Grace: Safe and Efficient Concurrent Programming

Emery D. Berger Ting Yang Tongping Liu Divya Krishnan Gene Novark
Dept. of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
emery,tingy,tonyliu,divya,gnovark@cs.umass.edu

Abstract

The shift from single to multiple core architectures means that, in order to increase application performance, programmers must write concurrent, multithreaded programs. Unfortunately, multithreaded applications are susceptible to numerous errors, including deadlocks, race conditions, atomicity violations, and order violations. These errors are notoriously difficult for programmers to debug.

We present Grace, a runtime system for multithreaded programs written in C/C++ that provides good scalability and performance while eliminating a range of concurrency errors. With Grace, multithreaded programs *behave* as if all threads were run sequentially. Grace exploits available CPU resources by combining speculative thread execution, supported by a novel virtual memory based transactional memory system, together with a sequential commit protocol that guarantees sequential semantics. We show that Grace ensures the correctness of otherwise-buggy multithreaded programs. Across a suite of CPU-intensive multithreaded applications, Grace often achieves performance and scalability comparable to unsafe thread libraries.

1. Introduction

While the past two decades have seen dramatic increases in processing power, the problems of heat dissipation and energy consumption now limit the ability of hardware manufacturers to speed up chips by increasing their clock rate. This phenomenon has led to a major shift in computer architecture, where complex, deeply-pipelined single-core CPUs have been replaced by multicore CPUs consisting of a number of processing cores.

The implication of this switch is that the performance of sequential applications is no longer increasing with each new generation of processors, because the individual processing components are not getting faster. On the other hand, applications rewritten to use multiple threads can take advantage of these available computing resources to increase their performance by executing their computations in parallel across multiple CPUs.

Unfortunately, writing multithreaded programs is challenging: concurrent multithreaded applications are susceptible to a wide range of errors that are notoriously difficult to debug. For example, multithreaded programs that fail to employ a canonical locking order can *deadlock* [14]. Because the interleavings of threads are non-deterministic, programs that do not properly lock shared data structures can suffer from *race conditions* [23]. A related problem is *atomicity violations*, where programs may lock and unlock individual objects but fail to ensure the atomicity of multiple object updates [19, 10]. Another distinct class of concurrency errors is *order violations*, when a program depends on an execution sequence of threads that the scheduler may not ensure [21].

Contributions: This paper introduces **Grace**, a runtime system that enables safe and efficient concurrent programming for an important class of multithreaded applications, namely those that use *fork-join* parallelism. Grace manages the execution of these multithreaded programs so that they become *behaviorally equivalent* to their sequential counterparts: every thread spawn becomes a sequential function invocation, and locks and thread joins become no-ops. This execution model eliminates the concurrency errors that arise due to multithreading (see Table 1). By treating lock operations as no-ops, Grace eliminates deadlocks caused by cyclic lock acquisition. By committing state changes deterministically, Grace eliminates race conditions. By executing all threads atomically, Grace eliminates atomicity violations. Finally, by always executing threads in program order, it greatly reduces the risk of order violations.

To exploit available computing resources (multiple CPUs or cores), Grace employs a combination of *speculative* thread execution, supported by a novel **virtual-memory based software transactional memory** system, together with a commit protocol that ensures sequential semantics. Grace's VM-based software transactional memory uses page-protection and virtual memory mapping to provide fast transactional support on conventional hardware. Under Grace, threads execute optimistically, writing their updates speculatively. As long as the threads do not conflict, that is, they do not have read-write dependencies on the same memory location, then Grace can safely commit their effects. In case of a conflict, Grace commits the earliest thread in a conflicting set of threads, and re-executes the later threads.

We evaluate Grace on a suite of CPU-intensive, multithreaded benchmarks, which all exhibit fork-join parallelism, as well as a selection of concurrency bugs taken from the literature. We show that Grace achieves comparable scalability and performance to the standard (unsafe) threads library across most of our benchmark suite, while ensuring the correct execution of otherwise-buggy concurrent code.

While Grace cannot prevent those concurrency errors that extend beyond the program itself, such as file system deadlocks, we believe that Grace represents a significant step towards enabling programmers to write multithreaded CPU-intensive applications without compromising correctness.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]. . . $5.00.

Concurrency Error	Cause	Prevention by Grace
Deadlock	cyclic lock acquisition	locks converted to no-ops
Race condition	unguarded updates	all updates committed deterministically
Atomicity violation	unguarded, interleaved updates	threads run atomically
Order violation	threads scheduled in unexpected order	threads execute in program order

Table 1. The concurrency errors that Grace addresses, their causes, and how Grace eliminates them.

```
// Run f(x) and g(y) in parallel.
t1 = spawn f(x);
t2 = spawn g(y);
// Wait for both to complete.
sync;
```

Figure 1. A multithreaded program (using Cilk syntax for clarity).

```
// Run f(x) to completion, then g(y).
spawn f(x);
spawn g(y);
sync;
```

Figure 2. The sequential counterpart of Figure 1, with concurrency operations elided (shown here in gray).

The remainder of this paper is organized as follows. Section 2 presents the sequential semantics that Grace provides. Section 3 presents the VM-based software transactional memory mechanism that Grace uses to enable speculative execution with low overhead. Section 4 describes the commit protocol that enforces sequential semantics, and explains how Grace supports I/O together with optimistic concurrency. Section 5 presents a variety of experimental results, including a detailed experimental evaluation through a series of microbenchmarks, a benchmark suite of concurrent, multithreaded computation kernels, and a suite of programs with concurrency errors. Section 6 surveys related work, Section 7 describes future directions, and Section 8 concludes.

2. Sequential Semantics

To illustrate the effect of running Grace, we use the example shown in Figure 1, which for clarity uses Cilk-style thread operations rather than the actual `pthreads` API that Grace supports. Here, `spawn` creates a thread to execute the argument function, and `sync` waits for all threads spawned in the current scope to complete.

This example program executes the two functions `f` and `g` asynchronously (as threads), and then waits for them to complete. If `f` and `g` share state, this execution could result in atomicity violations or race conditions; if these functions acquire locks in different orders, then they could deadlock. Now consider the version of this program shown in Figure 2, where all calls to spawn and sync (show in gray) are simply ignored.

The second program is the *serial elision* [5] of the first—all parallel function calls have been elided. The result is a serial program that, by definition, cannot suffer from concurrency errors. Because the executions of `f(x)` and `g(y)` are not interleaved and execute deterministically, atomicity violations or race conditions are impossible. Similarly, the ordering of execution of these functions is fixed, so there cannot be order violations. Finally, a sequential program does not need locks, so eliding them prevents deadlock.

2.1 Programming Model

Grace supports the subset of concurrent programs that exhibit what is known as *fork-join parallelism*. Such programs can create threads (which may also create sub-threads), and then wait for them to complete. These programs have a straightforward sequential counterpart, namely, the serial elision described above.

However, Grace does not support arbitrary concurrent programs. In particular, Grace does not support programs with either of the two following characteristics: (1) a program thread runs infinitely (or until program termination), or (2) threads perform inter-thread communication, e.g., via synchronization primitives like condition variables. Such programs are *inherently concurrent*: their serial elision does not result in a program that exhibits the same semantics. For example, a multithreaded execution of two infinite threads `f()` and `g()` would eventually execute some part of `g()` (assuming fairness), while in the serial elision, `g()` would never execute.

We believe that the loss of expressiveness imposed by this restriction is minimal for Grace's target class of applications, namely, applications running CPU-intensive operations (e.g., games and other applications with available task or data-based concurrency). Anecdotal experience suggests that, in such a context, programmers primarily use these constructs to control concurrency (e.g., to manage thread pools) rather than to divide up work for maximal concurrency.[1] In any event, we believe it can be reasonable to trade reduced expressiveness for ease of use.

3. VM-Based Transactional Memory

Grace achieves concurrent speedup of multithreaded programs by executing threads speculatively, and then committing their updates in program order (described in detail in Section 4). A key challenge is how to enable thread speculation without imposing substantial performance overheads.

What is needed here is some form of transactional memory [15, 26]. Unfortunately, no existing or proposed transactional memory system provides the features that Grace requires: support for long-lived transactions, full isolation of updates from other threads (a.k.a. *strong atomicity* [6]), support for I/O, and low overhead. Software transactional memory is optimized for extremely short transactions (typically demarcated with `atomic` clauses), often precludes the use of I/O inside transactions, and incurs substantial (around 3X) overhead for fully-isolated memory updates inside transaction. Proposed hardware transactional memory systems can provide far lower overhead, but also preclude the use of I/O and bound the number of memory addresses read or written in any transaction.

To meet its requirements, Grace employs a novel **virtual-memory based software transactional memory**. First, it supports fully-isolated transactions of arbitrary length (in terms of the number of memory addresses read or written). Second, its performance overhead is amortized over the length of the transaction rather than incurred on every access, so that threads that run for more than a few milliseconds effectively run at full speed. Third, it supports threads with arbitrary operations, including irrevocable I/O calls

---

[1] Cliff Click, personal communication.

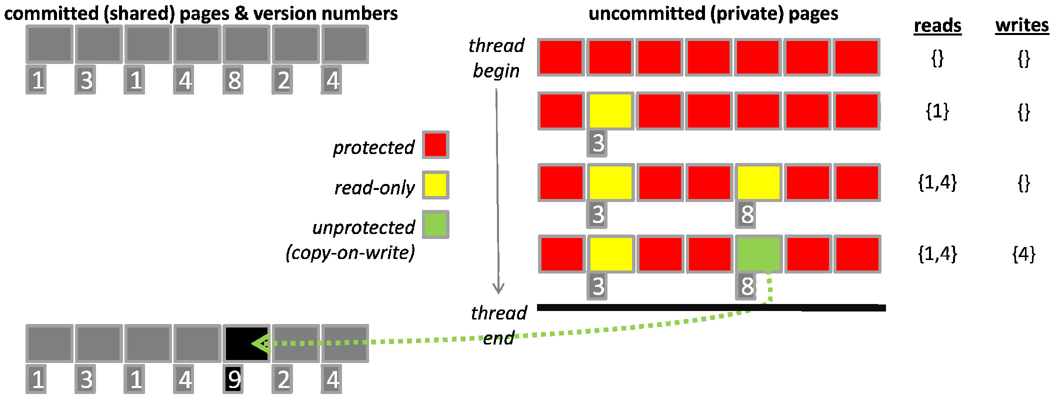
committed (shared) pages & version numbers
1
3
1
4
8
2
4
uncommitted (private) pages
reads
writes
thread begin
{}
{}
{1}
{}
3
protected
read-only
{1,4}
{}
3
8
unprotected (copy-on-write)
{1,4}
{4}
3
8
thread end
1
3
1
4
9
2
4

Figure 3. An overview of thread execution in Grace. Processes emulate threads (Section 3.1) with private mappings to mmapped files that hold committed pages and version numbers for globals and the heap (Sections 3.2 and 3.3). Threads run concurrently but are committed in sequential order: each thread waits until its logical predecessor has terminated in order to preserve sequential semantics (Section 4). Grace then compares the version numbers of the read pages to the committed versions. If they match, Grace commits the writes and increments version numbers; otherwise, it discards the pages and rolls back.

(see Section 4). Finally, this transactional memory system works with existing applications and on commodity hardware.

3.1 Processes as Threads

Our key insight is that we can implement efficient light-weight software transactional memory by treating threads as processes: instead of spawning new threads, Grace `forks` off new processes. Because each "thread" is in fact a separate process, it is possible to use standard memory protection functions and signal handlers to track reads and writes to memory. Grace tracks accesses to memory at a page granularity, trading imprecision of object tracking for speed. Because only the first read or write to each page needs to be tracked, all subsequent operations can proceed at full speed.

To create the illusion that these processes are executing in a shared address space, Grace uses memory mapped files to share the heap and globals across processes. Each process has two mappings to the heap and globals: a shared mapping that reflects the latest committed state, and a local (per-process), copy-on-write mapping that each process uses directly. In addition, Grace establishes a shared and local map of an array of version numbers. Grace uses these version numbers—one for each page in the heap and global area—to decide when it is safe to commit updates.

3.2 Globals

Grace uses a fixed-size file to hold the globals, which it locates in the program image through linker-defined variables. In ELF executables, the symbol `_end` indicates the first address after uninitialized global data. Grace uses an `ld`-based linker script to identify the area that indicates the start of the global data. In addition, this linker script instructs the linker to page align and separate read-only and global areas of memory. This separation reduces the risk of false sharing by ensuring that writes to a global object never conflict with reads of read-only data.

3.3 Heap

Grace also uses a fixed-size mapping (currently 512MB) to hold the heap. It embeds the heap data structure into the beginning of the memory-mapped file itself. This organization elegantly solves the problem of rolling back memory allocations. Grace rolls back memory allocations just as it rolls back any other updates to heap data. Any conflict causes the heap to revert to an earlier version.

However, a naïve implementation of the allocator would give rise to an unacceptably large number of conflicts: any threads that perform memory allocations would conflict. For example, consider a basic freelist-based allocator. Any allocation or deallocation updates a freelist pointer. Thus, any time two threads both invoke `malloc` or `free` on the same-sized object, one thread will be forced to roll back because both threads are updating the page holding that pointer.

To avoid this problem of inadvertent rollbacks, Grace uses a scalable "per-thread" heap organization that is loosely based on Hoard [3] and built with Heap Layers [4]. Grace divides the heap into a fixed number of sub-heaps (currently 16). Each thread uses a hash of its process id to obtain the index of the heap it uses for all memory operations (`malloc` and `free`).

This isolation of each thread's memory operations from the other's allows threads to operate independently most of the time. Each sub-heap is initially seeded with a page-aligned 64K chunk of memory. As long as a thread does not exhaust its own sub-heap's pool of memory, it will operate independently from any other sub-heap. If it runs out of memory, it obtains another 64K chunk from the global allocator. This allocation only causes a conflict with another thread if that thread also runs out of memory during the same period of time.

This allocation strategy has two benefits. First, it minimizes the number of false conflicts created by allocations from the main heap. Second, it reduces false sharing because each thread uses entirely different pages to satisfy object allocation requests. Thus, objects allocated by one thread are never on the same pages as objects allocated by another thread.

To further reduce false sharing, Grace's heap rounds up large object requests (8K or larger) to a multiple of the system page size (4K), ensuring that large objects never overlap, regardless of which thread allocated them.

3.4 Transaction Execution

Figure 3 presents an overview of Grace's execution of a thread as a transaction. Before the program begins, Grace establishes shared and local mappings for the heap and globals. It also establishes the mappings for the version numbers associated with each page in both the heap and global area. Because these pages are zero-filled on-demand, this mapping implicitly initializes the version numbers to zero. A page's version number is incremented only on a successful commit, so it is equivalent to the total number of successful commits of a given page to date.

3.4.1 Starting a Transaction

Grace initiates a transaction at the beginning of program execution and at the start of every thread. When a transaction begins, Grace saves the execution context (program counter, registers, and stack contents) and sets the protection of every page to `PROT_NONE`, so that any access triggers a fault. It also clears both its *read set* and *write set*, which hold the addresses of every page read or written during a transaction.

3.4.2 Transaction Execution

During each transaction, Grace tracks accesses to pages by handling `SEGV` protection faults. The first access to each page is treated as a read. Grace adds the page address to the read set, and then sets the protection for the page to read-only. If the application later writes to the page, Grace adds the page to the write set, and then removes all protection from the page. Thus, in the worst case, a thread incurs two minor page faults for every page that it visits. While protection faults and signals are expensive, their cost is quickly amortized even for relatively short-lived threads (e.g., a millisecond or more).

3.4.3 Ending a Transaction

At the end of each transaction—the end of `main()` or an individual thread—Grace attempts to commit the transaction's updates. It first checks to see whether the read set is empty. If so, it immediately ends the transaction. While this situation may appear to be unlikely, it is common when multiple threads are being created inside a `for` loop, and thus the application is only reading local variables from registers. Allowing transactions to commit in this case is an important optimization, because otherwise, Grace must pause the thread until its immediate predecessor—the last thread it has spawned—has committed. As Section 4 explains, this step is required to provide sequential semantics.

3.4.4 Committing a Transaction

After the preceding thread has ended, Grace establishes locks on all files holding memory mappings with `flock()` and proceeds with a commit. Notice that this serialization only occurs during commits; thread execution is entirely concurrent.

Grace first performs a consistency check, comparing the version numbers for every page in the read set against the committed versions. If they all match, it is safe for Grace to commit the writes, which it does by copying the contents of the page into the corresponding page in the shared image. It then relinquishes the file locks and resumes execution.

3.4.5 Aborting a Transaction

If any of the version numbers do not match, Grace must abort the current execution. Grace issues an `madvise(MADV_DONTNEED)` call on all of the private mappings, which discards any updates and forces all new accesses to use memory from the shared (committed) pages. It then unlocks the file maps and re-executes the thread, copying the saved stack over the current stack and then jumping into the previously saved execution context.

4. Sequential Commit

Grace's virtual memory based transactional memory system provides strong isolation of threads, ensuring that each one executes atomically. However, transactional memory on its own does not guarantee sequential semantics because it does not prescribe any order for these threads.

To provide the appearance of sequential execution of the threads, Grace not only needs to provide isolation of each thread, but also must enforce a particular commit order. Grace employs a simple commit algorithm that provides the effect of a sequential execution.

Grace's commit protocol maintains the following invariant: a thread is only allowed to commit after all of its logical predecessors have completed. The commit algorithm itself is simple. When a thread spawns a child thread, the parent thread stores the child thread ID in a local variable. The parent then continues execution until the end of the thread or if it joins another thread. At this time, if the parent thread has read any memory from the heap or globals (see Section 3.4.3), it waits on a semaphore that the child thread sets when it exits. After the child thread has committed, the parent can attempt to commit its state.

It is straightforward to show that this protocol is required to enforce sequential semantics. Assume that some parent did not need to wait for its immediate child to complete. If so, the parent could commit any updates it makes *after* the thread spawn, inverting the effect of a sequential execution.

4.1 Transactional I/O

Grace's commit protocol not only guarantees sequential semantics but has an additional important benefit. Because Grace imposes an order on thread commits, at any point in time, there is always one thread running that is guaranteed to be able to commit its state: the earliest thread in program order. This property ensures that a Grace program does not suffer from *livelock* caused a failure of any thread to make progress, a property of many transactional memory proposals.

While Grace is able to prevent a number of concurrency errors, it cannot eliminate errors due to interactions beyond the program itself. For example, Grace cannot detect or prevent errors like file system deadlocks (e.g., through `flock()`) or due to message-passing dependencies on distributed systems.

This fact allows Grace to overcome an even more important limitation of most proposed transactional memory systems: it enables the execution of I/O operations in a system with optimistic concurrency. Because some I/O operations are irrevocable (e.g., network reads after writes), most I/O operations appear to be fundamentally at odds with speculative execution. The usual approach is to ban I/O from speculative execution, or to arbitrarily "pick a winner" to obtain a global lock prior to executing its I/O operations. Another option would be to employ operating system support for speculation as in Speculator by Nightingale et al., but this can only support bounded speculative operations [24].

In Grace, each thread buffers its I/O operations and commits them at the same time it commits its updates to memory. However, if a thread attempts to execute an irrevocable I/O operation, Grace forces it to wait for its immediate predecessor to commit. Grace then checks to make sure that its current state is consistent with the committed state. Once both of these conditions are met, the current thread is then *guaranteed* to commit when it terminates. Grace then allows the thread to perform the irrevocable I/O operation, which is now safe because the thread's execution is guaranteed to succeed.

5. Evaluation

Our evaluation answers the following questions:

1. What is the class of programs that work best with Grace?
2. How well does Grace perform on CPU-intensive computation kernels?
3. How effective is Grace against a range of concurrency errors?

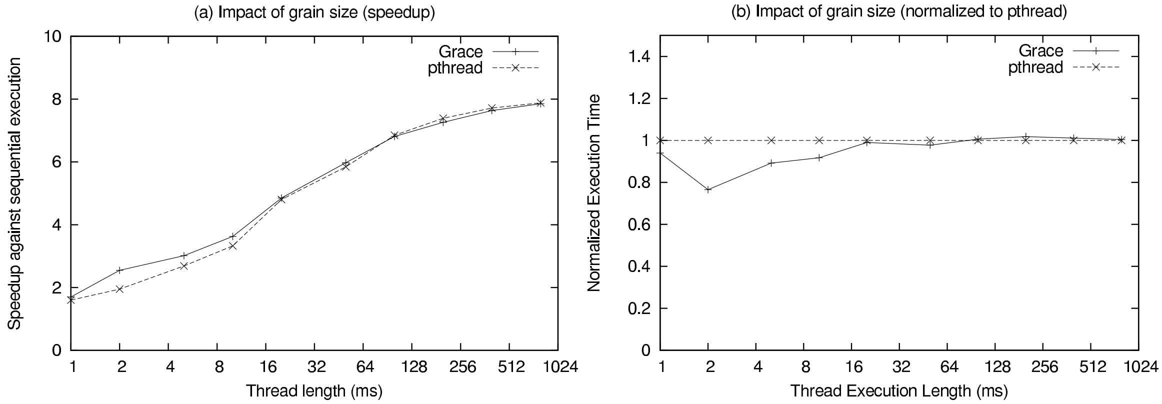

Figure 4. Impact of thread running time on performance: (a) speedup over a sequential version, (b) normalized execution time with respect to `pthreads`.

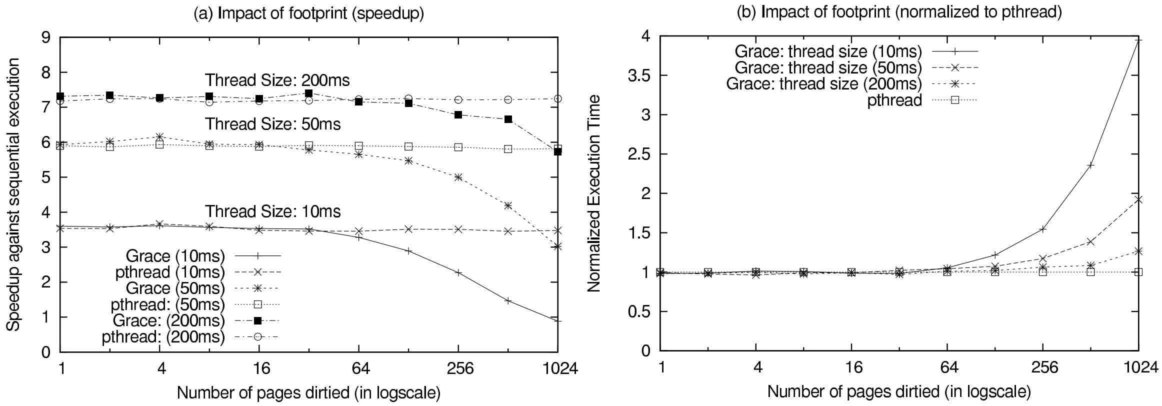

Figure 5. Impact of thread running time on performance (time=10ms, 50ms and 200ms): (a) speedup over a sequential version, (b) normalized execution time with respect to `pthreads`.

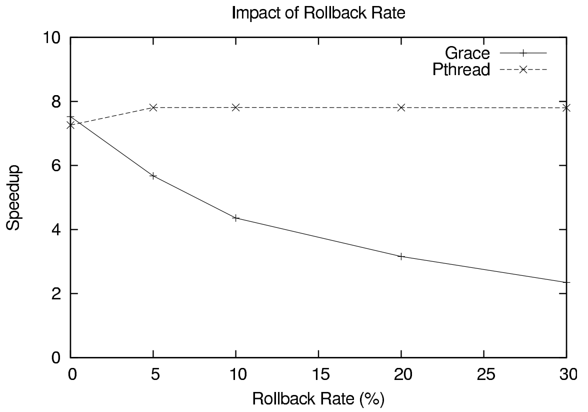

Figure 6. Impact of shared updates, corresponding to the probability that any thread will have to be rolled back, versus a `pthreads` baseline that never rolls back. Even at relatively high rollback rates, Grace is able to take advantage of multiple cores.

5.1 Experimental Methodology

We perform our evaluation on a quiescent dual-processor with 8 gigabytes of RAM. Each processor is a 4-core 64-bit Intel Xeon running at 2.33 Ghz and equipped with a 4MB L2 cache. We compare Grace to the Linux `pthreads` library (NPTL), on Linux 2.6.23 with GNU libc version 2.5.

5.2 Microbenchmarks

To understand the kind of concurrent applications that work best with Grace, we developed a microbenchmark that allows us to explore the effect of changing three key parameters: *grain size*, the running time of each thread; *footprint*, the number of pages updated by a thread; and *degree of sharing*, the proportion of shared pages updated by each thread.

These parameters isolate Grace's overheads. First, the shorter a thread's execution, the more the increased cost of thread spawns in Grace (process creation) should dominate. Second, increasing the number of pages accessed by a thread stresses the cost of Grace's page protection and signal handling. Finally, large numbers of updates of shared state forces Grace to rollback and re-execute threads in order to preserve sequential semantics.

Grain size: We first evaluate the impact of the length of thread execution on Grace's performance. We execute a range of tests, where each thread runs for some fixed number of milliseconds performing arithmetic operations in a tight loop. Notice that this benchmark only exercises the CPU and the cost of thread creation and destruction, because it does not reference heap pages or global data. Each experiment is configured to run for a fixed amount of time: $nTh \times len \times nIter = 16$ seconds, where nTh is the number of threads (16), *len* is the thread running time, and *nIter* is the number iterations.

Figure 4 shows the effect of thread running time on performance. Because we expected the higher cost of thread spawns to degrade Grace's performance relative to `pthreads`, we were surprised to view the opposite effect. We discovered that the operating system's scheduling policy plays an important rule in this set of experiments.

When the size of each thread is extremely small, neither Grace nor `pthreads` make effective use of available CPUs. In both cases, the processes/threads finish so quickly that the load balancer is not triggered and so does not run them on different CPUs. As the thread running time becomes larger, Grace tends to make better of CPU resources, sometimes up to 20% faster. We believe this is because the Linux CPU scheduler attempts to put threads from the same process on one CPU to exploit cache locality, which limits its ability to use more CPUs, but is more liberal in its placement of processes across CPUs. However, once thread running time becomes large enough (over 50ms) for the load balancer to take effect, both Grace and pthread scale well. Figure 4(b) shows that Grace has competitive performance compared to `pthreads`, and the overhead of process creation is never larger than 2%.

Footprint: In order to evaluate the impact of per-thread footprint, we extend the previous benchmark so that each thread also writes a value onto a number of *private* pages, which only exercises Grace's page protection mechanism without triggering rollbacks. We conduct an extensive set of tests, ranging thread footprint from 1 pages to 1024 pages (4MB). This experiment stresses is the worst case scenario for Grace, since each write triggers two page faults.

Figure 5 summarizes the effect of thread footprint over three representative thread running time settings: small (10ms), medium (50ms) and large (200ms). When the thread footprint is not too large ($\leq$ 64 pages), Grace has comparable performance to `pthreads`, with no more than a 5% slowdown. As the thread footprint continues to grow, the performance of Grace starts to degrade due the overhead of page protection faults. However, it stays within an acceptable range for the medium and large thread runtime settings. The overhead of page protection faults only becomes prohibitively large when the thread footprint is large relative to the running time, which is unlikely to be representative of compute-intensive threads.

Degree of sharing: We next measure the impact of sharing on Grace's performance, by having the microbenchmark trigger a range of different rollback rates—that is, the probability that any given thread will need to rollback and re-execute. Figure 6 shows the resulting impact on speedup (each thread runs for 50 milliseconds).

When the rollback rate is low, Grace's performance remains close to that of `pthreads`. The higher the rollback rate, the worse Grace's performance, dropping from nearly a 6-way speedup at 5% to a 2-way speedup at 30%. This result suggests that Grace is most effective for applications with relatively low rollback rates, although it does continue to increase throughput even at relatively high rollback rates.

5.3 CPU-Intensive Benchmarks

We next evaluate Grace's performance on real computation kernels with a range of benchmarks, listed in Table 2. One benchmark, `matmul`—a recursive matrix-matrix multiply routine—comes from the Cilk distribution. We hand-translated this program to use the `pthreads` API (essentially replacing Cilk calls like `spawn` with their counterparts). The remaining benchmarks are multithreaded applications from the Phoenix benchmark suite [25]. These benchmarks represent kernel computations and were designed to be representative of compute-intensive tasks from a range of domains, including enterprise computing, artificial intelligence, and image processing. We use the `pthreads`-based variants of these benchmarks with the largest available inputs.

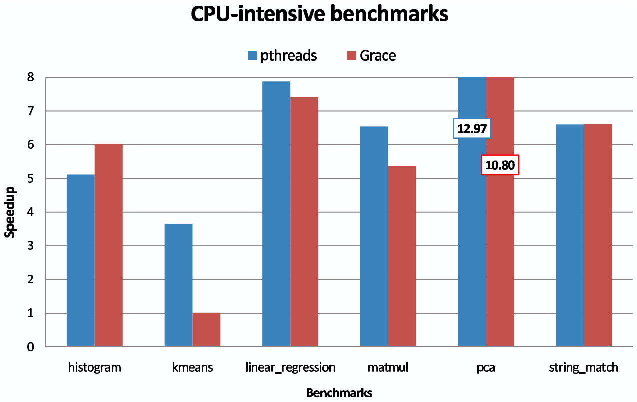

Figure 7. Performance of multithreaded benchmarks running with `pthreads` and Grace on a dual quad-core system. Grace generally performs nearly as well as the `pthreads` version while ensuring correct execution.

Benchmark	Description
histogram	Analyzes images' RGB components
kmeans	Iterative clustering of 3-D points
linear_regression	Computes best fit line for set of points
matmul	Recursive matrix-multiply [11]
pca	Principal component analysis on matrix
string_match	Searches file for encrypted word

Table 2. CPU-intensive multithreaded benchmark suite.

While Grace ran these programs correctly, we made minor modifications to them to avoid false sharing that would otherwise preclude scalability. For `matmul`, we made a one-line change to the code to increase the base matrix size of the recursion. This modification not only reduces false sharing across the threads—critical to avoid excessive rollbacks—but also improves the baseline performance of the benchmark by around 8% by improving its cache utilization.

Most of the remaining changes were local modifications to avoid false sharing, requiring the addition or modification of just one or two lines of code. In several of the benchmarks, we replaced some heap allocations with stack allocations—especially of the `pthread_t` data structure—to reduce the risk of conflicts. The most pervasive change was adding padding to the data structure used to pass in thread arguments. Increasing the size of this structure to 4K eliminated most of the rollbacks, allowing them to scale.

The most substantial change we made was to the `pca` benchmark. This benchmark divides work dynamically across a number of threads, with each thread updating a global variable that indicates which row of a matrix to process next. When executing with Grace, which preserves the effect of a serial execution, the first thread would end up performing all of the computations. To enable the program to scale, we statically partition the work by providing each thread with a range of rows to compute. This modification had little impact on the `pthreads` version but dramatically improved the scalability with Grace. These changes required the addition of 16 lines of code.

Figure 7 shows the result of running these applications, graphed as their speedup over a serial execution. The Grace-based versions achieve comparable performance while at the same time guaran-

Bug type	Benchmark description
deadlock	Cyclic lock acquisition
race condition	Race condition example from Lucia et al. [22]
atomicity violation	Atomicity violation drawn from MySQL [21]
order violations	Order violation from Mozilla 0.8 [20]

Table 3. Error benchmark suite.

teeing the absence of concurrency errors. The average speedup for Grace is 6.2X, while the average speedup for `pthreads` is 7.13X.

There are two notable outliers. The first one is `pca`, which exhibits superlinear speedups both for Grace and `pthreads`. The superlinear speedup is due to improved cache locality caused by the division of the computation into smaller chunks across multiple threads.

The more interesting case is the `kmeans` benchmark. While `kmeans` achieves a modest speedup with `pthreads` (3.65X), it is the only benchmark we tested that exhibits no speedup with Grace (1.02X). The problem is false sharing on the heap. The `kmeans` benchmark iteratively clusters points in 3-dimensional space, repeatedly updating several arrays that track the number of points assigned to each cluster as well as the cluster itself. Because of the large number of points (100,000) and the small size of the point data structure (3 integers), padding it to avoid false sharing is not practical. While it would be possible to modify `kmeans` to achieve scalability with Grace, this modification would entail a substantial modification. As currently written, `kmeans` falls into the space of problems for which Grace is not a reasonable approach.

5.4 Concurrency Bug Benchmarks

To verify Grace's ability to cope with concurrency bugs, we compiled a bug suite primarily drawn from actual bugs described in previous work on error detection and listed in Table 3 [20, 21, 22]. Because concurrency errors are by their nature non-deterministic and occur only for particular thread interleavings, we inserted delays (via the `usleep` function call) at key points in the code. These delays dramatically increase the likelihood of encountering these errors, allowing us to compare the effect of using Grace and `pthreads`.

5.4.1 Deadlocks

Figure 8 illustrates a deadlock error caused by cyclic lock acquisition. This example spawns two threads that each attempt to acquire two locks `A` and `B`, but in different orders: thread 1 acquires lock `A` then lock `B`, while thread 2 acquires lock `B` then lock `A`. When using `pthreads`, these threads deadlock if both of them manage to acquire their first locks, because each of the threads is waiting to acquire a lock held by the other thread. Inserting `usleep` after these locks makes this program deadlock reliably under `pthreads`. However, because Grace's atomicity and commit protocol lets it treat locks as no-ops, this program never deadlocks with Grace.

5.4.2 Race conditions

We next adapt an example from Lucia et al. [22], removing the lock in the original example to trigger a race. Figure 9 shows two threads both executing `increment`, which increments a shared variable `counter`. However, because access to `counter` is unprotected, both threads could read the same value and so can lose an update. Running this example under `pthreads` with an injected delay always exhibits this race, printing `0,0,1,1`. By contrast, Grace prevents the race by executing each thread deterministically, and always outputs the sequence `0,1,1,2`.

```
thread1 () {
  lock (A);
  // usleep();
  lock (B);
  // ...do something
  unlock (B);
  unlock (A);
}

thread2 () {
  lock (B);
  // usleep();
  lock (A);
  // ...do something
  unlock (A);
  unlock (B);
}
```

Figure 8. Deadlock example. This code has a cyclic lock acquisition pattern that triggers a deadlock under `pthreads` while running to completion with Grace.

```
// shared variable
int counter = 0;

increment() {
  print (counter);
  int temp = counter;
  temp++;
  // usleep();
  counter = temp;
  print (counter);
}

thread1() { increment(); }
thread2() { increment(); }
}
```

Figure 9. Race condition example: the race is on the variable `counter`, where the first update can be lost. Under Grace, both increments always succeed.

5.4.3 Atomicity Violations

To verify Grace's ability to cope with atomicity violations, we adapted an atomicity violation bug taken from MySQL's InnoDB module, described by Lu et al. [21]. In this example, shown in Figure 10, the programmer has failed to properly protect access to the global variable `thd`. If the scheduler executes the statement labeled `S3` in thread 2 immediately after thread 1 executes `S1`, the program will dereference NULL and fail.

Inserting a delay between statements `S1` and `S2` causes every execution of this code with `pthreads` to segfault because of a NULL dereference. With Grace, threads always appear to execute atomically, so the program always performs correctly.

5.4.4 Order violations

Finally, we consider order violations, which were recently identified as a common concurrency error by Lu et al. [21]. An order violation occurs when the program runs correctly under one ordering of thread executions, but incorrectly under a different schedule. Notice that order violations are orthogonal to atomicity violations: an order violation can occur even when the threads are entirely atomic.

Figure 11 presents a case where the programmer's intended order is not guaranteed to be obeyed by the scheduler. Here, if thread 2 manages to write into `proc_info` before it has been

```
// thread1
S1: if (thd->proc_info) {
      // usleep();
S2:   fputs (thd->proc_info,..)
    }

// thread2
S3: thd->proc_info = NULL;
```

Figure 10. An atomicity violation from MySQL [21]. A faulty interleaving can cause this code to trigger a segmentation fault due to a NULL dereference, but by enforcing atomicity, Grace prevents this error.

```
char * proc_info;

thread1() {
  // ...
  // usleep();
  proc_info = malloc(256);
}

thread2() {
  // ...
  strcpy(proc_info,"abc");
}

main() {
  spawn thread1();
  spawn thread2();
}
```

Figure 11. An order violation bug. If thread 2 executes before thread 1, it writes into unallocated memory. Grace ensues that thread 2 always executes after thread 1, avoiding this error.

```
int foo;

thread1() {
 foo = 0;
}

main() {
S1:  spawn thread1();
     //  usleep();
S2:  foo = 1;
     // ...
     assert (foo == 0);
}
```

Figure 12. An order violation that Grace cannot fix. Here, the intended effect violates sequential semantics, so the error is not fixed but occurs reliably.

allocated by thread 1, it will cause a segfault. However, because the scheduler is unlikely to be able to schedule thread 2 before thread 1 has executed the allocation call, this code will generally work correctly. Nonetheless, it will occasionally fail, and injecting `usleep()` forces it to fail reliably. With Grace, this microbenchmark always always runs correctly, because Grace ensures that the spawned threads exhibit sequential semantics. Thus, thread 2 can commit only after thread 1 completes, preventing the order violation.

Interestingly, while Grace prescribes the order of program execution, Figure 12 shows that the expected order might *not* be the order that Grace enforces. In this example, modeled after an order violation bug from Mozilla, the `pthreads` version is almost certain to execute statement `S2` immediately after `S1`; that is, well before the scheduler is able to run `thread1`. The final value of `foo` will therefore almost always be 0.

However, in the rare event that a context switch occurs immediately after `S1`, the thread may get a chance to run first, leaving the value of `foo` at 1 and causing the assertion to fail. Such a bug would be unlikely to be revealed during testing and could lead to failures in the field that would be exceedingly difficult to locate.

However, with Grace, the final value of `foo` will always be 1, because that result corresponds to the result of a sequential execution of `thread1`. While this result might not have been the one that the programmer expected, using Grace would have made the error both obvious and repeatable, and thus easier to fix.

6. Related Work

The literature relating to concurrent programming is vast. We briefly describe the most closely-related work here.

Transactional memory

The area of transactional memory, first proposed by Herlihy and Moss for hardware [15] and for software by Shavit and Touitou [26], is now a highly active area of research. Larus and Rajwar's book provides an overview of recent work in the area [18].

Fraser and Harris's transaction-based *atomic blocks* [13] are a programming construct that has been the model for many subsequent language proposals. However, the semantics of these language proposals are surprisingly complex. For example, Shpeisman et al. [27] show that proposed "weak" transactions can give rise to unanticipated and unpredictable effects in programs that would not have arised when using lock-based synchronization. With Grace, program semantics are straightforward and unsurprising.

Grossman has drawn an analogy between transactional memory and garbage collection [12], pointing out similar mechanisms and approaches, but the analogy between garbage collection and Grace is closer. Garbage collection presents the programmer with a simple memory model, providing the illusion of an infinite store but reclaiming memory without the need for programmer intervention. Likewise, Grace presents the programmer with a simple programming model, providing the illusion of a sequential execution, while implementing it with concurrency under the covers.

6.1 Concurrent programming models

Cilk [11] is a multithreaded extension of the C programming language. Like Grace, Cilk focuses on the use of multiple threads for CPU intensive workloads, rather than server applications. Unlike Grace, which works with C or C++ binaries, Cilk is restricted to C. Cilk also relies on the programmer to avoid race conditions and other concurrency errors, and there has been work on dynamic tools to locate these errors [2, 7], while Grace automatically prevents them.

A proposed variant of Cilk called "Transactions Everywhere" adds transactions to Cilk by having the compiler insert *cutpoints* (transaction end and begin) at various points in the code, including at the end of loop iterations. While this approach reduces the exposure to concurrency errors, it does not prevent them, and data race detection in this model has been shown to be an NP-complete problem [16].

Automatic mutual exclusion, or AME, is a recently-proposed programming model currently being implemented at Microsoft Research Cambridge. It is a language extension to C# that assumes that all shared state is private unless otherwise indicated [17]. These guarantees are weaker than Grace's, in that AME programmers can still generate code with concurrency errors. AME has a richer concurrent programming model than Grace that makes it more flexible, but precludes a sequential interpretation. In fact, the semantics of AME are complicated enough to warrant publication in POPL [1]. By contrast, Grace's semantics are straightforward and thus likely easier for programmers to understand.

von Praun et al. present Implicit Parallelism with Ordered Transactions (IPOT), that describes a programming model, like Grace, that supports speculative concurrency and enforces determinism [28]. However, unlike Grace, IPOT requires a completely new programming language, with a wide range of constructs including variable type annotations and constructs to support speculative and explicit parallelism. In addition, IPOT would require special hardware and compiler support, while Grace operates on existing C/C++ programs that use standard thread constructs.

The most closely related work to Grace in terms of its programming model is safe futures for Java, by Welc et al. [29]. A future denotes an expression that may be evaluated in parallel with the rest of the program; when the program uses the expression's value, it waits for the future to complete execution before continuing. As with Grace's threads, safe futures ensure that the concurrent execution of futures provides the same effect as evaluating the expressions sequentially. However, the safe future system assumes that writes are rare in futures (by contrast with threads), and uses an object-based versioning system optimized for this case. It also requires compiler support and integration with a garbage-collected environment.

Grace's use of virtual memory primitives to support speculation is a superset of the approach used by behavior-oriented parallelism (BOP) [9]. BOP allows programmers to specify possibly parallelizable regions of code in sequential programs, and uses a combination of compiler analysis and the strong isolation properties of processes to ensure that speculative execution never prevents a correct execution. While BOP seeks to increase the performance of sequential code by enabling safe, speculative parallelism, Grace provides sequential semantics for multithreaded programs, which it executes concurrently. Unlike BOP, however, Grace does not require programmer intervention.

7. Future Work

Our current implementation of Grace leaves plenty of room for optimization. We outline our planned directions here, first in the runtime space, and then with compiler support.

The current runtime system immediately spawns as many threads as the program requests. In addition to potentially consuming excessive resources, this approach runs the risk of impairing sequential performance in the worst-case. Consider an execution where each thread conflicts with every other thread. If the number of threads spawned exactly matches the number of available cores, then the first thread in serial order will always be able to proceed at full speed. However, if there are many more threads than cores, the first thread will have to share its processing with these other "doomed" threads, degrading performance. We plan to alter our runtime system to throttle thread creation, so that it never creates more threads than available processors.

While conflicts cause rollbacks, they also provide information that can be fed back into the runtime system so it can intelligently reschedule threads. For example, the runtime system could partition threads into conflicting sets, and then only schedule the first thread (in serial order) from each of these sets. This algorithm would maximize the utilization of available parallelism by preventing repeated rollbacks.

Another desirable enhancement to the runtime system would allow Grace to report memory areas that are the source of frequent conflicts. This information can guide programmers as they tune their programs for higher performance.

We are also considering custom memory allocation algorithms to eliminate false sharing. We plan to leverage a mechanism proposed by Dhurjati and Adve that places individual objects at different offsets on a different virtual pages, with multiple virtual pages mapped to the same physical page to conserve space [8]. While this mechanism was designed to detect dangling pointer errors, we can use it to prevent false sharing in Grace's page-based transactional memory system.

Finally, we believe that Grace's sequential semantics provides the opportunity for novel compiler optimizations. For example, Grace's sequential semantics could enable *cross-thread* optimizations that can hoist small conflicting memory operations out of multiple threads and combine them into a sequential operation in the final thread in a sequence.

8. Conclusion

This paper presents Grace, a runtime system that eliminates a broad class of concurrency errors, including deadlocks, race conditions, atomicity violations, and order violations. Grace is a plug-in substitute for the `pthreads` library and is especially well-suited for fine to medium-grained CPU-intensive concurrent computations. It works with existing C and C++ programs that use fork-join parallelism. It achieves good scalability and performance on multicore systems, while ensuring the absence of a wide range of difficult-to-debug errors.

We view Grace as analogous to "garbage collection for concurrency." Garbage collection effectively eliminates a range of memory errors while incurring a cost that is acceptable in many (but not all) contexts. Similarly, we believe that Grace shows promise as a runtime system approach to increase productivity and application reliability by eliminating most concurrency errors.

9. Acknowledgements

The authors would like to thank Ben Zorn for his valuable feedback during the development of the ideas that led to Grace, and to Luis Ceze for graciously providing benchmarks. This material is based upon work supported by Intel, Microsoft Research, and the National Science Foundation under CAREER Award CNS-0347339 and CNS-0615211. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 63–74, New York, NY, USA, 2008. ACM.

[2] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson. On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 133–144, New York, NY, USA, 2004. ACM.

[3] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 117–128, New York, NY, USA, Nov. 2000. ACM.

[4] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2001)*, pages 114–124, New York, NY, USA, June 2001. ACM.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996.

[6] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *WDDD '05: 4th Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.

[7] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in cilk programs that use locks. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 298–309, New York, NY, USA, 1998. ACM.

[8] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, pages 269–280, Washington, DC, USA, 2006. IEEE Computer Society.

[9] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 223–234, New York, NY, USA, 2007. ACM.

[10] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349, New York, NY, USA, 2003. ACM.

[11] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, 1998.

[12] D. Grossman. The transactional memory / garbage collection analogy. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 695–706, New York, NY, USA, 2007. ACM.

[13] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM.

[14] J. W. Havender. Avoiding deadlock in multitasking systems. *IBM Systems Journal*, 7(2):74–84, 1968.

[15] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

[16] K. Huang. Data-race detection in transactions-everywhere parallel programming. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 2003.

[17] M. Isard and A. Birrell. Automatic mutual exclusion. In *HotOS XI: 11th Workshop on Hot Topics in Operating Systems*, Berkeley, CA, May 2007.

[18] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[19] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[20] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *SOSP '07: Proceedings of the Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, pages 103–116, New York, NY, USA, 2007. ACM.

[21] 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, New York, NY, USA, 2008. ACM.

[22] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and surviving atomicity violations. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*, New York, NY, USA, June 2008. ACM Press.

[23] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 1992.

[24] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. *ACM Trans. Comput. Syst.*, 24(4):361–392, 2006.

[25] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 13th Intl. Symposium on High-Performance Computer Architecture (HPCA)*, feb 2007.

[26] 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, New York, NY, USA, 1995. ACM.

[27] 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 the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 78–88, New York, NY, USA, 2007. ACM.

[28] C. von Praun, L. Ceze, and C. Caşcaval. Implicit parallelism with ordered transactions. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–89, New York, NY, USA, 2007. ACM.

[29] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN Conference on Object oriented Programming, Systems, Languages, and applications*, pages 439–453, New York, NY, USA, 2005. ACM.