Efficient Deterministic Multithreading Without Global Barriers

Kai Lu[1,2] Xu Zhou[1,2] Tom Bergan[3] Xiaoping Wang[1,2]

1.Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, Changsha, PR China
2.College of Computer, National University of Defense Technology, Changsha, PR China
3.University of Washington, Computer Science and Engineering
{kailu, zhouxu, xiaopingwang}@nudt.edu.cn, tbergan@cs.washington.edu

Abstract

Multithreaded programs execute nondeterministically on conventional architectures and operating systems. This complicates many tasks, including debugging and testing. *Deterministic multithreading* (DMT) makes the output of a multithreaded program depend on its inputs only, which can totally solve the above problem. However, current DMT implementations suffer from a common inefficiency: they use frequent global barriers to enforce a deterministic ordering on memory accesses. In this paper, we eliminate that inefficiency using an execution model we call *deterministic lazy release consistency* (DLRC). Our execution model uses the Kendo algorithm to enforce a deterministic ordering on synchronization, and it uses a deterministic version of the lazy release consistency memory model to propagate memory updates across threads. Our approach guarantees that programs execute deterministically even when they contain data races. We implemented a DMT system based on these ideas (RFDet) and evaluated it using 17 parallel applications. Our implementation targets C/C++ programs that use POSIX threads. Results show that RFDet gains nearly 2x speedup compared with DThreads—a start-of-the-art DMT system.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; D.3.4 [*Programming Languages*]: Processors—Runtime environments

Keywords deterministic execution, multithreading, lazy release consistency

PPoPP '14, February 15–19, 2014, Orlando, Florida, USA.

http://dx.doi.org/10.1145/2555243.2555252

1. Introduction

Multithreaded programs execute nondeterministically on conventional systems: a program may produce different outputs in different executions even when provided with exactly the same input. This complicates development in many aspects: debugging is difficult because bugs may disappear on subsequent executions, and testing, fault-tolerant replication, and intrusion analysis become more difficult as well [3, 17, 26, 30]. Deterministic multithreading (DMT) has been recently proposed as a solution. DMT systems constrain execution so that multithreaded programs always execute the same thread interleavings and produce the same output when provided with the same input. Due to its many applications, DMT has become an increasingly attractive goal [3, 4, 7, 9, 11, 17, 18, 30, 37].

Prior general-purpose DMT systems take one of two basic approaches. First, systems like Kendo [30] enforce a deterministic order on synchronization *only*. In Kendo, the basic idea is that a thread cannot perform synchronization until all other threads have executed more instructions. This approach, known as *weak determinism*, can be implemented very efficiently in software, but it provides few guarantees for programs with data races—such programs may execute nondeterministically. The second approach is to enforce a deterministic order on *all* memory operations. This approach, known as *strong determinism*, typically proceeds by executing threads in bulk-synchronous quanta [4, 7, 17, 18, 21, 28, 37]. Within each quantum, threads are isolated. Each quantum ends with a *global barrier*, followed by a short phase in which threads communicate memory updates in a deterministic fashion. Strong DMT systems are attractive because they guarantee determinism even in the presence of data races. However, strong DMT systems are not yet practical: hardware-supported approaches cannot run on commodity architectures, while software-only implementations suffer from prohibitive overhead [6].

The strong DMT systems proposed previously all suffer from a common source of overhead: global barriers. These barriers provide a convenient place to make deterministic decisions, but they force all threads to synchronize even

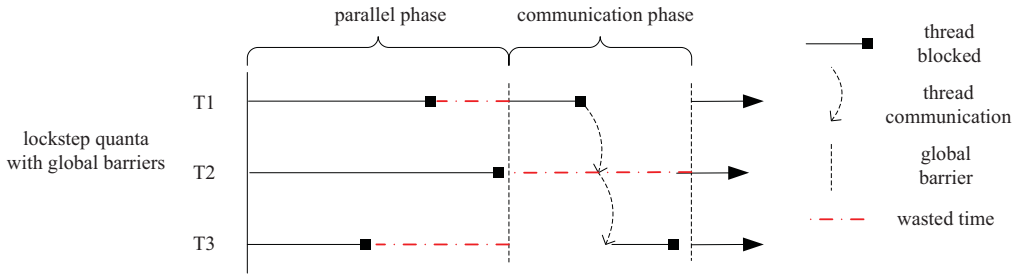

Figure 1. Global barriers in deterministic multithreading.

when such synchronization is unnecessary. This paper proposes a way to provide strong determinism without introducing *any* global barriers. Our insight is to combine the Kendo algorithm for deterministic synchronization with the *lazy release consistency* (LRC) memory model, which was first described in the context of distributed shared memory [24]. Hence, we call our approach *deterministic lazy release consistency* (DLRC). At a high-level, DLRC works in two parts. First, we use the Kendo algorithm [30] to ensure that synchronization operations happen in a deterministic order. Second, we give each thread a private memory space and enforce the following rule: a memory modification performed by thread $T1$ is made visible in thread $T2$ *if and only if* the modification *happens before* $T2$'s currently executing instruction. DLRC guarantees strong determinism but does not require global barriers.

We have implemented DLRC in a software runtime system called RFDet. Our implementation targets C/C++ programs that use POSIX threads (`pthreads`). We use page protection to give each thread a private memory space so that local modifications will not be immediately visible to other threads. We provide our own implementation of the `pthreads` library that is responsible for ensuring a deterministic order of synchronization (using the Kendo algorithm) and for propagating memory updates to other threads (following DLRC). A key challenge is propagating memory updates efficiently. We partition each thread's execution into synchronization-free *slices*, and for each slice we use page-diffing to compute the set of updates performed within that slice. We use copy-on-write techniques to minimize the number of pages that must be diffed. We evaluated RFDet on a range of parallel applications, including the deterministic stress test *racey* and 16 other programs from the SPLASH-2, Phoenix, and Parsec suites. Our evaluation shows that RFDet is deterministic and improves performance over DThreads [28]—a state of the art DMT system—by nearly 2x.

The rest of this paper is organized as follows. Section 2 provides background in DMT systems and summarizes related work. Section 3 describes deterministic lazy release consistency in detail. Section 4 describes our implementation of the RFDet runtime system. Section 5 describes our evaluation, and Section 6 concludes.

2. Background and Related Work

Strong Determinism with Lockstep Quanta. Strong determinism is a style of DMT that ensures deterministic results even in the presence of data races. There have been many recent attempts to provide efficient strong determinism. We leave a detailed discussion of that prior work to the survey by Bergan et al. [6] and to our bibliography [4, 7, 9, 17, 18, 21].

For this paper, we observe that all prior strong DMT systems use the same basic formula that is illustrated in Figure 1: execution is partitioned into quanta, where each quantum includes a parallel phase in which threads are isolated, followed by a short communication phase in which communication is resolved deterministically. A parallel phase ends after each thread has performed a deterministic amount of work, where "work" is usually measured by counting instructions, and phases are separated by global barriers. Note that DThreads [28] adopts a different mechanism: in DThreads, a parallel phase ends after each thread encounters any system-provided synchronization operation.

Global barriers introduce two sources of overhead. First, they introduce unnecessary serialization. Suppose threads $T1$ and $T3$ need to communicate, perhaps by writing to a shared queue. Following the formula in Figure 1, $T1$ and $T3$ must wait for the communication phase before their writes can proceed. Unfortunately, $T2$ must stop at the quantum barriers as well, *even though it has no need to communicate*—this serialization is unnecessary. Second, although all threads perform a deterministic amount of work per quantum, they might perform uneven amounts of work, leading to imbalance. This potential for imbalance is illustrated in Figure 1, and it has been shown to be a real performance issue that requires careful tuning [4, 17].

Prior authors have observed that some performance can be recovered by exploiting relaxed memory models. For example, the first system to provide strong determinism used sequentially consistent memory models [17]. Subsequent systems used a relaxed memory model derived from total-store-order (TSO) [4, 21], and most recently, RCDC proposed a new memory model called DMP-HB [18]. We continue this trend by introducing deterministic lazy release consistency (DLRC). DLRC is most similar to DMP-HB, but is more relaxed, as DLRC does *not* require global barri-

ers. As we will argue in Section 3, DLRC relaxes memory consistency and improves the efficiency of strong determinism *without* breaking the semantics of the original program.

Prior systems use a variety of implementation strategies, including compiler instrumentation [4], page protection tricks [3, 7, 9, 28, 29], and even custom hardware [17, 18, 21]. Our system, RFDet, uses implementation techniques that are most similar to DThreads [28], so our evaluation will use DThreads as a comparison point.

Weak Determinism. Kendo [30] was the first system to provide determinism for race-free programs by serializing all synchronization operations in a deterministic order. The basic idea is to let each thread run until it reaches a synchronization operation, at which point the thread must wait until all other threads have executed more instructions. We refer to the Kendo paper for details [30].

Note that this algorithm does not use global barriers: threads do not block until they attempt to perform synchronization, and even then they are allowed to proceed immediately if they have the lowest instruction count. Our DLRC memory model makes use of the Kendo algorithm as explained in Section 3.

Parrot [14] also provides weak determinism, but rather than serializing synchronization via instruction counting as in Kendo, Parrot schedules threads in a deterministic round-robin order and uses programmer annotations to guide the scheduler towards efficient schedules. However, even with programmer annotations, Parrot cannot always find an efficient deterministic schedule and must occasionally resort to nondeterminism. In contrast, Kendo's (and DLRC's) use of instruction counting provides efficient and deterministic schedules without requiring programmer annotations.

Strong vs. Weak Determinism. It is useful to compare the guarantees of strong and weak determinism. Both ensure determinism for race-free programs, but their guarantees differ in the presence of data races. *Weak* systems such as Kendo do not resolve data races deterministically, and thus, they provide determinism *up to the first data race, only*. In contrast, *strong* systems such as DMP resolve *all* data races in a deterministic way, and thus provide determinism for entire executions.

Kendo can help debug the first race encountered on a given execution. This is useful, as all data races should be considered bugs [12]. However, in practice, not all data races are equally harmful—some races lead to severe crashes, while others go relatively unnoticed [22]—and developers need to prioritize their debugging effort towards those severe bugs. Hence, we consider it vital to resolve *all* races deterministically to ensure that the most severe races are reproducible, and thus, debuggable.

Schedule Memoization. Tern [15] and Peregrine [16] memorize schedules encountered during testing and reuse those schedules during deployment when possible. This provides high reliability guarantees in cases where tested schedules can be reused. However, it is not possible to reuse tested schedules in all cases, so these systems must occasionally resort to nondeterministic execution. Hence, they provide *best-effort* determinism only.

A recent system by Bergan et al. [5] attempts to extend the approach introduced by Tern and Peregrine to use *input-covering schedules*. The idea is to compute a set of schedules S that is sufficiently large so that program execution can follow at least one schedule in S when given any input. However, this approach requires an expensive symbolic execution that has not been shown to scale to large systems.

Distributed Shared Memory (DSM) systems provide a logically shared memory space for distributed systems [23, 24] that does not share physical memory between nodes. In RFDet, we use similar techniques to implement memory modification propagation. There are two major differences: 1) RFDet ensures determinism while DSM systems do not (see Section 3); and 2) RFDet manages threads with physically shared memory, while DSM operates on distributed machines that do not share a physical address space.

Record and Replay systems (R+R) can deterministically replay a multithreaded execution that was recorded previously [25, 27, 32, 33]. These systems record a trace of thread interleavings in addition to program inputs. DMT systems like RFDet have two advantages over R+R systems. First, DMT systems guarantee that there is *one* possible execution for each input, so they can achieve deterministic replay by recording program inputs *only*—this can result in significantly lower recording overheads compared to R+R systems, which must record thread interleavings as well [7].

Second, while R+R systems can replay a specific execution for a given input, DMT systems ensure that *all* executions behave the same way for that given input. This allows DMT systems to provide benefits for program testing (by ensuring that a program behaves the same way in production as during testing) and for fault-tolerant state-machine replication (by ensuring that all state machine replicas make the same sequence of state changes when given the same sequence of inputs) [6, 15].

3. Deterministic Lazy Release Consistency

In deterministic lazy release consistency (DLRC), we divide program operations into two categories: synchronization operations (such as `pthread_mutex_lock`) and ordinary memory accesses (reads and writes). First, we use the Kendo algorithm to ensure that synchronization operations happen in a deterministic total order. Second, we give each thread a private memory space and enforce the following rule: a memory modification performed by thread $T1$ is made visible in thread $T2$ *if and only if* the modification *happens before* $T2$'s currently executing instruction. This has two implications: (1) if a modification happens before the currently executing instruction, the modification should be visible; and

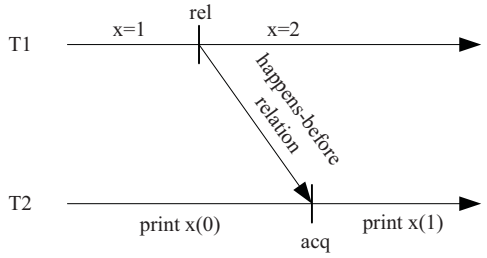

Figure 2. Deterministic lazy release consistency.

(2) any modification that does not happen before the currently executing instruction *must not* be visible.

More formally, the definition of DLRC depends on the happens-before relation. We use $\rightarrow$ to denote the happens-before relation, so $A \rightarrow B$ means operation A happens before operation B. The happens-before relation is the irreflexive transitive closure of program order and synchronization order, where $A \rightarrow B$ in program order if operations A and B are performed by the same thread and A appears in the program before B,[1] and where $A \rightarrow B$ in synchronization order if A and B are synchronization operations on the same object (e.g., the same lock) and A completes before B.

Now suppose that R and W are read and write operations, respectively, where R and W access the same location and may be performed by different threads. In DLRC, R reads the value written by W *only if* $W \rightarrow R$ and there does not exist another write $W2$, to the same location, such that $W \rightarrow W2$ and $W2 \rightarrow R$. If there exists a third write to the same location, $W3$, such that $W \rightarrow R$ and $W3 \rightarrow R$ but there is no happens-before relation between W and $W3$, then R may read the value written by *either* W or $W3$, as long as that choice is made deterministically. One strategy is to use thread creation order as a tie breaker.

As shown in Figure 2, the first `print` in $T2$ must not see the two modifications of $T1$ as there is no happens-before relation between them. Meanwhile, the second `print` must see the modification of `x=1` but must not see the modification of `x=2` because only `x=1` happens before that `print` due to a previous synchronization between $T1$ and $T2$. Note that in each case there is a data race that DLRC has resolved in a deterministic fashion.

DLRC differs from LRC [23, 24] in two respects. First, synchronization happens deterministically in DLRC due to Kendo, but synchronization order is unspecified (nondeterministic) in LRC. Second, although LRC guarantees that a write W is visible to a read R if $W \rightarrow R$, as in DLRC, it may also allow a write W to be visible to R *even when* $W \not\rightarrow R$. In contrast, DLRC guarantees that W *must not* be visible to R when $W \not\rightarrow R$. For the example in Figure 2, `x=2` *may* be visible to the second `print` in $T2$ in LRC, while this visibility *must* be disabled in DLRC. This limitation helps us to guarantee determinism but also makes the memory modification propagation procedure more complicated than that of LRC (see Section 4).

[1] In the C++, this is the *sequenced-before* relation [1, 13].

3.1 Advantages Over Prior Approaches

We have already said that DLRC improves prior approaches to determinism by eliminating global barriers. The following example illustrates our argument further. Suppose that in some program fragment, threads $T1$ and $T3$ attempt to acquire the same lock while thread $T2$ does not perform any synchronization. With DLRC, $T2$ executes in isolation and does not block. The only delays are, first, a small delay while $T1$ and $T3$ use Kendo to deterministically arbitrate the order of lock acquisition, and second, the unavoidable delay in which one thread waits for the other to release the lock.

In contrast, prior systems insert extra delays due to global barriers. In systems such as DMP [17], CoreDet [4], and Calvin [21], execution may proceed as shown in Figure 1. In these systems, synchronization cannot occur in the parallel phase, so $T1$ and $T3$ must wait for $T2$ to arrive at the global quantum barrier before they can synchronize. Due to imbalance, this delay can be significant [4, 6]. Further, even though $T2$ is not synchronizing, $T2$ must still wait for $T1$ and $T3$ to synchronize in the communication phase before it can continue execution. RCDC [18] improves this somewhat by allowing at most one thread to acquire a given lock in the parallel phase without waiting for the global barrier. However, two threads cannot acquire the same lock without a global barrier, as in our current example. In DThreads, the problem is potentially even worse as neither $T1$ or $T3$ can acquire the lock until $T2$ reaches some synchronization operation, which may be far in the future.

Even if no thread performs synchronization, many systems still require global barriers, leading to the potential for imbalance and wasted delays [4, 17, 18, 21]. DLRC requires no global barriers, and, as we argue in Section 5, this leads to improved performance.

3.2 Determinism

We demonstrate determinism with an informal argument by induction over an execution trace. In the base case, all threads execute for some time without performing synchronization. This is trivially deterministic since each thread is constrained to a private memory space. In the inductive case, some thread executes a synchronization operation. Our use of Kendo ensures that synchronization operations are ordered deterministically. Hence, the happens-before relation is updated deterministically, and from this fact and the inductive hypothesis, it follows that memory updates are propagated deterministically. Another way to argue determinism is the following: DLRC defines memory modification propagation as a deterministic function over the happens-before relation, and since our use of Kendo produces a deterministic happens-before relation, it follows that execution as a whole is deterministic.

3.3 Correctness

As our implementation is targeted to C and C++, we must show that source programs written in C and C++ can legally execute under DLRC. Here, we argue that DLRC correctly executes C++ programs that do not use low-level atomics—we will return to low-level atomics in Section 4.6. The C++ memory model requires that all program executions adhere to the following rule (see Section 6 of Boehm and Adve [1, 13]): If the program has a data race, its behavior is undefined; otherwise, the program's execution must be *consistent*. Boehm and Adve give a five-part definition of *consistent* in Section 6 of their paper. Below, we summarize the three parts of that definition that do not refer to low-level atomics, and we argue that, for race-free programs, DLRC preserves semantics:

1. *Execution respects single-threaded semantics.* This trivially holds as single-threaded semantics are not affected by DLRC.
3. *Each memory read R reads from a write W to the same location such that $W \rightarrow R$ and there does not exist another write $W2$ such that $W \rightarrow W2$ and $W2 \rightarrow R$.* DLRC follows this rule exactly (see above). Note that if there exists a third write, $W3$, where $W3 \rightarrow R$ and $W3$ is concurrent with W, then there is a data race and the C++ semantics allow R to read any arbitrary value.
5. *Lock and unlock operations on each individual lock are totally ordered by happens-before.* This property holds because, first, DLRC uses Kendo to order synchronization, and second, Kendo orders all synchronization operations in a (deterministic) total order.

Boehm and Adve further argue that the above rules guarantee sequentially consistent execution of race-free programs (see Section 7 of their paper). Thus, DLRC preserves sequential consistency for race-free programs, and we conclude that DLRC does not violate C++ semantics.

3.4 Discussion

Guarantees. DLRC guarantees that execution of a given program with a given input will produce *arbitrary but deterministic and semantically-valid results*, even in the presence of data races. Essentially, DLRC achieves determinism in the presence of data races by sequencing conflicting (racing) accesses in a deterministic order—note that this deterministic order depends on input, meaning that DLRC may resolve the same race in two different orders on two different inputs.

Inputs. We assume a broad definition of the term *input*. Namely, in addition to the usual notions of input such as commandline flags, user actions, and data read from files or the network, our notion of input includes environmental parameters such as psuedorandom seeds, number of processors, and system load. Particularly, if a program is designed to adjust its number of threads according to the current system load, then we consider the current system load an *input*, and as a result, our system may produce different executions in environments with different system loads.

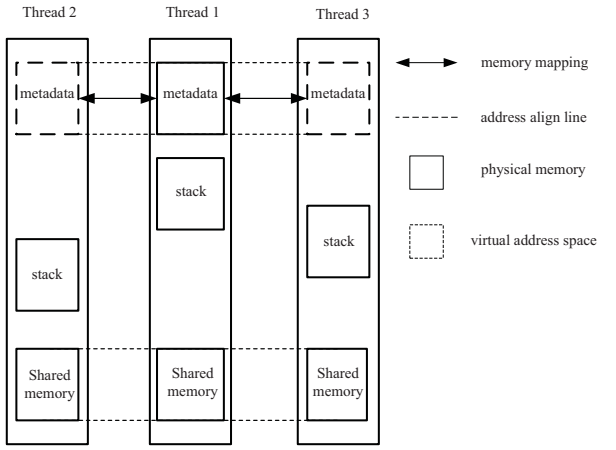

Figure 3. Memory spaces in FPDet. Between different threads, *metadata* spaces share both virtual addresses and physical memory, *shared memory* spaces share virtual addresses only, and *stacks* share neither of them.

Other Languages. We see no reason why DLRC could not be implemented for other languages, such as Java. However, our specific implementation merges conflicting (racing) updates in a way that can violate the Java memory model (see Section 4.6), so, for Java, an alternative implementation is needed.

4. Implementation

This section presents our implementation of a DMT system (RFDet) based on DLRC. The first issue is to make local memory modifications invisible to other threads. To this end, we use processes to replace threads so that the memory spaces are separated, as shown in Figure 3. In Linux, this is implemented by using the `clone` system call so that processes share everything (file descriptors, sockets, etc.) except for a memory address space [28]. As a result, we have to provide our own memory allocator to avoid address conflicts (Section 4.4). To implement inter-process communication, we reserve a *metadata* space which is a shared memory region between these processes.

Based on these *isolated threads*, we implement thread communication according to DLRC in the following way. We provide our own implementation for the standard synchronization operations in POSIX `pthreads` (Section 4.1). We dynamically cut thread execution into slices using these synchronizations. For each slice, we monitor its modifications, record them into the metadata space, and use a vector clock timestamp to describe its happens-before relation with other slices (Section 4.2). In each synchronization operation that induces a happens-before relation with another thread, we propagate the memory modifications of all slices that happen before the synchronization operation to the lo-

cal thread, and merge them into the local memory to make them visible (Section 4.3). We perform four optimizations to reduce the overhead in both time and space (Section 4.5).

4.1 Synchronization

We divide the standard POSIX `pthreads` synchronizations into *acquire* operations and *release* operations. Specifically, the acquire operations include *lock*, *conditional wait*, *thread entry*, *thread join* and *barrier*. The corresponding release operations are *unlock*, *conditional signal/broadcast*, *thread create*, *thread exit* and *barrier* respectively. Note that *barrier* is both an acquire and a release operation.

Deterministic Synchronization. As stated, we use the Kendo algorithm [30] to make synchronization deterministic. Briefly, in Kendo, a thread is allowed to perform synchronization only if it has executed fewer instructions than all other threads. We count instructions using compile-time instrumentation. Specifically, we insert a call to `instrTick(k)` in each basic block, where `k` is the number of memory instructions in that basic block. When `instrTick` is invoked at runtime, we increment the current thread's instruction count.

By contrast, the original Kendo implementation used performance counters to count instructions. We did not adopt this method because the determinism of performance counters is not proven [34]. We otherwise follow the Kendo algorithm as described, and we refer to the Kendo paper for further details [30].

Internal Synchronization Variables. Synchronization variables, such as mutexes, must be updated atomically across all threads. However, we cannot update the application's memory atomically because each thread has an isolated memory space. Our approach is to map each synchronization variable to an *internal synchronization variable* that is allocated in the metadata space. At each synchronization operation, we lookup the corresponding internal variable in the metadata space and operate on that internal variable directly.

Additionally, we add two fields to each internal synchronization variable: *lastTid* and *lastTime*. These represent, respectively, the ID of the last thread to *release* the synchronization variable, and the time at which that *release* occurred. We represent times using vector clocks [19] as explained in Section 4.2.

Mutexes and Condition Variables. At each *acquire* operation, such as `pthread_mutex_lock` or `pthread_cond_wait`, we check the *lastTid* field of the synchronization variable. If the last *release* was performed by a different thread, we propagate all memory modifications that happen before *lastTime* into the current (acquiring) thread as explained in Section 4.3. Otherwise, if the last *release* was performed by the same thread, we merge the current thread's previous slice and new slice for efficiency, as explained in Section 4.5. At each *release* operation, such as `pthread_mutex_unlock` or `pthread_cond_signal/pthread_cond_broadcast`, we set the *lastTid* and *lastTime* fields before we release the synchronization variable.

Barriers. Barriers are special synchronizations as they perform both *acquire* and *release*. At each barrier, we first select the arriving thread with the smallest thread ID (call it thread T), and then merge all modifications that happened-before the barrier into T's local memory. The merging order is determined by thread ID (the thread with the smallest ID merges its modifications first) to ensure determinism. All threads are given a copy of T's local memory (using copy-on-write) after the merging completes.

Thread Create and Join. In `pthread_create`, we use the `clone` system call to implement threads so that each thread is actually a lightweight process. We assign each new thread a deterministic thread ID—calling `pthread_self` will return this ID instead of the ID assigned by the operating system. Note that there is a happens-before relation between thread creation and the child thread's entry point. However, we do not need to propagate memory modifications at this moment as the child process will inherit the memory of its creating process automatically. Further, we do not need to monitor memory modifications in the main thread before the first child thread is created. To implement `pthread_join`, we map the deterministic thread ID to the process ID returned by `clone`, and use `waitpid` to wait for the specified process. Note that we have to propagate all the modifications of the joined thread to the main thread at this moment.

4.2 Slices

A *slice* refers to a period of single-threaded execution between two consecutive synchronizations. In other words, each slice is immediately preceded and succeeded by synchronization and there is no synchronization within the slice itself. Hence, at each synchronization operation, we should end the previous slice and begin a new slice. Slices have a useful *atomic property*: **all memory accesses inside a slice will have the same happens-before relation to any instruction outside the slice**. This property enables us to make slices our basic unit for memory modification propagation.

Each slice is a triple *<tid, modifications, timestamp>*, where *tid* is a thread ID, *modifications* describes the ordered sequence of memory updates made by thread *tid* during the slice, and *timestamp* is a vector clock timestamp for the slice.

Vector Clocks. We can easily know the happens-before relation between any two slices by comparing their vector clock timestamps. Namely, given two slices A and B, $A \rightarrow B$ if and only if $Time(A) < Time(B)$ [19]. We maintain a vector clock for each thread and increase it in the standard way: 1) before each synchronization operation, we increase the vector clock so that the next slice is older than the previous slice; and 2) at each *acquire* that synchronizes with a *release* in a different thread, we update the vector clock to *timestamp* $\sqcup\, Time(R)$, where *timestamp* is the vector clock

```
1  void RecordStore(void *addr, size_t len) {
2      foreach pageid in pagesTouchedBy(addr, len) {
3          if (isInSharedMemory(pageid) &&
4              !currentSlice.hasPageSnapshot(pageid)) {
5              void *pagedata = metadata->allocOnePage();
6              memcpy(pagedata, PageAddr(pageid), PAGE_SIZE);
7              curentSlice.addPageSnapshot(pageid, pagedata);
8          }
9      }
10 }
```

Figure 4. Algorithm for Store instrumentation.

```
1  void DoMemoryModificationPropagation(thread from,
2                                       vtime upperlimit,
3                                       vtime lowerlimit){
4      foreach slice in from.slicepointers {
5          if(slice.time < upperlimit &&
6                  ! (slice.time < lowerlimit) ) {
7              copyToLocalMemory(slice.modifications);
8              localthread.slicepointers->append(slice);
9          }
10     }
11 }
```

Figure 5. Algorithm for memory modification propagation.

just before the *acquire*, $Time(R)$ is the vector clock of the release, and $\sqcup$ is a least-upper-bound.

Monitoring Memory Modifications. We represent *modifications* using a list of pairs <*addr, data*>, where each pair represents a write of the value *data* to address *addr*. The granularity of *data* is one byte. We need to collect the modifications performed during each slice. As in DThreads [28], our approach is to use page diffing: the first time a page is written in a slice, we take a snapshot of the page and add it to a *modified pages list*. Then, at the end of the slice, we compare the snapshot pages with the corresponding modified pages byte-by-byte to compute the modifications.

We collect modified pages by instrumenting all *Store* instructions at compile time. We instrument each *Store* instruction as shown in Figure 4. Specifically, we check if the page written by the *Store* is in shared memory (recall Figure 3) and if this is the first time the page has been written in the current slice. If so, we take snapshot of the page and add it to the *modified pages list* of the current slice.

We assume that stack variables are not shared across threads. At compile time, we use a conservative static escape analysis to filter out *Stores* to stack variables, similarly to prior work [4]. At runtime, we ignore stores to stack pages (line 3 of Figure 4). Further, our compiler instrumentation assumes the entire source code is available, including for libraries. If the source is not available for library L, then we assume that either (a) library L does not write to shared memory locations, in which case instrumentation is not needed, or (b) the shared memory locations written by library L can be determined from its interface, in which case we can instrument calls to L directly. Note that case (b) holds for standard C library functions such as `memset`, `memcpy`, and `strcpy`.

Another way to collect the modified pages is to use the `mprotect` system call to protect shared memory with no write permission at the beginning of each slice, and then use copy-on-write to collect the snapshot pages modified by the slice. We experimented with this approach, as it is the approach taken by DThreads [28], but we observed that this approach was less efficient than compile-time instrumentation due to the high frequency of page faults and `mprotect` system calls for programs with frequent synchronization (see our evaluation in Section 5).

4.3 Memory Modification Propagation

To do memory modification propagation, each thread maintains a list of *slice pointers* that contains pointers to all slices that happen-before the thread's current program counter. The slice pointers are organized in the happens-before order of these slices. Concurrent slices are organized in a deterministic order that is defined below (see "handling conflicts"). As shown in Figure 5, when we need to do propagation at an *acquire*, we collect the slices that happen before the *release* in the remote thread and append them to the *slice pointers* list of the local thread. As each new slice is appended to this list, we write the slice's modifications to local memory to make those modifications visible.

When deciding *which* slices to propagate, four issues must be considered. The first issue is to **propagate only happens-before slices**—that is, slices should be propagated only if they happen-before the current operation. For example, in the first propagation between $T1$ and $T2$ in Figure 6, $T1$ may already have produced the modification `x=3`, which is contained in the second slice of $T1$, but this slice should not be propagated as, according to DLRC, the modification `x=3` is not yet visible in $T2$. We set the vector time of the slice that succeeds the current *acquire* (the first lock in $T2$ for this example) as an *upperlimit* time to filter out these slices (line 5 of Figure 5).

The second issue is **transitive propagation**. Memory modification propagation must be transitive as the happens-before relation is transitive. Specifically, a slice of modifications can be propagated along several happens-before edges. As shown in Figure 6, `x=1` is propagated from $T1$ to $T2$ at the first synchronization, and is also propagated from $T2$ to $T3$ at the second synchronization. Since we copy all slices into the local *slice pointers* list during propagation, the slice containing `x=1` will be appended into the *slice pointers* list for $T2$ when $T2$ acquires the lock. That is, the *slice pointers* list for thread $T2$ contains all slices that must be propagated from $T2$ to $T3$, so transitive propagation will happen naturally.

The third issue is to avoid **redundant propagation**. Redundant propagation happens when the modifications of a slice are propagated to a thread which has already seen those modifications. In Figure 6, `x=1` is redundant in the propaga-

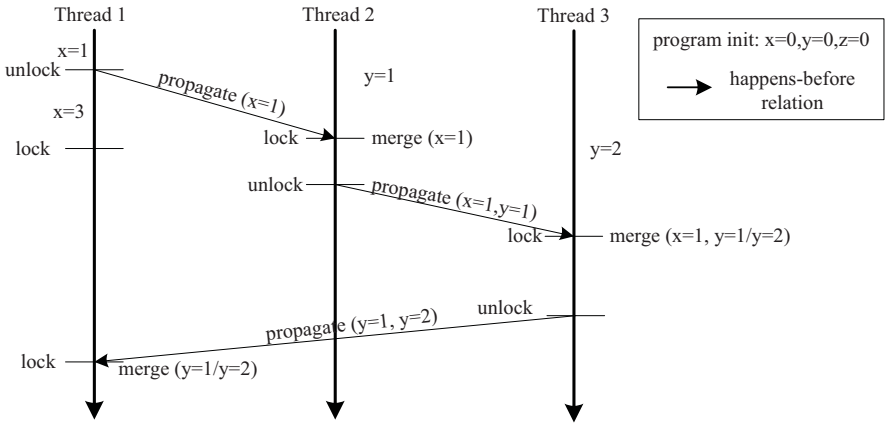

Figure 6. An example showing modification propagations in RFDet. The operation '`y=1/y=2`' indicates modification `y=1` overwrites modification `y=2`.

tion from $T3$ to $T1$. One way to identify a redundant slice is to check if it already exists in the set of local slice pointers. This method may be inefficient if the set of local slice pointers is large. A better method is to set a *lowerlimit* time to filter out the slices that have already been seen (line 6 of Figure 5), where *lowerlimit* is simply the vector timestamp of the slice that precedes the current *acquire* operation. Hence, any slice whose timestamp is smaller than *lowerlimit* must have been seen by the local thread (due to DLRC), and thus should be filtered out in propagation.

The last issue is **handling conflicts**. There are modification conflicts between threads due to W/W data races. As shown in Figure 6, `y=1` performed by $T2$ and `y=2` performed by $T3$ is a modification conflict, as they are not ordered by any happens-before relation. We resolve conflicts deterministically as follows. For barrier synchronizations, we sort modifications as described in Section 4.1. For all other synchronizations there is a unique "remote" thread, so we resolve these conflicts deterministically by always overwriting the local modifications with the remote modifications. So, in Figure 6, `y=1` overwrites `y=2`. The exception to this policy is that we prefer "local" writes when the "remote" write is redundant—the reason for this policy is subtle and will be explained in Section 4.6.

4.4 Memory Allocation

Since threads are implemented in processes, the default memory allocator in Linux (e.g., `malloc` in glibc) is invalid—dynamic memory allocations in different threads may cause *memory address conflicts*. For example, if `malloc` is called twice from two threads, it may return the same virtual address for these two newly allocated heap objects. These addresses will conflict in memory modification propagation. To avoid address conflicts, we modified the Hoard [8] memory allocator to store allocation information in the metadata space so it is shared among threads. Therefore, when a thread tries to allocate a memory region, we also reserve the virtual addresses of the memory region in other threads, which solves address conflicts. Note that these reserved virtual addresses may not be mapped with physical memory until they are touched by the local threads.

4.5 Optimizations

Garbage Collection. A slice stored in the metadata space becomes garbage when it has been propagated to every thread. We have to collect these garbage slices to prevent them from exhausting the metadata space. Therefore, we trigger garbage collection (GC) to reclaim unused slices when the metadata space usage reaches a predefined threshold. A slice is garbage when the timestamp of the slice is less than the current vector clock of every thread—such slices have already been merged into the local memory spaces of all threads.

Slice Merging. If we encounter an *acquire* operation that acquires a synchronization variable which was released by the same thread previously, we do not end the current slice. By continuing the current slice, we can avoid taking page snapshots and we reduce the number of pages that must be diffed. This optimization effectively merges the slices on both sides of the *acquire*. Note that this merging preserves the atomic property of slices stated in Section 4.2: all memory accesses in the merged slice will have the same happens-before relation to any instruction outside the slice. We omit the proof due to space limitation.

Prelock. In each critical section within a *lock*/*unlock* pair, we should propagate memory modifications according to the happens-before relation. However, memory propagation enlarges the original critical sections, leading to poor performance when lock contention is heavy. We cannot move memory propagation entirely out of the critical section as we cannot confirm the happens-before relation before the lock is acquired.

The *prelock* optimization is designed to shorten these long critical sections. The idea of *prelock* is to reserve the

lock first. The reservation phase defines the order in which threads will enter user's critical section. For example, suppose thread $T1$ attempts to acquire a heavily contended lock currently held by thread $T2$. $T1$ first adds itself to the reservation order. We do not yet know the complete happens-before relation for $T1$'s eventual acquire operation. However, we know that acquire must happen-after the current vector times of $T2$ and of every thread before $T1$ in the reservation order. Thus, $T1$ can begin merging memory updates that must happen-before its eventual acquire. It can do this in parallel with $T2$, even before $T2$ releases the lock. When $T1$ finally gets the token to enter the lock's critical section, it should first finish the unhandled memory modification propagations. After the critical section, it passes the token to the next thread according to the reservation order. This optimization can move a large percentage of propagation work into parallel mode (almost 80% in our experiment). Note that the reservation order is determined by the Kendo algorithm to ensure determinism.

Lazy Writes. This optimization reduces the number of unnecessary memory writes performed during memory propagation. We leverage the observation that not all the propagated memory modifications are needed by the local thread, thus we could postpone the write of these modifications until they are actually read by the local thread.

This optimization could reduce memory accesses in two ways. First, if the modifications are not used at all, then the writes for these modifications are omitted. Second, if the modifications are not accessed by the local thread for a long time, eager modification propagation would make multiple updates to the location before the first access, while lazy propagation makes just one update (containing the most-recent value). For example, suppose a thread executes 20 critical sections between two accesses of memory location X. In the worst case, the thread may receive 20 updates for location X (one at each critical section). With the lazy writes optimization, only the *last* update will be written.

When the lazy writes optimizations is enabled, we do not write the propagated modifications into the local memory directly. Instead, given a set of local pages to modify, we use page protection to protect each local page with no *Read* or *Write* permissions. Afterwards, when a memory access hits one of these pages, we write the modifications of the page into the local memory and unprotect the page.

4.6 Discussion

Correctness of Page Diffing to Accumulate Modifications. We are careful to store modifications at *byte* granularity (Section 4.2). The C++ memory model defines all memory actions as operations over scalars [13], and since the smallest scalar value in C++ is a byte, we must track memory modifications at byte granularity for correctness.

Recall that we construct modification lists for a slice by diffing each modified page with a snapshot containing the page's original values. It is not obvious that this diffing procedure produces correct modification lists. Specifically, what happens when a thread overwrites a memory location with the *same* value? For example, suppose `x==0` at the beginning of a slice, and suppose the slice executes the redundant assignment `x=0`. Because the final value of `x` is the same as its initial value, we do not include a modification for `x` in the slice's modification list, which means that the update `x=0` will not be propagated to other threads. Perhaps surprisingly, this is both deterministic and semantically correct. Consider two cases:

First, suppose the program is race-free. In this case, each read R of location `x` should read the value written by a unique write operation, W_1, where W_1 happens-before R. Specifically, there must a sequence of writes $\{W_1, W_2, \ldots\}$, where each W_i writes to `x`, and where each write is progressively older according to the happens-before relation. Suppose that $W_1 \ldots W_k$ all write `x=0`, and suppose that W_{k+1} wrote some value to `x` other than `0`. In this case, W_k is the youngest write that is not redundant, and the slice containing W_k will have the modification `x=0`. Hence, we guarantee that R reads the correct value (`x=0`) due to transitive propagation from the slice containing W_k to the slice containing R (recall Section 4.3).

Second, suppose the program has a data race. We must resolve all races deterministically. Our policy is to prefer writes that are *not* redundant. For example, in Figure 6, suppose that the initial value of `y` is `y=2`, making the write `y=2` in $T3$ a redundant write. The first slice in $T3$ will be empty, so when $T3$ acquires the lock, it will merge the write `y=1` from $T2$—this is effectively the same "remote write wins" policy that we described in Section 4.4. Now suppose that the initial value of `y` is `y=1`, making the write `y=1` in $T2$ a redundant write. The first slice in $T2$ is now empty, so the write `y=1` will not be propagated from $T2$ to $T3$. The result is that `y=2` will be kept in $T3$ and propagated to $T1$. Hence, our page diffing procedure effectively implements the following conflict-resolution policy: we prefer the "local" writes when the "remote" writes are redundant.

This policy can lead to unexpected results in programs with data races. For example, continuing with Figure 6, suppose that `y` is a 32-bit integer initialized to `y=0`, and suppose that $T2$ writes `y=256` while $T3$ writes `y=255`. Due to our policy of preferring non-redundant modifications combined with the fact that we compute page diffs at byte granularity, the final value merged in $T3$ will be `y=511` (as `255=0x00ff`, `256=0x0100`, and `0x01ff=511`). Of course, this result is deterministic, and it is also semantically valid, as the behavior of a C++ execution is undefined in the presence of a data race.

Ad Hoc and Lock-Free Synchronization. As in implementations of LRC [23], we assume that programs use system-provided synchronization operations only. Our implementation does not support ad hoc synchronization, such as via shared variable flags or lock-free algorithms. Pro-

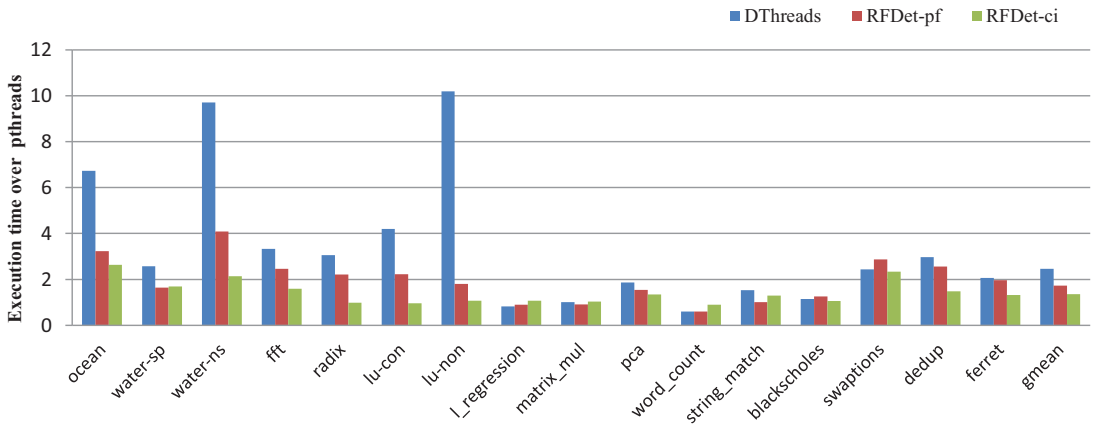

Figure 7. Execution time normalized to `pthreads`. All applications are running with 4 threads. RFDet-pf uses `mprotect` to monitor modifications, while RFDet-ci uses compile-time instrumentation to monitor modifications.

grams using ad hoc synchronization may be incorrect in DLRC (e.g., they may deadlock or violate atomicity) as the happens-before relations implied by ad hoc synchronization are not monitored and thus missed. There are two ways to remedy this. First, the programmer can wrap ad hoc synchronizations with a global lock to ensure they are serialized [36]. Second, we could extend our implementation to include an interface for ad hoc or lock-free synchronization, similarly to the new interface for low-level atomic operations in C++ [1, 2, 13], which we currently do not support. This new interface includes support for operations such as *atomic increment* and *atomic compare-and-swap*.

Totally supporting ad hoc synchronization and lock-free algorithms is future work. However, we do not foresee any major problems. To support C++ low-level atomics, we must use the Kendo algorithm to ensure that atomic operations happen in a deterministic order, and we must propagate memory modifications as described in Section 4.1, depending on whether the atomic operation being executed is an *acquire* and/or a *release*.

5. Evaluation

In this section, we present our evaluation results. All experiments were conducted on an AMD server with a 2.2 GHz, 12-core CPU (AMD Opteron 6174) and 16 GB physical memory, running Linux kernel version 2.6.31.5.

5.1 Methodology

To test whether RFDet can ensure determinism for multithreaded programs, we stressed it with *racey*—a parallel program which is designed to contain numerous data races to expose nondeterminism [20]. We verified the determinism of RFDet by running *racey* 1000 times with 2, 4 and 8 threads respectively. For each configuration, we got the same output for all the 1000 executions.

In the performance evaluation, we compare RFDet with DThreads [28]—a state-of-the-art DMT system, and `pthreads`—the conventional nondeterministic multithreading library. To test performance, we chose 16 parallel programs from three different benchmark suites, which are SPLASH-2 [35], Phoenix [31], and Parsec [10]. The SPLASH-2 suite was configured with c.m4.null.POSIX. This configuration uses *lock* and *unlock* to implement *barrier*. We use this configuration to make applications execute more synchronizations to stress performance. Moreover, successfully running benchmarks with this configuration shows that RFDet could support C/C++ codes as long as they correctly use these system-provided synchronization operations. Since RFDet does not support ad hoc synchronizations, we omit some benchmarks that contain complex ad hoc synchronizations in our experiments—they either cause deadlock (e.g., *fmm*) or violate atomicity (e.g., *canneal*). Note that DThreads does not support ad hoc synchronizations either [28].

5.2 Performance

To test performance, we ran each application 10 times with RFDet, DThreads and `pthreads` respectively, and gather their mean execution times, as shown in Figure 7. We provide two versions of RFDet which use different methods to monitor memory modifications. RFDet-ci uses compile-time instrumentation to monitor memory modifications, while RFDet-pf adopts the page protection method (recall the discussion in Section 4.2). As we can see, RFDet-ci and RFDet-pf incur an overhead of 35.2% and 72.9% respectively compared with `pthreads`. RFDet-ci performs better than RFDet-pf as it eliminates the overhead of page faults and system calls such as `mprotect` (as discussed in Section 4.2). Compared with DThreads, whose performance overhead is about 2.5x, both RFDet-ci and RFDet-pf have a smaller overhead—the performance improvements

Table 1. Profiling data of benchmark executions with 4 threads. In this table, *wait* refers to `pthread_cond_wait`, *signal* refers to both `pthread_cond_signal` and `pthread_cond_broadcast`, and *fork* refers to `pthread_create`. These programs normally execute equal number of *lock* and *unlock*, and execute equal number of *fork* and *join*, so we just show one number in these two columns.

	sync ops			memory ops				memory footprint & GC			
benchmark	lock/ unlock	wait/ signal	fork/ join	mem	load	store	store w/ copy	pthreads (MB)	RFDet (MB)	DThreads (MB)	GC
ocean	1100	671/199	6	36078529	29797587	6280942	77477	27	77.8	34.8	0
water-ns	6314	60/20	6	39256331	27183299	12073032	128983	5.9	53.3	11.0	9
water-sp	1103	90/30	6	89898824	64170352	25728472	13164	0.9	22.6	4.4	0
fft	54	21/7	6	163328252	87957717	75370535	49199	384	1012	450	0
radix	96	39/25	6	19087619	11675872	7411747	9422	40.5	295	107	0
lu-con	550	393/131	6	286770015	195163260	91606755	55806	16	60.1	25.7	0
lu-non	550	393/131	6	281461557	189840962	91620595	67364	8	100	43.9	0
linear_regression	0	0	16	35173933	19185782	15988151	2	0.004	4.0	1.6	0
matrix_multiply	0	0	16	3830399	3808551	21848	18	0.06	5.6	1.7	0
pca	816	0	32	3930114	3911170	18943	2034	1.5	76.9	1.7	0
wordcount	0	0	60	3607902	3215400	392502	149	2.1	56.6	3.8	0
string_match	0	0	8	15769972	12348432	3421540	2	0.02	4.1	1.6	0
blackscholes	24	0/1	4	1171467	1084629	86838	5	0.4	5.1	2.0	0
swaptions	24	0/1	4	28848349	21900213	6948136	2671	97.6	264	99.5	0
dedup	9304	152/3599	12	3345249	3327108	18141	12511	1310	5602	1506	5
ferret	43025	1/16	18	488092	419263	68834	4562	45.9	353	49.8	2

of RFDet-ci and RFDet-pf over DThreads are 81.6% and 42% respectively. Moreover, we noticed that the performance of RFDet is more stable than that of DThreads. As shown in Figure 7, the worst-case performance of RFDet is 2.6x slowdown (*ocean*), while the worst-case performance for DThreads is about 10x slowdown (*lu-non*).

Note that the major difference between DThreads and RFDet-pf is that we remove global barriers. DThreads introduces global barriers that may lead to poor synchronization schedules, thus causing load imbalance problems. Since RFDet does not introduce global barriers, it is more adaptable to a variety of synchronization patterns and has more stable performance overheads.

5.3 Performance in Detail

To analyze the performance results, we collected the profiling data of these program running in RFDet, as shown in Table 1. In this table, we provide the number of synchronization operations (we omit the number of barriers as none of the programs execute barriers in our configuration), the number of memory operations and the memory footprint for each benchmark application.

Theoretically, the performance of RFDet should be sensitive to the synchronization frequency of the user program. We have two reasons for this: 1) each synchronization may cause RFDet to perform memory modification propagation, which is the major overhead of RFDet; and 2) as we record memory modifications for each slice, higher synchronization frequency indicates more slices need to be recorded, which results in larger overhead. We can confirm this assumption from the profiling data of synchronization operations. For those applications that execute only a few synchronizations, such as *linear_regression*, *matrix_multiply* and *wordcount* in the Phoenix suite, the runtime overheads are small—they even improve performance over `pthreads` due to better cache behavior [28]. On the other hand, most applications in the SPLASH-2 suite execute a large number of synchronizations, thus their performance overheads are more significant.

The frequency of memory operations may also affect performance, especially for *Store* operations. As we can see from Table 1, the number of *Store* instructions is much smaller than the number of *Load* instructions. Moreover, in the common case for most *Store* instructions, our added instrumentation (Figure 4) performs only a few branch instructions to check if the *Store* hits a new page. Hence, only a small portion of these *Store* instructions will trigger a memory copy (see Column 9 in Table 1).

5.4 Memory Usage

One limitation of RFDet is that it consumes much more memory than `pthreads`, as shown in Table 1. The extra memory consumption comes from two sources. First, the isolated threads maintain a local copy of each shared variable, thus incurring an extra memory usage of $(N-1) * SharedMemory$, where N is the number of threads and $SharedMemory$ is the amount of non-stack memory allocated by the application. Second, the metadata space consumes extra memory. Memory consumption is shown in the last three columns of Table 1. Column 10 is the memory usage of the application alone, and Column 11 is the memory usage of the application running with RFDet. Specifically, we define Column 10 and Column 11 as the equations below, where N and $SharedMemory$ are defined above, and where $StackMemory$ is equal to total memory used by all stacks and $MetadataSpaceMemory$ is equal to total memory used in the metadata space.

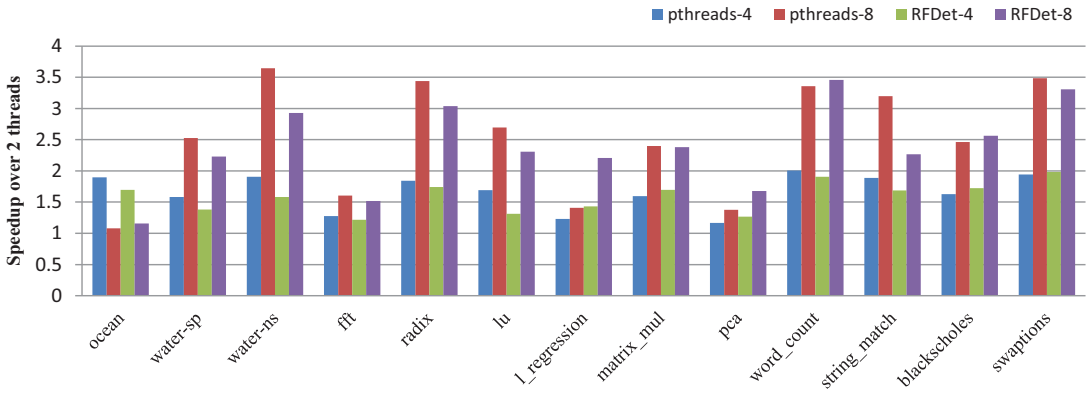

Figure 8. Scalability of RFDet-ci compared to `pthreads`.

$$Column10 = (SharedMemory + StackMemory)$$
$$Column11 = (N * SharedMemory + StackMemory + MetadataSpaceMemory)$$

Note that the memory consumption of the metadata space can affect the frequency of garbage collection, so we show the garbage collection count in the last column. We allocate memory in the metadata space when we either take a snapshot of a page (recall Figure 4) or convert these page snapshots to memory modification lists. The memory for storing a snapshot page is released immediately after we construct a byte-granularity modification list via page diffing. The memory for storing modification lists is released when we perform garbage collection. In our experiments, we set the size of the metadata space to 256MB and the threshold for GC to 90% metadata space usage, thus the corresponding GC count is shown in Table 1. However, when we set the metadata space to 512MB, there will be no GC for all these applications.

In pathological cases, even with garbage collection enabled, the slices in our metadata space can grow unboundedly large—this can happen if two threads execute for a long time without synchronizing. A specific example of this phenomenon is *linear_regression*: this benchmark has the least amount of communication (as it uses simple fork/join) but has the highest relative memory overheads. We could improve RFDet by using programmer annotations to identify threads that never communicate—this would enable eager collection of garbage slices and reduce the memory overheads of *linear_regression*, though we have not yet explored this idea.

Our Space/Time Tradeoff. Our approach investigates a space/time tradeoff in DMT. A fundamental cost of strong DMT is isolating threads' memory, which is often done using store buffers (CoreDet [4], RCDC [18], Calvin [21]) or memory protection (RFDet, DThreads, Conversion [29]). Previous systems use global barriers to limit the growth of isolated memory. Specifically, at each global barrier, isolated memory regions (e.g., store buffers) are flushed into a *global store*, which is then read-only during the next parallel phase (recall Figure 1). Hence, global barriers reduce memory pressure in these previous systems.

In contrast, RFDet eliminates the need for global barriers by giving each thread an isolated memory space *and eliminating the global store entirely*—since there is no global store to update, there is never any need for global communication, beyond that already required by the program's explicit synchronization pattern, and we can eliminate those global barriers required by previous DMT systems. The downside, as we have just seen, is that isolated memory regions can grow arbitrarily large in RFDet, in the worst case. Hence, while previous systems trade lower memory pressure for lesser-performance, we make the opposite tradeoff.

5.5 Scalability and Optimizations

We also tested the scalability of RFDet-ci. In this experiment, we ran each application with 2, 4 and 8 threads respectively, and calculated the speedups of the 4-thread and 8-thread executions with respect to the 2-thread execution. Currently, we cannot run *dedup* and *ferret* with 8 threads due to running out of memory, so they are not included in this experiment. We also use *lu-con* to represent *lu-non* as the results of these two applications are similar. As shown in Figure 8, RFDet scales well for most applications (its scalability is comparable to that of `pthreads`). Note that *ocean* does not scale from 4 threads to 8 threads for both RFDet and `pthreads` in our platform due to poor parallelism.

We used applications in the SPLASH-2 suite to test the effectiveness of our *prelock* and *lazy write* optimizations. We chose these applications because they use plenty of synchronization operations, so the effect of these optimizations is magnified. In this experiment, we first disable both optimizations (the baseline execution). Then we enable one of

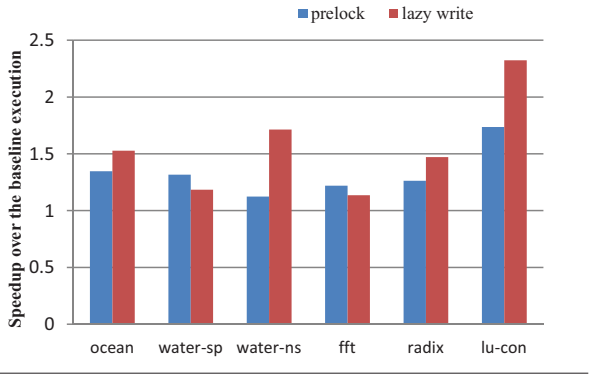

Figure 9. Optimization of *prelock* and *lazy write*.

the two optimizations each time and test the performance improvement over the baseline execution. Figure 9 shows the results of this experiment. As we can see, both optimizations improve performance, sometimes considerably.

6. Conclusion and Future Work

In this paper, we propose a new memory consistency model—deterministic lazy release consistency for C/C++ programs. In DLRC, only those modifications that happen before the current instruction should be visible. We use DLRC to implement an efficient deterministic multithreading system (RFDet) that, unlike prior systems, does not require global barriers. Our evaluation shows that RFDet ensures determinism with low overhead (35.2% on average, for the 16 parallel applications we tested). In the future, we will work on solutions for ad hoc synchronizations, e.g., providing interfaces for using ad hoc synchronizations or developing an automated tool for identifying and processing ad hoc synchronizations.

Acknowledgments

We thank our reviewers for their comments, which helped improve this paper greatly. This work is partially supported by National High-tech R&D Program of China (863 Program) under Grants 2012AA01A301 and 2012AA010901, by program for New Century Excellent Talents in University and by National Science Foundation (NSF) China 61272142, 61103082, 61003075, 61170261 and 61103193.

References

[1] S. V. Adve and J. K. Aggarwal, “A Unified Formalization of Four Shared-Memory Models,” IEEE Trans. Parallel Distrib. Syst., vol. 4, pp. 613-624, 1993.

[2] S. V. Adve and M. D. Hill, “Weak ordering—a new definition,” presented at the Proceedings of the 17th annual international symposium on Computer Architecture, Seattle, Washington, USA, 1990.

[3] A. Amittai, W. Shu-Chun, H. Sen, and F. Bryan, “Efficient system-enforced deterministic parallelism,” presented at the Proceedings of the 9th USENIX conference on Operating systems design and implementation, Vancouver, BC, Canada, 2010.

[4] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, “CoreDet: a compiler and runtime system for deterministic multithreaded execution,” presented at the Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems, Pittsburgh, Pennsylvania, USA, 2010.

[5] T. Bergan, L. Ceze, and D. Grossman, “Input-Covering Schedules for Multithreaded Programs,” in Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA), Indianapolis, Indiana, USA, 2013.

[6] T. Bergan, J. Devietti, N. Hunt, and L. Ceze, “The Deterministic Execution Hammer: How Well Does it Actually Pound Nails?,” in WoDET, 2011.

[7] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble, “Deterministic process groups in dOS,” in Proceedings of the 9th USENIX conference on Operating systems design and implementation, 2010.

[8] 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, Cambridge, MA, Nov. 2000.

[9] E. D. Berger, T. Yang, T. Liu, and G. Novark, “Grace: Safe multithreaded programming for C/C++,” in OOPSLA, 2009, pp. 81-96.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” in Proceedings of the 17th international conference on Parallel architectures and compilation techniques, 2008.

[11] R. L. Bocchino Jr, V. S. Adve, S. V. Adve, and M. Snir, “Parallel programming must be deterministic by default,” in Proceedings of the First USENIX conference on Hot topics in parallelism, 2009, pp. 4-4.

[12] H.-J. Boehm, “Position Paper: Nondeterminism is Unavoidable, but Data Races are Pure Evil,” in Proceedsing of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability (RACES), 2012.

[13] H.-J. Boehm and S. V. Adve, “Foundations of the C++ concurrency memory model,” presented at the Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, Tucson, AZ, USA, 2008.

[14] H. Cui, J. Simsa, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant, “Parrot: A Practical Runtime for Deterministic, Stable, and Reliable Threads,” in Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, Farmington, PA, USA, 2013.

[15] H. Cui, J. Wu, and J. Yang, “Stable deterministic multithreading through schedule memoization,” in Proceedings of the 9th USENIX conference on Operating systems design and implementation, 2010.

[16] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang, "Efficient Deterministic Multithreading through Schedule Relaxation," in Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, Cascais, Portugal, 2011.

[17] J. Devietti, B. Lucia, L. Ceze, M. Oskin, "DMP: deterministic shared memory multiprocessing,"presented at the Proceeding of the 14th international conference on Architectural support for programming languages and operating systems, Washington, DC, USA, 2009.

[18] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman, "RCDC: a relaxed consistency deterministic computer," in Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, Newport Beach, California, USA, 2011, pp. 67-78.

[19] C. J. Fidge., "Partial orders for parallel debugging," in ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, January 1989, pp. 24(1): 183-194.

[20] M. Hill and M. Xu. Racey: A Stress Test for Deterministic Execution. Available: http://www.cs.wisc.edu/ markhill/racey.html

[21] D. R. Hower, P. Dudnik, M. D. Hill, and D. A. Wood, "Calvin: Deterministic or not? Free will to choose," in High Performance Computer Architecture (HPCA), 2011, pp. 333-334.

[22] B. Kasikci, C. Zamfir, and G. Candea. "Data Races vs. Data Race Bugs: Telling the Difference with Portend," in Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2012.

[23] P. Keleher, A. L. Cox, and W. Zwaenepoel, "Lazy release consistency for software distributed shared memory," SIGARCH Comput. Archit. News, vol. 20, pp. 13-21, 1992.

[24] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "TreadMarks: distributed shared memory on standard workstations and operating systems," presented at the Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference, San Francisco, California, 1994.

[25] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," Computers, IEEE Transactions on, vol. 100, pp. 471-482, 1987.

[26] E. A. Lee, "The problem with threads," Computer, vol. 39, pp. 33-42, 2006.

[27] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy, "Chimera: Hybrid Program Analysis for Determinism," presented at the Proceedings of the 2012 ACM SIGPLAN conference on Programming language design and implementation, Beijing, China, 2012.

[28] T. Liu, C. Curtsinger, and E. D. Berger, "DTHREADS: Efficient Deterministic Multithreading," in Proceedings of the 22nd ACM Symposium on Operating Systems Principles, 2011.

[29] T. Merrifield, and J. Eriksson, "Conversion: Multi-Version Concurrency Control for Main Memory Segments," in EuroSys, 2013.

[30] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," in Proceeding of the 14th international conference on Architectural support for programming languages and operating systems, 2009, pp. 97-108.

[31] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," in Proceedings of the 13th International Symposium on High Performance Computer Architecture, Washington, DC, USA, 2007, pp. 13-24.

[32] D. Subhraveti and J. Nieh, "Record and transplay: partial checkpointing for replay debugging across heterogeneous systems," in SIGMETRICS 2011, pp. 109-120.

[33] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, et al., "DoublePlay: Parallelizing Sequential Logging and Replay," in Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, Newport Beach, California, USA, 2011.

[34] V. M. Weaver and S. A. McKee, "Can hardware performance counters be trusted?," in IISWC, 2008, pp. 141-150.

[35] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in Proceedings of the 22nd annual international symposium on Computer architecture, 1995, pp. 24-36.

[36] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma, "Ad hoc synchronization considered harmful," in Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation, 2010, pp. 163-176.

[37] X. Zhou, K. Lu, X. Wang, and X. Li, "Exploiting parallelism in deterministic shared memory multiprocessing," J. Parallel Distrib. Comput., pp. 72(2012)716-727, 2012.