a 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). 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.

In order to address this side effects of 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 use this feature to our advantage. Recall that our runtime system maps globals and heap (described in 4.1) using shared memory objects and memory mapped files. Using the same shared memory objects and memory mapped file, Sammati creates an identical secondary mapping of the global data sections and heap at a *high address* (45th bit set) in the VMA of each cord. The application is unaware of this mapping, and performs its accesses (reads/writes) at the original low address space. In effect, the high address mapping creates a *shadow* address space for all shared program data and modifications (unless privatized) are visible in both address spaces (shown in Figure 5). The high address space shadow is always shared among threads and is never privatized.

To perform mutual exclusion operations, we transform the address of a mutual exclusion lock by setting the high address bit and performing 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.

4.5 Inclusion

When a cord exits a lock context, updates contained in its privatized data must be made visible and reconciled with other cords. In order to perform this *inclusion*, we need to identify the exact write set of the lock context. To do this, for every page modified within a lock context, we compute an XOR difference (byte wise XOR) between the 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. To perform inclusion, we apply the XOR difference to the high address shadow region of the VMA (shown in Figure 5) 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. 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.

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 5 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.

4.6 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 during the inclusion phase 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 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. Since the race is detected at inclusion, Sammati can identify the race, but doesn't have enough information to identify the other cords involved in the conflict.

5. PERFORMANCE EVALUATION

5.1 Experimental setup

To analyze the impact of Sammati on threaded applications, we evaluated its performance over SPLASH [31], Phoenix [24] and synthetic benchmark suites. The SPLASH suite contains applications from several domains including high-performance computing, signal processing and graphics. The Phoenix suite contains applications from enterprise computing, artificial intelligence, image processing and scientific computing domains. The synthetic benchmarks contain programs written intentionally to create deadlocks and deadlock examples from [3, 17]. We were able to run all the benchmarks unmodified, except *radix* from SPLASH since it uses condition variables for communication among threads, which as mentioned earlier is not supported in the current implementation of Sammati. We ran each benchmark under two scenarios, one involving Sammati, where we pre-load

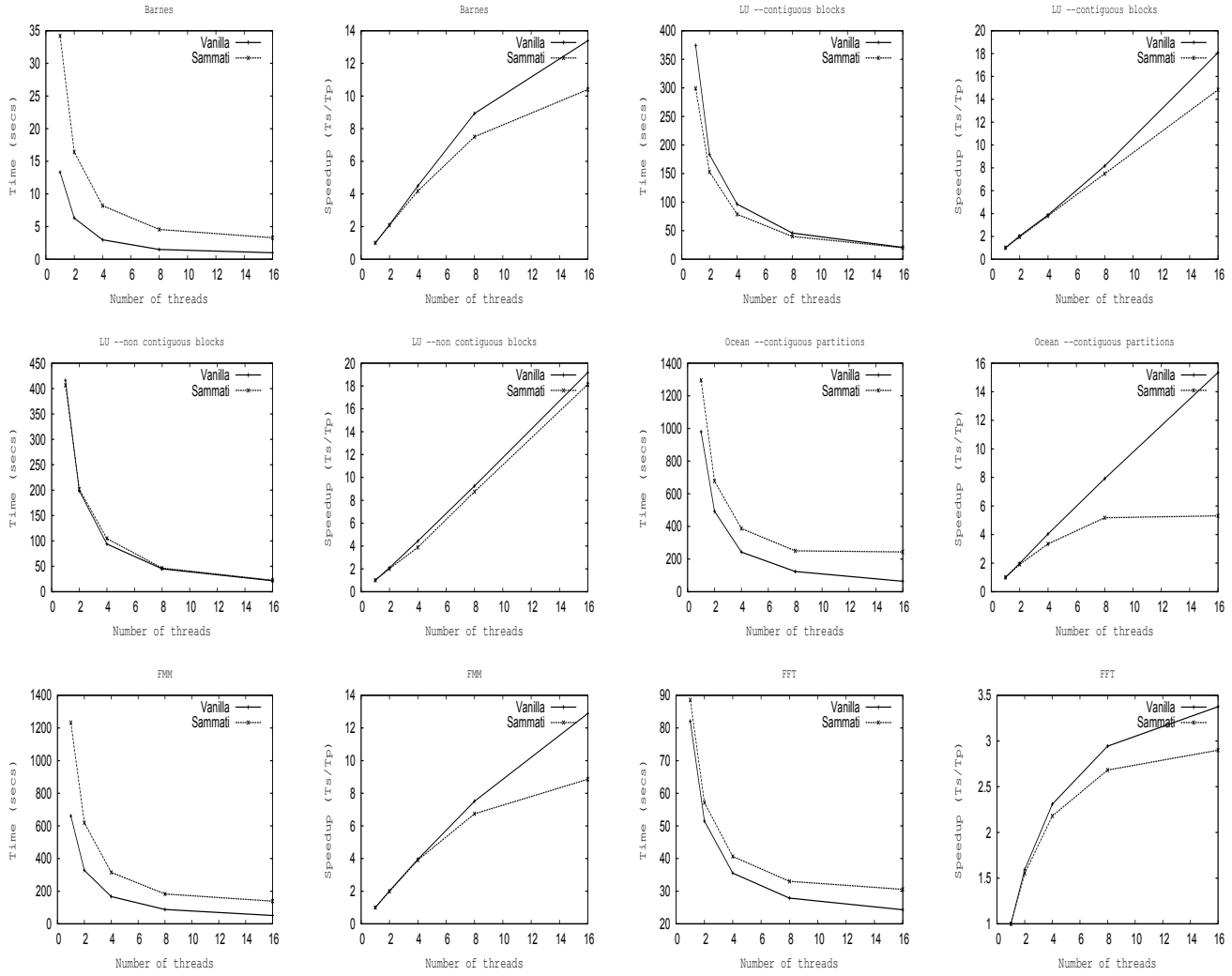

Figure 6: Performance of applications from the SPLASH benchmark suite with Sammati on a 16 core system. The results show that Sammati has relatively small impact on speedup (except when the runtime is small), but impacts runtime, particularly for applications with fine-grain locking. In contrast, transactional memory implementations may have significant impact on speedup.

our runtime system using LD_PRELOAD, and the other by running the vanilla pthreads application. The performance results presented are the average of five independent runs of each scenario. Runtime for each benchmark run was obtained through wallclock time, using the UNIX *time* command.

All performance measurements were obtained from a 16 core shared memory machine (NUMA) running Linux 2.6.24-21 with 64GB of RAM. The test system contains four 2 GHz Quad-Core AMD Opteron processors.

5.2 Performance

Figure 6 illustrates the performance of Sammati and pthreads with SPLASH benchmarks. In order to avoid scheduling noise and the runtime's fixed startup costs, we report results of the SPLASH benchmarks that have a runtime of at least a few seconds. The performance of Sammati is comparable to pthreads for most applications with the exception of *Ocean*.

The primary reason for the fall-off in speedup for Ocean stems from our mechanism to detect updates within a critical section. Recall that all shared data is maintained in read-only form. When updates to shared data occur outside a lock context, we store the page address in a write-set list and change the page permissions to read/write. At the acquisition of the next lock, we only change the page permissions of pages in the write-set list to read only, thereby avoiding the cost of write protecting the entire data space. This implementation is biased towards fine-grain locking where the write-set between lock contexts is small, which is not true for OCEAN. When the write-set between lock contexts is large, the cost of handling the access error and changing the page permissions outweighs the benefits from this approach. Note that in this scenario, the rate of lock acquisitions is small. In the future we expect to automatically deduce the write set between lock contexts and tune the policy based on application behavior. Furthermore, we are

Table 1: Illustrates the number of locks acquired and lock acquisition rate for a few benchmarks from SPLASH and Phoenix suite.

Benchmark	Number of Threads	Number of Locks	Lock acquisition rate (locks/sec)
Ocean (contiguous)	1	173	0.1765
	2	346	0.702
	4	692	2.85
	8	1384	11.178
	16	2768	43.328
PCA	1	10001	4.73
	2	10002	9.40
	4	10004	18.51
	8	10008	26.161
	16	10016	28.46
Barnes	1	275216	20630.88
	2	275222	43616.79
	4	139216	46842.53
	8	112228	75119.14
	16	537538	53968.87
FMM	1	4517437	6838.26
	2	4521242	13782.92
	4	4543919	27171.02
	8	4559515	51836.23
	16	4586903	89490.069

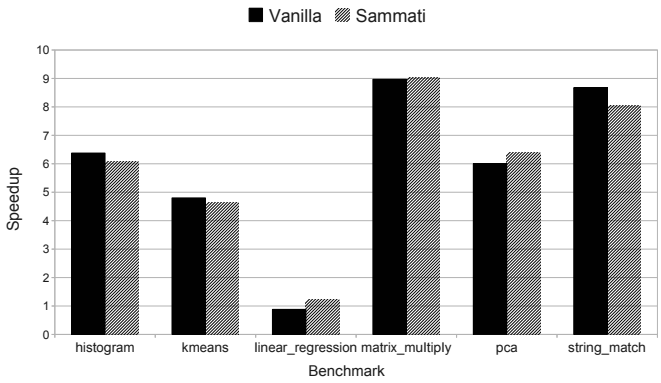

Figure 7: Illustrates the performance of Sammati and pthreads for Phoenix benchmarks on a 16 core machine. The performance of Sammati is identical to pthreads for most applications.

```
  Thread 1      Thread 2

acquire(L1);   sleep(1);
 sleep(2);   acquire(L2);
acquire(L2); acquire(L1);

          (a)


 Thread 1      Thread 2       Thread 3

acquire(L1); acquire(L2);   acquire(L3);
 sleep(1);     sleep(2);      sleep(3);
acquire(L2); acquire(L3);   acquire(L1);

          (b)
```

Figure 8: (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.

working on reducing this overhead by moving parts of the runtime - the cords implementation, shared memory infrastructure, memory protection, privatization and inclusion - into the Linux operating system. We expect this to reduce the cost of privatization by reducing the number of protection faults. Currently the OS faults on our write-protection scheme and then has to schedule a signal to be delivered to the Sammati runtime.

In Figure 7 we present the results of Phoenix benchmark suite. Sammati achieves almost identical performance to that of native pthreads with the exception of *linear_regression* and *pca* where Sammati achieves slightly better speedup than pthreads. This is because by default when threads are created they are "affinitized" to the same core and it takes a while for the operating system load balancer to move the threads from one core to another in an attempt to balance the CPU workload. In contrast, processes in Linux are started on distinct cores when they are spawned and benefit from the larger cache almost immediately.

We measured the number of locks acquired and lock-acquisition-rate (total locks acquired/total runtime) for all applications used in this study. In Table 1 we present the results that show widely varying rates across the spectrum of applications from the SPLASH and Phoenix suites. While Sammati's runtime is impacted by lock acquisition rate, it shows speedup comparable to the native pthreads case. This is in contrast to transactional memory systems, which have significant impact on speedup, largely due to privatization at the instruction level and the need to guard every read from read/write conflicts. [6].

5.3 Memory Overhead

Sammati's memory overhead consists of two components - Sammati metadata and the overhead of privatization. Sammati's metadata is relatively small at around 1 - 2 MB independent of the number of cords. The privatization memory overhead is incurred when the application is in a critical section. This overhead is caused by the runtime maintaining twin copies of the page, which are then used to compute the XOR differences during inclusion. Note that the twin

pages are freed at the end of the critical section, i.e. this memory overhead is transient. The memory footprint of the 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 we operate at page granularity. To quantify this overhead, we measured the maximum number of twin pages within any critical section, which yields the upper bound on the transient memory overhead of Sammati. We found that a majority of the benchmarks modified only a few pages within any given critical section and sum of the maximum number of twin pages for all cords was trivial (approx 16 pages). The highest overhead is for BARNES at 298 pages for 16 cords, which represents a modest 1.192 MB of memory.

5.4 Deadlock Detection and Recovery

In Figure 8 we present a few examples from our synthetic benchmark suite. In the first example, two threads (shown in Figure 8(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 8(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.

6. ONGOING AND FUTURE WORK

We are working in several extensions to Sammati to reduce the runtime overhead. First, we are working on moving the implementation of cords and privatization into the Linux kernel. During lock acquisition, Sammati write protects all pages in the shared VMA. The first write access to a page in a lock context results in an access fault inside the Linux kernel, which then generates a fault to our user level segmentation violation handler. By handling the fault and creating the twin within the kernel, we can halve the cost of fault handling, which is the main component of the performance overhead of fine-grain locks. Second, we are working on an API that can be used by program analysis tools to indicate the write-set within a lock, which can then be privatized at lock acquisition without incurring the fault handling overhead of our current solution. We believe that there are a significant number of cases where program-analysis can determine the write-set, and where it cannot, we can automatically fall back to the current runtime solution. Finally, we are working on a speculative execution model that uses Sammati's privatization to enable several hard to parallelize algorithms to execute concurrently in the same data space. The basic idea here is to extend the acquire/release semantics of a lock to acquire/commit/abort. Multiple algorithms can operate concurrently over the same data – for instance multiple sort algorithms, or multiple numerical quadrature codes – with the results coming from the first algorithm to complete successfully.

7. CONCLUSION

In this paper we presented Sammati, a runtime system for transparent deadlock detection and recovery in POSIX threaded applications. 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 presented the results of a performance evaluation of Sammati using SPLASH, Phoenix and synthetic benchmark suites. Our results indicate that the speedup of Sammati is comparable to native Pthreads for most applications with modest memory overhead. The source code for Sammati will be available at `http://www.csrl.cs.vt.edu/sammati`.

8. REFERENCES

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

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

[3] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications*, pages 81–96. ACM, 2009.

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

[5] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. *SIGARCH Computer Architecture News*, 38(1):167–178, 2010.

[6] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *ACM Queue*, 6(5):46–58, 2008.

[7] E. C. L. Colin Blundell and M. Milo. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2):17, 2006.

[8] Doug Lea. A memory allocator. `http://g.oswego.edu/dl/html/malloc.html`, March 2010.

[9] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. In *Concurrency and Computation: Practice and Experience*, volume 15, pages 485–499. USENIX, 2008.

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

[11] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming*

Language Design and Implementation, pages 234–245. ACM, 2002.

[12] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, volume 38, pages 388–402. ACM, 2003.

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

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

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

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

[17] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 110–120. ACM, 2009.

[18] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *In Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2008.

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

[20] S. Microsystems. Lock_Lint - Static Data Race and Deadlock Detection Tool for C. `http://developers.sun.com/solaris/articles/locklint.html`, Mar 23 2010.

[21] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 386–396. IEEE Computer Society, 2009.

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

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

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

[25] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 187–197. ACM, 2006.

[26] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

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

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

[29] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *PLDI '07: Proceedings of ACM SIGPLAN 2007 conference on Programming Language Design and Implementation*, volume 42, pages 78–88. ACM, 2007.

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

[31] SPLASH-2. SPLASH-2 benchmark suite. `http://www.capsl.udel.edu/splash`, Mar 2010.

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

[33] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for java libraries. In *ECOOP 2005 - Object-Oriented Programming*, pages 602–629, 2005.

[34] R. M. Yoo, Y. Ni, A. Welc, B. Saha, A.-R. Adl-Tabatabai, and H.-H. S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *SPAA '08: Proceedings of the twentieth annual Symposium on Parallelism in Algorithms and Architectures*, pages 265–274. ACM, 2008.