Grace: Safe Multithreaded Programming for C/C++

Emery D. Berger Ting Yang Tongping Liu Gene Novark

Dept. of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003
{emery,tingy,tonyliu,gnovark}@cs.umass.edu

Abstract

The shift from single to multiple core architectures means that programmers must write concurrent, multithreaded programs in order to increase application performance. 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.

This paper presents Grace, a software-only runtime system that eliminates concurrency errors for a class of multithreaded programs: those based on fork-join parallelism. By turning threads into processes, leveraging virtual memory protection, and imposing a sequential commit protocol, Grace provides programmers with the appearance of deterministic, sequential execution, while taking advantage of available processing cores to run code concurrently and efficiently. Experimental results demonstrate Grace's effectiveness: with modest code changes across a suite of computationally-intensive benchmarks (1–16 lines), Grace can achieve high scalability and performance while preventing concurrency errors.

Categories and Subject Descriptors D.1.3 [*Software*]: Concurrent Programming–Parallel Programming; D.2.0 [*Software Engineering*]: Protection mechanisms

General Terms Performance, Reliability

Keywords Concurrency, determinism, deterministic concurrency, fork-join, sequential semantics

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 This phenomenon has led to a major shift in computer architecture, where single-core CPUs have been replaced by CPUs consisting of a number of processing cores.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 [29]. For example, multithreaded programs that fail to employ a canonical locking order can *deadlock* [16]. Because the interleavings of threads are non-deterministic, programs that do not properly lock shared data structures can suffer from *race conditions* [30]. A related problem is *atomicity violations*, where programs may lock and unlock individual objects but fail to ensure the atomicity of multiple object updates [14]. Another class of concurrency errors is *order violations*, where a program depends on a sequence of threads that the scheduler may not provide [26].

This paper introduces **Grace**, a runtime system that eliminates concurrency errors for a particular class of multithreaded programs: those that employ fully-structured, or fork-join based parallelism to increase performance.

While fork-join parallelism does not capture all possible parallel programs, it is a popular model of parallel program execution: systems based primarily on fork-join parallelism include Cilk, Intel's Threading Building Blocks [35], OpenMP, and the fork-join framework proposed for Java [24]. Perhaps the most prominent use of fork-join parallelism today is in Google's Map-Reduce framework, a library that is used to implement a number of Google services [9, 34]. However, none of these prevent concurrency

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

errors, which are difficult even for expert programmers to avoid [13].

Grace manages the execution of multithreaded programs with fork-join parallelism so that they become *behaviorally equivalent* to their sequential counterparts: every thread spawn becomes a sequential function invocation, and locks become no-ops.

This execution model eliminates most concurrency errors that can arise due to multithreading (see Table 1). By converting lock operations to no-ops, Grace eliminates deadlocks. By committing state changes deterministically, Grace eliminates race conditions. By executing threads in program order, Grace eliminates atomicity violations and greatly reduces the risk of order violations. Finally, by enforcing sequential semantics and thus sequential consistency, Grace eliminates the need for programmers to reason about complex underlying memory models.

To exploit available computing resources (multiple CPUs or cores), Grace employs a combination of *speculative* thread execution, together with a sequential commit protocol that ensures sequential semantics. By replacing threads with processes and providing appropriate shared memory mappings, Grace leverages process isolation, page protection and virtual memory mappings to provide isolation and full support for speculative execution on conventional hardware.

Under Grace, threads execute optimistically, writing their updates speculatively but locally. 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 program order from the conflicting set of threads. Rather than executing threads atomically, Grace uses events like thread spawns and joins as commit points that divide execution into pieces of work, and enforces a deterministic execution that matches a sequential execution.

This deterministic execution model allows programmers to reason about their programs as if they were serial programs, making them easier to understand and debug [2]. Traditionally, when programmers reorganize thread interactions to obtain reasonable performance (e.g., by selecting an appropriate grain size, reducing contention, and minimizing the size of critical sections), they run risk of introducing new, difficult-to-debug concurrency errors. Grace not only lifts the burden of using locks or atomic sections on programmers, but also allows them to optimize performance without the risk of compromising correctness.

We evaluate Grace's performance on a suite of CPU-intensive, fork-join based multithreaded applications, as well as a microbenchmark we designed to explore the space of programs for which Grace will be most effective. We also evaluate Grace's ability to avoid a selection of concurrency bugs taken from the literature. Experimental results show that Grace ensures the correct execution of otherwise-buggy concurrent code. While Grace does not guarantee concurrency for unchanged programs, we found that minor changes (1–16 lines of source code) were enough to allow Grace to achieve comparable scalability and performance to the standard (unsafe) threads library across most of our benchmark suite, while ensuring safe execution.

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

Figure 2. Its sequential counterpart (elided operations struck out).

The remainder of this paper is organized as follows. Section 2 outlines the sequential semantics that Grace provides. Section 3 describes the software mechanisms that Grace uses to enable speculative execution with low overhead. Section 4 presents the commit protocol that enforces sequential semantics, and explains how Grace can support I/O together with optimistic concurrency. Section 5 describes our experimental methodology. Section 6 then presents experimental results across a benchmark suite of concurrent, multithreaded computation kernels, a microbenchmark that explores Grace's performance characteristics, and a suite of concurrency errors. Section 7 surveys related work, Section 8 describes future directions, and Section 9 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 subset of the `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 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 calls to spawn and sync (struck out) are 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 concur-

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.

rency 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 enforces deterministic execution of programs that rely on "fully structured" or *fork-join parallelism*, such as master-slave parallelism or parallelized divide-and-conquer, where each division step forks off children threads and waits for them to complete. These programs have a straightforward sequential counterpart: the serial elision described above. For convenience, Grace exports its operations as a subset of the popular POSIX `pthreads` API, although it does not support the full range of pthreads semantics.

Grace's current target class of applications is applications running fork-join style, CPU-intensive operations. At present, Grace is not suitable for *reactive programs* like server applications, and does not support programs with *concurrency control* through synchronization primitives like condition variables, or other programs that are *inherently concurrent*: that is, their serial elision does not result in a program that exhibits the same semantics.

Note that while Grace is able to prevent a number of concurrency errors, it cannot eliminate errors that are external to the program itself. For example, Grace does not attempt to detect or prevent errors like file system deadlocks (e.g., through `flock()`) or due to message-passing dependencies on distributed systems.

3. Support for Speculation

Grace achieves concurrent speedup of multithreaded programs by executing threads speculatively, then committing their updates in program order (see Section 4). A key challenge is how to enable low-overhead thread speculation in C/C++.

One possible candidate would be some form of transactional memory [17, 36]. Unfortunately, no existing or proposed transactional memory system provides all of the features that Grace requires:

- full compatibility with C and C++ and commodity hardware,
- full support for long-lived transactions,
- complete isolation of updates from other threads, i.e., *strong atomicity* [6],
- support for irrevocable actions including I/O and memory management, and
- extremely low runtime and space overhead.

Existing software transactional memory (STM) systems are optimized for short transactions, generally demarcated with `atomic` clauses. These systems do not effectively support long-lived transactions, which either abort whenever conflicting shorter-lived transactions commit their state first, or must switch to single-threaded mode to ensure fair progress. They also often preclude the use of irrevocable actions (e.g., I/O) inside transactions [40].

Most importantly, STMs typically incur substantial space and runtime overhead (around 3X) for fully-isolated memory updates inside transactions. While compiler optimizations can reduce this cost on unshared data [37], transactions must still incur this overhead on shared data.

In the absence of sophisticated compiler analyses, we found that the overheads of conventional log-based STMs are unacceptable for the long transactions that Grace targets. We attempted to employ Sun's state-of-the-art TL2 STM system [11] using Pin [28] to instrument reads and writes that call the appropriate TL2 function (transactional reads and writes). Unlike most programs using TL2 (including the STAMP transaction benchmark suite), the "transactions" here comprise every read and write. In all of our tests, the length of the logs becomes excessive, causing TL2 to run out of memory.

To meet its requirements, Grace employs a novel virtual-memory based software transactional memory with a number of key features. First, it supports fully-isolated threads of arbitrary length (in terms of the number of memory addresses read or written). Second, its performance overhead is amortized over the length of a thread's execution rather than being 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 (see Section 4). Finally, Grace works with existing C and C++ applications running on commodity hardware.

3.1 Processes as Threads

Our key insight is that we can implement efficient software transactional memory *by treating threads as processes*: in-

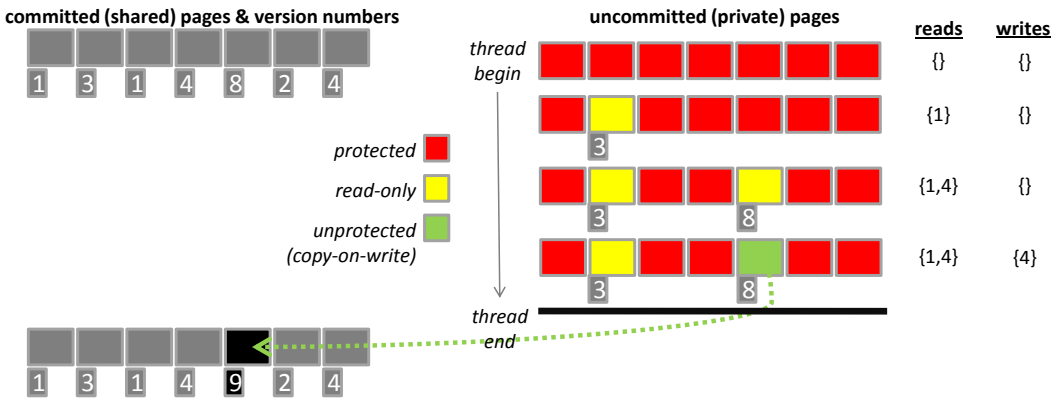

Figure 3. An overview of execution in Grace. Processes emulate threads (Section 3.1) with private mappings to mmapped files that hold committed pages and version numbers for globals and the heap (Sections 3.2 and 3.3). Threads run concurrently but are committed in sequential order: each thread waits 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.

stead 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. Crucially, because only the first read or write to each page needs to be tracked, all subsequent operations 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 Organization

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 avoids an important source of false sharing. Because each thread uses different pages to satisfy object allocation requests, objects allocated by one thread are unlikely to be on the same pages as objects al-

located by another thread (except when both threads hash to the same sub-heap). This heap organization ensures that conflicts only arise when allocated memory from a parent thread is passed to children threads, or when objects allocated by one thread are then accessed by another, later 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 Thread Execution

Figure 3 presents an overview of Grace's execution of a thread. This example is simplified: recall that Grace does not always execute entire threads atomically. Atomic execution begins at program startup (`main()`), and whenever a new thread is spawned. It ends (is committed) not only when a thread ends, but also when a thread spawns a child or joins (syncs) a previously-spawned child thread.

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 its total number of successful commits to date.

Initialization

Grace initializes state tracking at the beginning of program execution and at the start of every thread by invoking `atomicBegin` (Figure 4). Grace first 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* and *write sets*, which hold the addresses of every page read or written.

Execution

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), as Section 6.2 shows.

Completion

At the end of each atomically-executed region—the end of `main()` or an individual thread, right before a thread spawn, and right before joining another thread—Grace invokes `atomicEnd` (Figure 5), which attempts to commit all updates by calling `atomicCommit` (Figure 6). It first checks to see whether the read set is empty, at which point it can safely commit. 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 commits in this case is an important optimization, because otherwise, Grace would have to 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.

```
void atomicBegin (void) {
  // Roll back to here on abort.
  // Saves PC, registers, stack.
  context.commit();
  // Reset pages seen (for signal handler).
  pages.clear();
  // Reset global and heap protection.
  globals.begin();
  heap.begin();
}
```

Figure 4. Pseudo-code for atomic begin.

Committing

Once a thread has finished executing and any logically preceding threads have already completed, Grace establishes locks on all files holding memory mappings using interprocess mutexes (in the call to `lock()`) and proceeds to check whether it is safe to commit its updates. 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 both for the heap and the globals. If they all match, it is safe for Grace to commit the writes, which it does by copying the contents of each page into the corresponding page in the shared images. It then relinquishes the file locks and resumes execution.

If, however, any of the version numbers do not match, Grace invokes `atomicAbort` to abort the current execution (Figure 5). Grace issues a `madvise(MADV_DONTNEED)` call to discard any updates to the heap and globals, which forces all new accesses to use memory from the shared (committed) pages. It then unlocks the file maps and re-executes, copying the saved stack over the current stack and then jumping into the previously saved execution context.

4. Sequential Commit

Grace provides strong isolation of threads, ensuring that they do not interfere with each other when executing speculatively. However, this isolation on its own does not guarantee sequential semantics because it does not prescribe any order.

```
void atomicEnd (void) {
  if (!atomicCommit())
    atomicAbort();
}

void atomicAbort (void) {
  // Throw away changes.
  heap.abort();
  globals.abort();
  // Jump back to saved context.
  context.abort();
}
```

Figure 5. Pseudo-code for atomic end and abort.

```
bool atomicCommit (void) {
  // If haven't read or written anything,
  // we don't have to wait or commit;
  // update local view of memory & return.
  if (heap.nop() && globals.nop()) {
    heap.updateAll();
    globals.updateAll();
    return true;
  }
  // Wait for immediate predecessor
  // to complete.
  waitExited(predecessor);
  // Now try to commit state. Iff we succeed,
  // return true.
  // Lock to make check & commit atomic.
  lock();
  bool committed = false;
  // Ensure heap and globals consistent.
  if (heap.consistent() &&
      globals.consistent()) {
    // OK, all consistent: commit.
    heap.commit();
    globals.commit();
    xio.commit(); // commits buffered I/O
    committed = true;
  }
  unlock();
  return committed;
}
```

Figure 6. Pseudo-code for atomic commit.

To provide the appearance of sequential execution, 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 algorithm implements the following policy: a thread is only allowed to commit after all of its logical predecessors have completed. It might appear that such a commit protocol would be costly to implement, possibly requiring global synchronization and complex data structures. Instead, Grace employs a simple and efficient commit algorithm, which threads the tree of dependencies through all the executing threads to ensure sequential semantics.

```
void * spawnThread (threadFunction * fn,
                    void * arg) {
  // End atomic section here.
  atomicEnd();
  // Allocate shared mem object
  // to hold thread's return value.
  ThreadStatus * t =
    new (allocateStatus()) ThreadStatus;
  // Use fork instead of thread spawn.
  int child = fork();
  if (child) {
    // I'm the parent (caller of spawn).
    // Store the tid to allow later sync
    // on child thread.
    t->tid = child;
    // The spawned child is new predecessor.
    predecessor = child;
    // Start new atomic section
    // and return thread info.
    atomicBegin();
    return (void *) t;
  } else {
    // I'm the child.
    // Set thread id.
    tid = getpid();
    // Execute thread function.
    atomicBegin();
    t->retval = fn(arg);
    atomicEnd();
    // Indicate that process has ended
    // to alert its successor (parent)
    // that it can continue.
    setExited();
    // Done.
    _exit (0);
  }
}
```

Figure 7. Pseudo-code for thread creation. Note that the actual Grace library wraps thread creation and joining with a `pthreads`-compatible API.

Executing threads form a tree, where the post-order traversal specifies the correct commit order. Parents must wait for their last-spawned child, children wait either for their preceding sibling if it exists, or the parent's previous sibling. Grace threads the tree of dependencies through all the executing threads to ensure sequential semantics.

The key is that only thread spawns affect commit dependence, and then only affect those of the newly-spawned child and parent processes. Each new child always appears immediately before its parent in the post-order traversal. Updating the predecessor values is akin to inserting the child pro-

```
void joinThread (void * v, void ** result) {
  ThreadStatus * t = (ThreadStatus *) v;
  // Wait for a particular thread
  // (if argument non-NULL).
  if (v != NULL) {
    atomicEnd();
    // Wait for 'thread' to terminate.
    if (t->tid)
      waitExited (t->tid);
    // Grab thread result from status.
    if (result != NULL) {
      *result = t->retval;
      // Reclaim memory.
      freeStatus(t);
    }
    atomicBegin();
  }
}
```

Figure 8. Pseudo-code for thread joining.

cess into a linked list representing this traversal. Each child sets its predecessor to the parent's predecessor (which happens automatically because of the semantics of `fork`), and then the parent sets its predecessor to the child's ID (see Figure 7).

The parent then continues execution until the next commit point (the end of the thread, a new thread spawn, or when it joins another thread). At this time, if the parent thread has read any memory from the heap or globals (see Section 3.4), it then waits on a semaphore that the child thread sets when it exits (see Figures 7 and 8).

4.1 Transactional I/O

Grace's commit protocol not only enforces sequential semantics but also has an additional important benefit. Because Grace imposes an order on thread commits, 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 Grace programs cannot suffer from *livelock* caused by a failure of any thread to make progress, a problem with some transactional memory 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.

In Grace, each thread buffers its I/O operations and commits them at the same time it commits its updates to memory, as shown in Figure 6. 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. Methodology

We perform our evaluation on a quiescent 8-core system (dual processor with 4 cores), and 8GB of RAM. Each processor is a 4-core 64-bit Intel Xeon running at 2.33 Ghz 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.

5.1 CPU-Intensive Benchmarks

We 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). We performed the same translation for the remaining Cilk benchmarks, but because they use unusually fine-grained threads, none of them scaled when using `pthreads`.

The remaining benchmarks are from the Phoenix benchmark suite [34]. 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.

In addition to describing the benchmarks, Table 2 also presents detailed benchmark characteristics measured from their execution with Grace, including the total number of commits and rollbacks, together with the average number of pages read and written and average wall-clock time *per atomic region*. With the exception of `matmul` and `kmeans`, the benchmarks read and write from relatively few pages in each atomic region. `matmul` has a coarse grain size and large footprint, but has no interference between threads due to the regular structure of its recursive decomposition. On the other hand, `kmeans` has a benign race which forces Grace to trigger numerous rollbacks (see Section 6.1).

5.1.1 Modifications

All of these programs run correctly with Grace "out of the box", but as we explain below, they required slight tweaking to allow them to scale (with *no* modifications, none of the programs scale). These changes were typically short and local, requiring one or two lines of new code, and required no understanding of the application itself. Several of

		(average per atomic region)				
Benchmark	**Description**	**Commits**	**Rollbacks**	**Pages Read**	**Pages Written**	**Runtime (ms)**
histogram	Analyzes images' RGB components	9	0	7.3	5.9	1512.3
kmeans	Iterative clustering of 3-D points	6273	4887	404.5	2.3	8.7
linear_regression	Computes best fit line for set of points	9	0	5.6	4.8	1024.0
matmul	Recursive matrix-multiply	11	0	4100	1865	2359.4
pca	Principal component analysis on matrix	22	0	3.1	2.2	0.204
string_match	Searches file for encrypted word	11	0	5.9	4.3	191.1

Table 2. CPU-intensive multithreaded benchmark suite and detailed characteristics (see Section 5.1).

these changes could be mechanically applied by a compiler, though we have not explored this. (We note that the modification of benchmarks to explore new programming models is standard practice, e.g., in papers exploring software transactional memory or map-reduce.)

Thread-creation hoisting / argument padding: In most of the applications, the only modification we made was to the loop that spawned threads. In the Phoenix benchmarks, this loop body typically initializes each thread's arguments before spawning the thread. False sharing on these updates causes Grace to serialize all of threads, precluding scalability. We resolved this either by hoisting the initialization (initializing thread arguments first in a separate loop and then spawning the threads), or, where possible, by padding the thread argument data structures to 4K. In one case, for the `kmeans` benchmark, the benchmark erroneously reuses the same thread arguments for each thread, which not only causes Grace to serialize the program but also is a race condition. We fixed the code by creating a new heap-allocated structure to hold the arguments for each thread.

Page-size base case: We made a one-line change to the `matmul` benchmark, where we increased the base matrix size of the recursion to a multiple of the size of a page to prevent false sharing. Interestingly, this modification was beneficial not only for Grace but also for the `pthread` version. It not only reduces false sharing across the threads but also improves the baseline performance of the benchmark by around 8% by improving its cache utilization.

Changed concurrency structure: Our most substantial change (16 lines of code) was to `pca`, where we changed the way that the program manages concurrency. The original benchmark divided work dynamically across a number of threads, with each thread updating a global variable to indicate which row of a matrix to process next: with Grace, the first thread performed all of the computations. To enable `pca` to scale, we statically partitioned the work by providing each thread with a range of rows. This modification had little impact on the `pthreads` version but dramatically improved the scalability with Grace.

Summary: The vast majority of the code changes were local, purely mechanical and required minimal programmer intervention, primarily in the thread creation loop. In almost every case, the modifications required no knowledge of the underlying application. The reordering or modification involved a small number of lines of code (1–16).

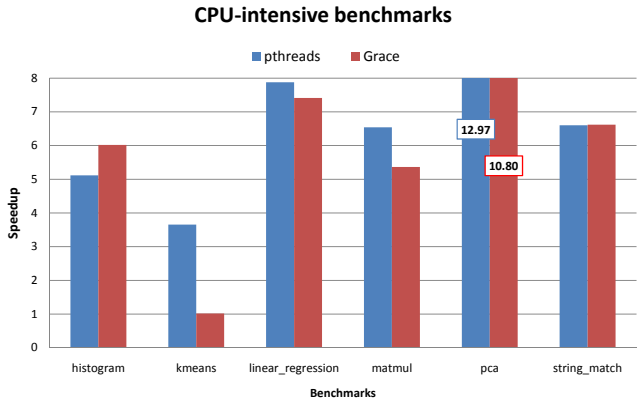

Figure 9. Performance of multithreaded benchmarks running with `pthreads` and Grace on an 8 core system (higher is better). Grace generally performs nearly as well as the `pthreads` version while ensuring the absence of concurrency errors.

6. Evaluation

Our evaluation answers the following questions:

1. How well does Grace perform on real applications?
2. What kind of applications work best with Grace?
3. How effective is Grace against a range of concurrency errors?

6.1 Real Applications

Figure 9 shows the result of running our benchmark suite of applications, graphed as their speedup over a serial execution. The Grace-based versions achieve comparable performance while at the same time guaranteeing 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.

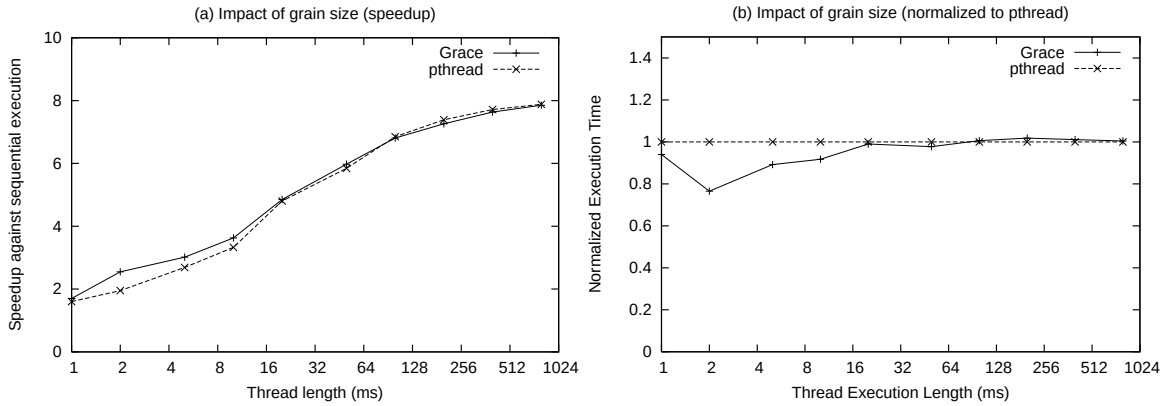

Figure 10. Impact of thread running time on performance: (a) speedup over a sequential version (higher is better), (b) normalized execution time with respect to `pthreads` (lower is better).

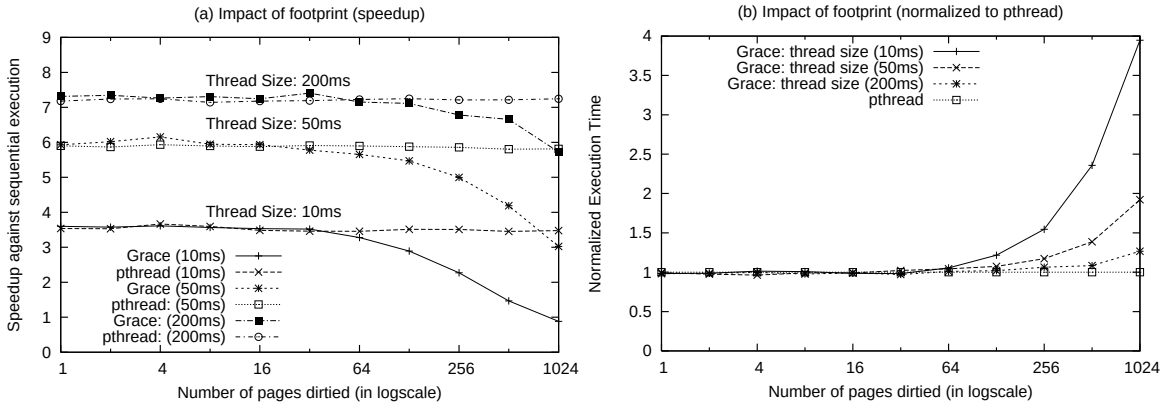

Figure 11. Impact of thread running time on performance: (a) speedup over a sequential version (higher is better), (b) normalized execution time with respect to `pthreads` (lower is better).

While the `kmeans` benchmark achieves a modest speedup with `pthreads` (3.65X), it exhibits no speedup with Grace (1.02X), which serializes execution. This benchmark iteratively clusters points in 3D space. Until it makes no further modifications, `kmeans` spawns threads to find clusters (setting a cluster id for each point), and then spawns threads to compute and store mean values in a shared array. It would be straightforward to eliminate all rollbacks for the first threads by simply rounding up the number of points assigned to each thread, allowing each thread to work on independent regions of memory. However, `kmeans` does not protect accesses or updates to the mean value array and instead uses *benign races* as a performance optimization. Grace has no way of knowing that these races are benign and serializes its execution to prevent the races.

6.2 Application Characteristics

While the preceding evaluation shows that Grace performs well on a range of benchmarks, we also developed a microbenchmark to explore a broader space of applications. In particular, our microbenchmark allows us to vary the following parameters: *grain size*, the running time of each thread; *footprint*, the number of pages updated by a thread; and *conflict rate*, the likelihood of conflicting updates by a thread.

These parameters isolate Grace's overheads. First, the shorter a thread's execution (the smaller its grain), the more the increased cost of thread spawns in Grace (actually process creation) should dominate. Second, increasing the number of pages accessed by a thread (its footprint) stresses the cost of Grace's page protection and signal handling. Third, increasing the number of conflicting updates forces Grace to rollback and re-execute code more often, degrading performance.

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 10 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 role 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 `pthreads` scale well. Figure 10(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 is the worst case scenario for Grace, since each write triggers two page faults.

Figure 11 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, even when each thread dirties one megabyte of memory (256 pages), Grace's performance is 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.

Conflict rate: We next measure the impact of conflicting updates on Grace's performance by having each thread in the microbenchmark update a global variable with a given probability, which the result that any other thread reading or writing that variable will need to rollback and re-execute. Grace makes progress even with a 100% likelihood of conflicts because its sequential semantics provide a progress guarantee: the first thread in commit order is guaranteed to succeed without rolling back Figure 12 shows the resulting impact on speedup (where each thread runs for 50 milliseconds).

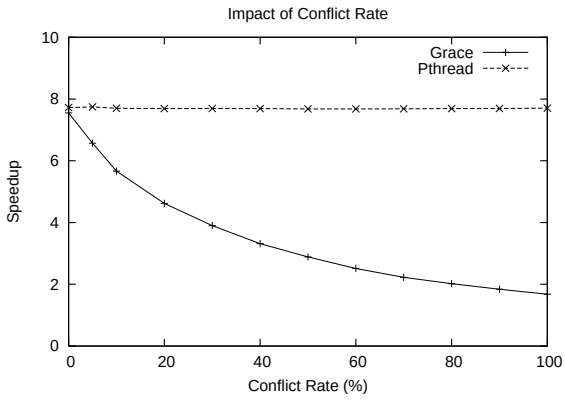

Figure 12. Impact of conflict rate (the likelihood of conflicting updates, which force rollbacks), versus a `pthreads` baseline that never rolls back (higher is better).

When the conflict rate is low, Grace's performance remains close to that of `pthreads`. Higher conflict rates degrade Grace's performance, though to a diminishing extent: a 5% conflict rate leads to a 6-way speedup, while a 100% conflict rate matches the performance of a serial execution. In this benchmark, one processor is always performing useful work, so performance matches the serial baseline. In a program with many more threads than processors, however, a 100% conflict rate under Grace would result in a slowdown.

Summary: This microbenchmark demonstrates that the use of processes versus threads in Grace has little impact on performance for threads that run as little as 10ms, adding no more than 2% overhead and actually providing slightly *better* scalability than `pthreads` in some cases. Memory protection overhead is minimal when the number of pages *dirtied* is not excessively large compared to the grain size (e.g., up to 2MB for 50ms threads). Rollbacks triggered by conflicting memory updates have the largest impact on performance. While Grace can provide scalability for high conflict rates, the conflict rate should be kept relatively low to ensure reasonable performance relative to `pthreads`.

6.3 Concurrency Errors

We illustrate Grace's ability to eliminate most concurrency bugs by compiling a bug suite primarily drawn from actual bugs described in previous work on error detection and listed in Table 3 [25, 26, 27]. 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`.

Bug type	Benchmark description
deadlock	Cyclic lock acquisition
race condition	Race condition example, Lucia et al. [27]
atomicity violation	Atomicity violation from MySQL [26]
order violations	Order violation from Mozilla 0.8 [25]

Table 3. Error benchmark suite.

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

6.3.1 Deadlocks

Figure 13 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.

6.3.2 Race conditions

We next adapt an example from Lucia et al. [27], removing the lock in the original example to trigger a race. Figure 14 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 exhibits this race, printing `0,0,1,1`. By contrast, Grace prevents the race by executing each thread deterministically, and invariably outputs the sequence `0,1,1,2`.

```
// Race condition.
int counter = 0;

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

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

Figure 14. Race condition example: the race is on the variable `counter`, where the first update can be lost. Under Grace, both increments always succeed.

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

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

Figure 15. An atomicity violation from MySQL [26]. 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.

6.3.3 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. [26]. In this example, shown in Figure 15, 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 `S1` and `S2` causes every execution of this code with `pthreads` to segfault because of a NULL dereference. With Grace, threads appear to execute atomically, so the program always performs correctly.

6.3.4 Order violations

Finally, we consider order violations, which were recently identified as a common class of concurrency errors by Lu et

```
// Order violation.
char * proc_info;

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

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

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

Figure 16. An order violation. If thread 2 executes before thread 1, it writes into unallocated memory. Grace ensures that thread 2 always executes after thread 1, avoiding this error.

al. [26]. 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 16 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 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 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 17 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` (once `thread1` executes) 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.

```
// Order violation.
int foo;

thread1() {
 foo = 0;
}

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

Figure 17. An order violation. Here, the intended effect violates sequential semantics, so the error is not fixed but occurs reliably.

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.

7. Related Work

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

7.1 Transactional memory

The area of transactional memory, first proposed by Herlihy and Moss for hardware [17] and for software by Shavit and Touitou [36], is now a highly active area of research. Larus and Rajwar's book provides an overview of recent work in the area [23]. We limit our discussion here to the most closely related software approaches that run on commodity hardware.

Transactional memory eliminates deadlocks but does not address other concurrency errors like races and atomicity, leaving the burden on the programmer to get the atomic sections right. Worse, software-based transactional memory systems (STM) typically interact poorly with irrevocable operations like I/O and generally degrade performance when compared to their lock-based counterparts, especially those that provide strong atomicity [6]. STMs based on weak atomicity can provide reasonable performance but expose programmers to a range of new and subtle errors [37].

Fraser and Harris's transaction-based *atomic blocks* [15] 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. [37] 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.

Welc et al. introduce support for irrevocable transactions in the McRT-STM system for Java [40]. Like Grace, their system supports one active irrevocable transaction at a time. McRT-STM relies on a lock mechanism combined with compiler-introduced read and write barriers, while Grace's support for I/O falls out "for free" from its commit protocol. The McRT system for C++ also includes a malloc implementation called McRT-malloc, which resembles Hoard [3] but is extended to support transactions [19]. Ni et al. present the design and implementation of a transactional extension to C++ that enable transactional use of the system memory allocator by wrapping all memory management functions and providing custom commit and undo actions [31]. These approaches differ substantially from Grace's memory allocator, which employs a far simpler design that leverages the fact that in Grace, all code, including `malloc` and `free`, execute transactionally. Grace also takes several additional steps that reduce the risk of false sharing.

7.2 Concurrent programming models

We restrict our discussion of programming models here to imperative rather than functional programming languages. Cilk [5] is a multithreaded extension of the C programming language. Like Grace, Cilk uses a fork-join model of parallelism and 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 currently restricted to C. Cilk also relies on programmers to avoid race conditions and other concurrency errors; while there has been work on dynamic tools to locate these errors [8], 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 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 [18]. Concurrency errors remain common even in fork-join programs: Feng and Leiserson report that their Nondeterminator race detector for Cilk found races in several Cilk programs written by experts, as well as in half the submitted implementations of Strassen's matrix-matrix multiply in a class at MIT [13].

Intel's Threading Building Blocks (TBB) is a C++ library that provides lightweight threads ("tasks") executing on a Cilk-like runtime system [35]. TBB comprises a non-POSIX compatible API, primarily building on a fork-join programming model with concurrent containers and high-level loop constructs like `parallel_do` that abstract away details like task creation and barrier synchronization (although TBB also includes support for pipeline-based parallelism, which Grace does not). TBB relies on the programmer to avoid concurrency errors that Grace prevents.

Automatic mutual exclusion, or AME, is a recently-proposed programming model developed at Microsoft Research Cambridge. It is a language extension to C# that assumes that all shared state is private unless otherwise indicated [20]. 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 its substantially more complex semantics preclude a sequential interpretation [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 [38]. 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.

Welc et al. present a future-based model for Java programming that, like Grace, is "safe" [39]. 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 currently requires integration with a garbage-collected environment, making it generally unsuitable for use with C/C++.

Grace's use of virtual memory primitives to support speculation is a superset of the approach used by behavior-oriented parallelism (BOP) [12]. 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 concurrently-executing, fork-join based multithreaded programs.

7.3 Deterministic thread execution

A number of runtime systems have recently appeared that are designed to provide a measure of deterministic execution of multithreaded programs. Isolator uses a combination of programmer annotation, custom memory allocation, and virtual memory primitives to ensure that programs follow a locking discipline [33]. Isolator works on existing lock-based codes, but does not address issues like atomicity or

deadlock. Kendo also works on stock hardware and provides deterministic execution, but only of the order of lock acquisitions [32]. It also requires data-race free programs. DMP uses hardware support to provide a total ordering on multithreaded execution, which aims to ensure that programs reliably exhibit the same errors, rather than attempting to eliminate concurrency errors altogether [10].

In concurrent work, Bocchino et al. present Deterministic Parallel Java (DPJ), a dialect of Java that adds two parallel constructs (`cobegin` and `foreach`) [21]. A programmer using DPJ provides region annotations to describe accesses to disjoint regions of the heap. DPJ's type and effect system then verifies the soundness of these annotations at compile-time, allowing it to execute non-interfering code in parallel with the guarantee that the parallel code executes with the same semantics as a sequential execution (although it relies on the correctness of commutativity annotations). Unlike Grace, DPJ does not rely on runtime support, but requires programmer-supplied annotations and cannot provide correctness guarantees for ordinary multithreaded code outside the parallel constructs.

7.4 Other uses of virtual memory

A number of distributed shared memory (DSM) systems of the early 90's also employed virtual memory primitives to detect reads and writes and implement weaker consistency models designed to improve DSM performance, including Munin [7] and TreadMarks [22]. While both Grace and these DSM systems rely on these mechanisms to trap reads and writes, the similarities end there. Grace executes multithreaded shared memory programs on shared memory systems, rather than creating the illusion of shared memory on a distributed system, where the overheads of memory protection and page fault handling are negligible compared to the costs of network transmission of shared data.

8. Future Work

In this section, we outline several directions for future work for Grace, including extending its range of applicability and further improving performance.

We intend to extend Grace to support other models of concurrency beyond fork/join parallelism. One potential class of applications is request/response servers, where a single controller thread spawns many mostly-indepedent child threads. For these programs, Grace could guarantee isolation for child threads while maintaining scalability. This approach would require modifying Grace's semantics to allow the controller thread to spawn new children without committing in order to allow it to handle the side-effects of socket communication without serializing spawns of child threads.

While conflicts cause rollbacks, they also provide potentially useful information that can be fed back into the runtime system. We are building enhancements to Grace that will both report memory areas that are the source of frequent conflicts and act on this information. This information can guide programmers as they tune their programs for higher performance. More importantly, we are currently developing a tool that will allow this data to be used by Grace to automatically prevent conflicts (without programmer intervention) by padding or segregating conflicting heap objects from different call sites.

While we have shown that process invocation is surprisingly efficient, we would like to further reduce the cost of threads. While we do not evaluate it here, we recently developed a technique that greatly lowers the cost of thread invocation by taking advantage of the following key insight. Once a divide-and-conquer application has spawned a large enough number of threads to take advantage of available processors, it is possible to practically eliminate the cost of thread invocation at deeper nesting levels by directly executing thread functions instead of spawning new processes. While this approach has no impact on our benchmark suite, it dramatically decreases the cost of thread spawns, running at under 2X the cost of Cilk's lightweight threads.

Another possible use of rollback information would be for scheduling: 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.

We are also investigating the use of compiler optimizations to automatically transform code to increase scalability. For example, Grace's sequential semantics could enable *cross-thread* optimizations, such as hoisting conflicting memory operations out of multiple threads.

9. Conclusion

This paper presents Grace, a runtime system for fork-join based C/C++ programs that, by replacing the standard threads library with a system that ensures deterministic execution, eliminates a broad class of concurrency errors, including deadlocks, race conditions, atomicity violations, and order violations. With modest source code modifications (1–16 lines of code in our benchmark suite), Grace generally achieves good speed and scalability on multicore systems while providing safety guarantees. The fact that Grace makes multithreaded program executions deterministic and repeatable also has the potential to greatly simplify testing and debugging of concurrent programs, even where deploying Grace might not be feasible.

10. Acknowledgements

The authors would like to thank Ben Zorn for his feedback during the development of the ideas that led to Grace, to Luis Ceze for graciously providing benchmarks, and to Cliff Click, Dave Dice, Sam Guyer, and Doug Lea for their invaluable comments on earlier drafts of this paper. We also thank Divya Krishnan for her assistance. 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] D. F. Bacon and S. C. Goldstein. Hardware-assisted replay of multiprocessor programs. In *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 194–206, New York, NY, USA, 1991. 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] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *SOSP '91: Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, New York, NY, USA, 1991. ACM.

[8] 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.

[9] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[10] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *ASPLOS '09: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, New York, NY, USA, 2009. ACM.

[11] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In S. Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2006.

[12] 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.

[13] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in cilk programs. In *SPAA '97: Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, pages 1–11, New York, NY, USA, 1997. ACM.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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, New York, NY, USA, 2006. ACM.

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

[21] R. L. B. Jr., V. S. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *OOPSLA '09: Proceedings of the 24th ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, New York, NY, USA, 2009. ACM.

[22] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference*, pages 10–10, Berkeley, CA, USA, 1994. USENIX Association.

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

[24] D. Lea. A Java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, New York, NY, USA, 2000. ACM.

[25] 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.

[26] 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.

[27] 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.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[29] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, 1989.

[30] 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.

[31] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. 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, New York, NY, USA, 2008. ACM.

[32] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *ASPLOS '09: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 97–108, New York, NY, USA, 2009. ACM.

[33] S. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani. ISOLATOR: dynamically ensuring isolation in comcurrent programs. In *ASPLOS '09: Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 181–192, New York, NY, USA, 2009. ACM.

[34] 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.

[35] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA '08: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, pages 285–296, New York, NY, USA, 2008. ACM.