CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution

Tom Bergan Owen Anderson Joseph Devietti Luis Ceze Dan Grossman

University of Washington, Computer Science and Engineering

{tbergan,owen,devietti,luisceze,djg}@cs.washington.edu

http://sampa.cs.washington.edu/

Abstract

The behavior of a multithreaded program does not depend only on its inputs. Scheduling, memory reordering, timing, and low-level hardware effects all introduce nondeterminism in the execution of multithreaded programs. This severely complicates many tasks, including debugging, testing, and automatic replication. In this work, we avoid these complications by eliminating their root cause: we develop a compiler and runtime system that runs arbitrary multithreaded C/C++ POSIX Threads programs deterministically.

A trivial non-performant approach to providing determinism is simply deterministically serializing execution. Instead, we present a compiler and runtime infrastructure that ensures determinism but resorts to serialization rarely, for handling interthread communication and synchronization. We develop two basic approaches, both of which are largely dynamic with performance improved by some static compiler optimizations. First, an ownership-based approach detects interthread communication via an evolving table that tracks ownership of memory regions by threads. Second, a buffering approach uses versioned memory and employs a deterministic commit protocol to make changes visible to other threads. While buffering has larger single-threaded overhead than ownership, it tends to scale better (serializing less often). A hybrid system sometimes performs and scales better than either approach individually.

Our implementation is based on the LLVM compiler infrastructure. It needs neither programmer annotations nor special hardware. Our empirical evaluation uses the PARSEC and SPLASH2 benchmarks and shows that our approach scales comparably to nondeterministic execution.

Categories and Subject Descriptors D.1.3 [*Programming Languages*]: Concurrent Programming—Parallel Programming; D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization, Run-time environments

General Terms Reliabity, Design, Performance

1. Introduction

1.1 Motivation

Nondeterminism makes the development of multithreaded software substantially more difficult. Software developers must reason about much larger sets of possible behaviors and attempt to debug without repeatability. Software testers face daunting incompleteness challenges because nondeterministic choices cause an exponential explosion in possible executions. Even otherwise useful techniques for making systems more reliable stop working. For example, if we run multiple replicas of a software system to guard against transient hardware failures, nondeterminism means the nonfailing runs may not agree on the result. In short, conventional wisdom strongly suggests minimizing sources of software nondeterminism.

Unfortunately, the state of practice in multithreaded programming is awash in nondeterminism. Thread scheduling, memory reordering, timing variations, and hardware features like bus control mechanisms can all lead to a multithreaded program producing different results. Indeed, we believe these sources of nondeterminism are a key reason to avoid multithreaded programming where possible. But with the move toward multicore architectures, multithreading is becoming less avoidable even though nondeterminism remains problematic. Parallel applications are intended to be deterministic: parallelism is for performance and should not affect outputs. By contrast, concurrent applications, such as servers, can accept some nondeterminism (e.g., the order requests are dispatched), but determinism would still simplify testing and replication. Rarely is the nondeterminism introduced by a parallel platform actually desirable.[1]

What we want, then, is a programming environment with the scalability and performance of multithreaded programming but the determinism of sequential programming. In most sequential languages, the outputs depend only on the inputs. While some parallel languages such as Data Parallel Haskell [10], Jade [29], NESL [6], and StreamIt [32] have a deterministic sequential semantics, much parallel code is written in conventional imperative languages such as C, C++, Java, etc. Even when languages are given a thread-aware semantics [8, 9, 21], implementations are allowed to be wildly nondeterministic — and typically are.

1.2 Our Approach to Determinism

In this paper we present CoreDet, a COmpiler and Runtime Environment that executes multithreaded C and C++ programs DETerministically. We require no additions to the language, no new hardware support, and no annotations from programmers. (Our determinism *guarantee* requires that the original program is memory-safe, though in practice even memory-safety violations are unlikely to introduce nondeterminism.) CoreDet is implemented entirely via a modified compiler that inserts extra instrumentation and a novel runtime system that controls how threads execute. Our general approach would work equally well for type-safe managed languages.

Unlike record-and-replay systems [20, 24, 26, 34], we do not need to save any information about fine-grained memory interleavings in order to replay a program's execution precisely — Core-

[1] Randomized algorithms desire nondeterminism, but in a specific way controlled by the application via some input source.

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.
Copyright © 2010 ACM 978-1-60558-839-1/10/03...$10.00

DET completely eliminates this source of nondeterminism. Clearly, though, the timing of asynchronous I/O remains a source of nondeterminism because it is controlled by the external world.

One key aspect of our system is that we do not specify to programmers *which* deterministic behavior we will produce, only that any given program will always produce the same outputs given the same inputs; this flexibility enables an efficient implementation but comes at a price, which is that a small change to a program can affect which deterministic program is produced.

If we ignore performance momentarily, a naïve approach could simply run one thread at a time in a fixed order and switch threads at deterministic points (e.g., after n instructions are executed). COREDET optimizes this naïve approach by recovering parallelism when threads do not communicate with each other and by serializing threads when they do communicate. Determinism follows from all interthread communication happening in a unique deterministic order. Scalable performance is achieved by minimizing serialization, but unfortunately, we cannot determine the minimal amount of necessary serialization without overhead. So, in this paper, we describe three deterministic execution strategies that explore the tradeoff between good scalability (requiring high overhead) and poorer scalability (requiring less overhead).

1.3 Evaluation and Contributions

We have evaluated COREDET using the SPLASH2 [33] and PARSEC [5] benchmark suites. Because our compiler adds instrumentation to many memory loads and stores, we slow down each application thread (roughly 1.2x-6x). However, we have developed static optimizations that remove some checking when it is provably unnecessary. More importantly, our instrumentation does not prevent our benchmark applications from scaling. When we run the applications with 2, 4, or 8 processors, we typically see a similar relative performance improvement over 1 processor as we do with a conventional nondeterministic implementation.

Recently, as part of the DMP project, Devietti et al. [11] described a few ways to execute arbitrary imperative programs deterministically. This work builds and improves upon that work substantially. First, we present the first weakly-consistent model of deterministic parallel execution and demonstrate that relaxing memory ordering can enable more scalable performance in deterministic execution. Second, prior work in DMP focused on hardware-based enforcement of determinism. The associated compiler was a direct implementation of the proposed hardware support in software and did not have static optimizations. Finally, this paper conducts a thorough empirical evaluation and a careful study of the effect of several configuration parameters.

1.4 Outline

Section 2 describes our approach to determinism at a high level. Sections 3, 4, and 5 explain our deterministic execution strategies in detail. Section 6 provides details of our compiler. Section 7 provides details of our runtime system and its interaction with threading libraries and the operating system, and includes a discussion of how memory safety affects our deterministic guarantee. Section 8 presents a comprehensive evaluation of performance, analyzes the impact of configuration parameters, and shows characterization data to help understand the behavior of COREDET. Finally, Section 9 discusses related work and Section 10 concludes.

2. High-Level Approach

This section presents a series of ways to compile and run a program such that it runs deterministically. The first is serial execution, which is included for expository purposes. The next four, DMP-TM, DMP-O, DMP-B, and DMP-PB, optimize the serial execution to recover parallelism. Only the last three, DMP-O, DMP-B, and DMP-PB, have been implemented in COREDET; we discuss the implementation difficulties of DMP-TM below and in Section 8.4.

2.1 Starting Serial

A naïve way to run a multithreaded program deterministically is to serialize its execution in a deterministic way, which is our starting point. At runtime we schedule threads in a simple round-robin fashion so that execution is serial. Each thread is scheduled for one finite logical time slice, or *quantum*; a *round* consists of all threads executing one quantum each.

To ensure determinism it suffices to ensure that the length of each quantum and the scheduling order are both deterministic. Conceptually, the compiler inserts code to count how many instructions are executed and ends each quantum after a fixed number. Section 6.1 describes how this is done efficiently in our compiler. Also, we always add new threads to the end of the scheduling order, remove them from the order when they exit, and do not change the order otherwise.

2.2 Going Parallel

The conceptually simplest way to recover parallelism is to use transactional memory. Starting with a serial execution, we can schedule multiple quanta in parallel by enclosing each quantum in an implicit transaction. As long as transactions commit according to the serial scheduling order, the resulting parallel execution is exactly equivalent to the (deterministic) serial execution. Thus, the parallel execution is deterministic. This is called DMP-TM, and is described more thoroughly in [11].

DMP-TM is an attractively simple strategy. It produces an efficient schedule in which threads are serialized only when they communicate; there is very little unnecessary serialization. Unfortunately, we cannot implement DMP-TM in COREDET with just a few simple modifications to existing STM implementations. There are multiple difficulties, a few of which we highlight here. (It is worth noting that when HTM with support for large transactions is available, these difficulties will be less serious.)

First, while STM has been shown to have good performance for some applications, this was under the assumptions that transactions are small and most execution is outside transactions. Neither assumption applies in our case. In DMP-TM, all code executes transactionally and transactions would need to be large in order to amortize the cost of commit. Moreover, quanta in COREDET are not lexically scoped. (Recall that we form quanta by instruction counting; as will be seen later, this strategy creates balanced quanta which are important for scalable performance.) Since quanta are not lexically scoped, implicit transactions in DMP-TM may begin and end at any point in the execution. The runtime, then, must be prepared to rollback the call stack to an arbitrary prior state whenever a transaction aborts. This requirement would add significant complexity and overhead to our purely software implementation. For all these reasons, we have chosen to develop approaches that do not rely on speculation.

2.3 Two-Stage Rounds

In the rest of this paper we present three alternatives that avoid the need for speculation. We start by adopting a more conservative, and less speculative, approach. The basic idea is to divide each round into a *parallel mode* and a *serial mode*. In parallel mode, threads run in parallel but we do not allow them to communicate. In serial mode, threads are scheduled serially and they are allowed to communicate arbitrarily. A thread ends its parallel mode once it has either exhausted its quantum or reached an instruction that might communicate with other threads. Serial mode begins once all threads have completed parallel mode, and ends once all threads

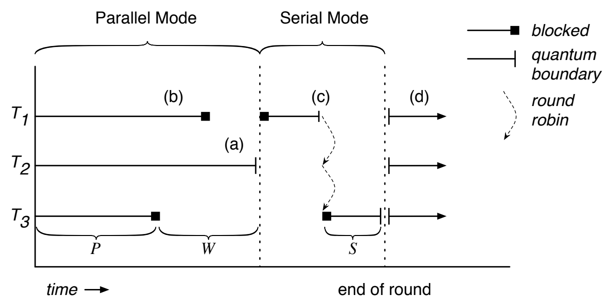

Figure 1. Timeline of a quantum round, showing the division into *parallel mode* and *serial mode*. T_2 finishes its quantum in parallel mode (a), while T_1 and T_3 have work left for serial mode (b,c). Once T_3 finishes its serial mode, the round ends and the next one begins (d).

have had a chance to run. In this way the parallel and serial modes are isolated by global barriers, producing the two-stage round illustrated in Figure 1.

Already we can begin to understand the requirements for determinism and scalable performance. Because the two modes are isolated by global barriers, execution is deterministic when each mode is deterministic in isolation — the barriers provide deterministic transitions between each mode. Parallel mode is deterministic if it ends at a deterministic point and if there is no communication; the absence of communication eliminates the possibility of nondeterministic data races. Similarly, serial mode is deterministic if it ends at a deterministic point.

Scalable performance requires minimal serial modes and balanced parallel modes. In a given round each thread executes for a total time of $P + S$, where P is the time spent executing in parallel mode and S is the time spent executing serially in serial mode (indicated by brackets on T_3 in Figure 1). It is good to remember that scalability is limited by $S/(P + S)$ (Amdahl's Law). Each thread spends on average an additional time W waiting at the barrier between parallel and serial mode. Both W and S represent lost parallelism, so we obviously want to minimize them. W is minimized when parallel mode is *balanced*. S is minimized when all quanta execute completely in parallel mode, making serial mode unnecessary.

Section 3 presents DMP-O, which has very low overhead but the poorest scalability; it is very conservative about what code can run in parallel mode. DMP-O was first implemented, in hardware, by [11] as DMP-SHTAB. Our presentation here is phrased using software terminology and is included for completeness. Section 4 presents DMP-B, which has higher overheads but provides very good scalability by relaxing memory ordering. Section 5 presents DMP-PB, which is a hybrid of DMP-O and DMP-B that provides a middle ground in terms of both overhead and scalability. Those three sections describe how we make loads and stores deterministic. Section 7.1.2 describes our implementation of a deterministic pthreads library which implements objects such as mutexes and barriers in terms of a deterministic compare-and-swap primitive. Finally, Section 8.4 argues empirically that DMP-B scales comparably to DMP-TM even though it does not require speculation.

3. DMP-O: Ownership Tracking

We can divide memory locations into those that are *thread-private* (i.e., only ever accessed by one thread T) and those that *may-be-shared*. A conventional static escape analysis can identify memory accesses that must always access thread-private data (versus those that may access shared locations). A thread could then execute in parallel mode as long as it only accesses thread-private data. At the first may-be-shared access, it blocks until serial mode. The serial mode, then, is used to deterministically serialize all accesses to may-be-shared data. Determinism follows directly from this observation.

This approach exploits thread-private data but assumes that the ownership information for a location does not change during execution. This is a poor assumption for arbitrary C/C++ code, in which the same memory location may dynamically transition from being thread-private to thread-shared and back, possibly to a different thread. We do not want to block on every access to a location that might at some point in the execution be shared. Moreover, any static escape analysis is imprecise and so it will conservatively categorize many accesses as *may-be-shared*. Therefore, we track the ownership of memory locations at runtime and instrument *may-be-shared* memory accesses to check and possibly change the ownership information.

A runtime hashtable we call the Memory Ownership Table (MOT) maintains the ownership status for each location. (A location can have any granularity, but we use the same granularity for all locations.) At any point, the status of an entry indicates which thread T owns the location. If T owns a location, then T can access the location in parallel mode. If not, then it must wait until serial mode. At this point, in serial mode, T changes the MOT so that T owns the location and proceeds. Because status changes occur only during serial mode, access to the MOT is well-synchronized. In parallel mode, when multiple threads are accessing the MOT, the MOT is immutable. Determinism is preserved because the point at which each thread blocks is deterministic (the transition from parallel to serial mode is deterministic) and prior to this point there is no interthread communication.

Note that any changes to the MOT are purely a matter of policy. When T_1 accesses a location owned by thread T_2, the access occurs in serial mode and our policy is to change ownership to T_1. The advantage of this policy is locality: if T_1 is likely to access the location again soon, then it may not need to block before doing so. But any deterministic policy—e.g., leaving the ownership with T_2, setting it to another thread T_3, etc.—is correct. Similarly, the initial state of the MOT does not affect correctness provided it is deterministic.

3.1 Nonblocking Shared Reads

The approach described thus far does not yet support parallel reads of the same location, which is essential for scalable performance in most applications. So far, for an access not to block, the location accessed must be owned by the running thread and there can be only one owner.

To fix this, we extend the MOT to support a *shared* status for locations and allow reads (but not writes) to shared locations to proceed during parallel mode. To do this without introducing nondeterminism we must disallow all parallel writes to data in *shared* status. Hence our access rules are now as follows:

1. If the location is owned by T, then T proceeds immediately.
2. If the location is shared and the access is a read, then T proceeds immediately.
3. If the location is shared and the access is a write, then T can proceed once T is scheduled in serial mode.
4. If the location is owned by T_{other}, then T can proceed once T is scheduled in serial mode.

Figure 2 represents these possibilities in a flowchart. While (3) delays executing writes to shared locations until all other threads are blocked, we favor reads over writes, which is the right performance trade-off. Moreover, in cases (3) and (4) we can now use any

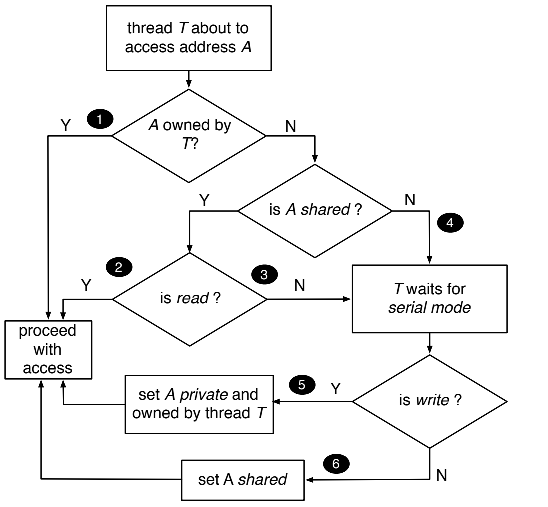

Figure 2. Memory Ownership Table policy

(deterministic) policy to change the MOT so as to hopefully avoid blocking on subsequent accesses. For writes, we choose the same policy as before: a write by T changes the status to owned by T (see (5) in the flowchart). For reads, we choose to change the status to shared (see (6) in the flowchart), so that reads by other threads in future rounds can proceed in parallel. As a slight modification to this policy, Section 6.2.2 describes a compiler optimization that uses static information to choose (deterministically) to change the status to owned by T for some reads in order to remove instrumentation on subsequent accesses.

While program execution is still deterministic, it is *not* the *same* deterministic execution we would have had without shared reads. In fact the quanta are no longer necessarily serializable. To see why, consider this example, assuming the round-robin order begins with thread T_1 and x begins shared and holding 0:

	Thread T_1	Thread T_2
First Quantum	...	...
	`x:=1`	`y:=x`
	...	...

The original serial execution would assign 1 to y, but now thread T_1 will not proceed beyond its assignment to x until all other threads are blocked, which will be after T_2 reads 0 from x if nothing before `y:=x` causes T_2's quantum to block.

3.2 The Length Of Serial Mode

We have not yet discussed how much work should be done in serial mode. Recall that each quantum has a budget of n instructions. Parallel mode ends when either the budget is exhausted or a nonprivate access is reached. If the budget has not been exhausted, then the thread has work to do in serial mode. Again, how much work should be done in serial mode?

In answering this question we must consider competing factors. First, if serial mode is too long, the program will be over-serialized and will suffer from poor scalability. Second, programs usually exhibit temporal locality, so a nonprivate access will likely be followed by other nonprivate accesses. If serial mode is too short, these other accesses will block the thread early in its parallel mode of the next round, again causing poor scalability. Finally, the global round barriers incur some overhead which is best amortized by using fewer rounds (and thus, longer serial modes).

We tried two approaches: *full serial mode*, which completely exhausts the quantum budget in serial mode that remains from the previous parallel mode; and *reduced serial mode*, which ends serial mode immediately once a thread releases a mutex if it no longer holds any mutexes. The insight of this second technique is to end serial mode at the end of a critical section, *i.e.* at the end of a period of interthread communication. Section 8.3 explores these two techniques.

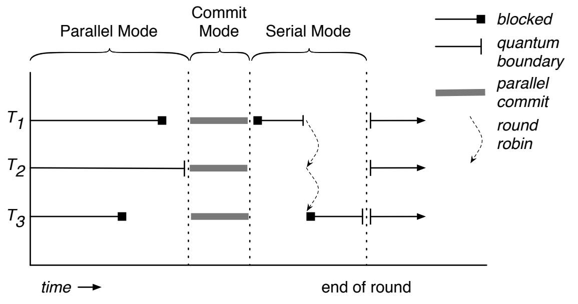

Figure 3. Timeline of a quantum round in DMP-B, showing the addition of the new parallel *commit mode* prior to serial mode.

4. DMP-B: Store Buffering

Our second approach to deterministic execution dispenses with the idea of ownership tracking altogether. Instead, we use a multi-versioned model where each thread has its own private store buffer. During parallel mode, all writes are entered directly into the private store buffer. Each read of some location L first consults the store buffer. If L was written to earlier in the same quantum then the value is fetched from the store buffer; otherwise, the value of L is fetched from shared-memory. At the end of parallel mode we enter a *commit mode* in which threads commit their writes to the global shared-memory space. The commit happens serially according to the deterministic scheduling order. (Later we explain how to perform this commit in parallel.)

Determinism follows from two observations. First, during parallel mode each thread has its own private view of memory which it updates independently from all other threads. As a consequence, shared-memory is read-only during parallel mode and there is no communication. Second, updates to shared-memory happen in a deterministic order during commit mode.

Notice, however, that these updates are not sequentially consistent. Stores performed by thread T in some quantum are delayed to the end of parallel mode, effectively reordering the stores with respect to loads performed by T in parallel mode. What we have described is a weakly-consistent memory model. For correctness, then, we must add the following two restrictions. First, parallel mode must end once a memory fence is reached. This implements the semantics of a full memory fence, allowing the programmer some guarantees about the order of operations across threads. Second, atomic operations such as compare-and-swap cannot execute in parallel mode. The reason is that the parallel mode of thread T does not happen atomically with the commit mode of T; the commit mode of some other thread may intervene and violate atomicity. Therefore we delay atomic operations until serial mode. In this way, DMP-B is composed of three-stage rounds as illustrated in Figure 3.

4.1 Memory Consistency Model

DMP-B improves scalability by relaxing memory ordering, so execution is no longer sequentially consistent as in DMP-O. In the following, we describe DMP-B's memory consistency guarantees

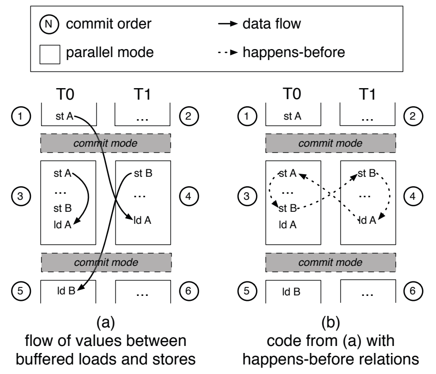

Figure 4. Execution under DMP-B adheres to the weak ordering memory consistency model.

```
CommitChunk(c: Chunk) {
  if (NumAllocatedChunks[c.addr] == 1) {
    Publish(c)
    return
  }
  lock(LockTable[c.addr])
  foreach (x ∈ CommitTable[c.addr]) {
    if (x.thread > c.thread)
      c.bitvector &= ~x.bitvector
  }
  Publish(c)
  CommitTable[c.addr] += {c}
  unlock(LockTable[c.addr])
}
```

Figure 5. Deterministic parallel commit

in more detail and explain why DMP-B does *not* violate the semantics of the original program.

Figure 4 illustrates why DMP-B is not sequentially consistent. The circled numbers indicate the order in which each quantum's stores are serialized during commit mode. The solid arrows in Figure 4a show where each load gets its data. The concurrent loads in $T0$ and $T1$ read different values of A as $T0$ reads from its store buffer and $T1$ reads from shared-memory. The deterministic commit order ensures that $T0$'s load of B in quantum 5 receives its value from quantum 4's store, not quantum 3's. Figure 4b shows the same code as in Figure 4a annotated with causal happens-before relations indicated via dashed arrows. The downward arrows within quanta 3 and 4 are due to program order, the `st B` $\rightarrow$ `st B` arrow is due to commit ordering, and the `ld A` $\rightarrow$ `st A` arrow is due to the fact that the load of A received its value from quantum 1's store (see Figure 4a). The happens-before cycle shows how the logical reordering due to store buffering results in a non-sequentially consistent outcome.

Thus, to honor the semantics of memory fences in a program, at every fence we must transition to commit mode and flush all store buffers to shared-memory via the commit process. This is analogous to the implementation of a memory fence instruction in hardware, which flushes hardware buffers that contain the values of pending store instructions. Since fences force the end of a quantum, they may introduce imbalance into quantum formation. However, when fences are infrequent, balanced quantum formation is not perturbed substantially.

Most importantly for programmers, DMP-B does not violate the semantics of the original program. Specifically, DMP-B preserves sequential consistency for data race free programs as required by the Java and C++ memory models [9, 21]. This fact follows from the following three observations: (1) our commit mode imposes a total order on all stores to shared-memory, (2) our serial mode imposes a total order on all synchronization operations, and (3) every synchronization operation acts as a full memory fence. Further, DMP-B obeys the "out-of-thin-air" requirement of the Java memory model because all loads return a value from a store that either happens-before the load or commits in an earlier quantum (and thus cannot happen-after the load).

As DMP-B flushes store buffers on every fence (not distinguishing between different types of fences), it implements a weak ordering memory consistency model [2], specifically total store ordering (PowerPC and Sparc processor architectures have a flavor of such a model). Thus, DMP-B does not noticeably weaken the consistency model provided by modern execution stacks. While the DMP-O and DMP-TM strategies execute programs in a sequentially consistent way, DMP-B shows that, just as with nondeterministic parallel execution, stronger consistency guarantees can be traded for performance gains.

4.2 Deterministic Parallel Commit

For scalable performance it is vital that we extract as much parallelism from commit mode as possible. In this section we describe our algorithm for deterministic parallel commit. To preserve determinism, our algorithm must publish store buffers in the same logical order as if they were committed serially. We achieve this by allowing store buffers to commit in any order, while maintaining enough metadata to prevent one thread's writes from being overwritten by some other thread that precedes it in the logical commit order. In the following, we start by describing the layout of a store buffer, then describe our locking protocol which ensures the commit order is deterministic, and finally describe an optimization to reduce locking overhead.

A store buffer is a collection of *chunks*. A chunk buffers a fixed amount of contiguous data, for example one cacheline. Each chunk is linked into a hash table (used to service loads) and a queue (which is processed during commit). For each chunk we keep a bitvector with one bit per byte of data in the chunk. A bit is set in the bitvector if the corresponding byte was written earlier in the same quantum.

During commit mode threads can commit their chunks in any order. To commit chunk C at address A, thread T first locks address A. (We use a table of locks where each lock guards some region of memory.) If T is the first thread to commit a chunk at address A, it publishes chunk C to shared-memory and then inserts chunk C into a global *commit table*. Otherwise, T enumerates from the commit table all chunks with address A that were published by threads that logically happen-after T. Recall that in order to preserve the deterministic commit order, T must not overwrite any bytes published by those threads. So, T masks the already-published bytes out of C's bitvector, publishes C, and inserts C into the commit table. Finally, T unlocks address A.

Acquiring a lock for each chunk can incur heavy overheads. As an optimization, we publish chunks without acquiring any locks when we are certain there are no conflicting writes. To facilitate this optimization, we maintain a global hash table which maps addresses to counts; when allocating chunk C with address A, we increment the counter for address A. (We use a fixed-sized hash table so that updates can be performed by simple atomic increments.) If at the end of the quantum the counter for address A is 1, then C was the only chunk allocated for A during the quantum and it can be published without acquiring a lock.

Figure 5 summarizes this algorithm with pseudocode. Lines 2–4 show the fast path: chunk c was the only chunk allocated at address $c.addr$ during the quantum, so we can publish chunk c without acquiring a lock. Lines 6–13 show the slow path: we acquire a lock (line 6), mask out bytes from chunks that happen-after chunk c in the serial commit order (lines 7–10), publish the chunk (line 11), save the chunk in the global commit table (line 12), and finally release the lock (line 13).

5. DMP-PB: Partial Buffering

DMP-B has good scalability because almost all operations can proceed in parallel. However, the instrumentation cost is higher since the store buffer needs to be accessed frequently. To reduce this overhead we borrow an idea from DMP-O: We partition the address space into locations that are dynamically *thread-private* and locations that are *shared*. Accesses by thread T to its own private data can proceed in parallel mode without consulting the store buffer, while accesses by T to another thread's private data must wait for serial mode. Accesses by T to shared data (be they reads or writes) can proceed in parallel mode but must consult the store buffer.

This scheme, which we call DMP-PB, is effectively a hybrid of DMP-O and DMP-B. We use a MOT to partition the address space into private and shared data, and we use DMP-B to allow concurrent access to shared data in parallel mode. Notice DMP-PB is equivalent to DMP-B when all locations are *shared* in the MOT. This trivial policy is clearly undesirable because it does not reduce the overhead of DMP-B.

To define a useful MOT policy for DMP-PB, we extend the MOT to support an *always-shared* state. This state behaves exactly like the *shared* state; accesses to *always-shared* data can proceed in parallel mode by consulting the store buffer. However, once a location has transitioned to the *always-shared* state, it never transitions back to *shared* or *thread-private*. The insight is that we use *always-shared* for locations that are being written by multiple threads so that those writes may proceed in parallel mode. Our policy for evolving the MOT in serial mode of DMP-PB is now defined by the following rules, where T is the thread performing the access: (1) If the location is *shared* and the access is a write, change the status to *owned-by* T; (2) If the location is *owned-by* T', change the status to *always-shared*; (3) Else leave the state as is. We initialize all MOT states to *shared* to allow parallel access to shared-memory by default.

In summary, DMP-PB uses thread ownership information offered by the MOT to filter accesses to the store buffer that cannot be proven private statically. This reduces overhead for applications with infrequent interthread communication.

6. Compiler Transformations

COREDET's compiler transformations are implemented as an additional pass in the LLVM [19] compiler infrastructure. This pass instruments the application to enforce deterministic execution according to either DMP-O or DMP-B (DMP-PB is instrumented identically to DMP-B). As discussed later, our runtime also provides a custom threading library, so we use preprocessor macros to automatically convert pthreads calls into our runtime calls.

To form quanta of bounded size, the compiler statically inserts calls to `CDcommit`(i) as described below. To enforce deterministic interthread communication, the compiler instruments loads and stores with calls to our runtime. These calls differ slightly depending on which execution strategy is being compiled for:

DMP-O: Calls to `CDload`($addr, size$) and `CDstore`($addr, size$) are inserted just before loads and stores, respectively. These functions apply our MOT policy for an access at the given location, of the given size.

DMP-B: Calls to `CDloadT`($addr$) and `CDstoreT`($addr, value$) replace loads and stores, respectively. These type-specialized functions emulate loads and stores by redirecting them to the local store buffer as necessary.

This instrumentation pass is fully compatible with all standard compiler optimizations. However, the calls to `CDcommit`, `CDload`, and `CDstore` drastically inhibit the opportunity for optimization once they have been added. Because of this we always run a full optimization pass before applying our deterministic transformations.

In the remainder of this section we explain how our pass achieves reasonably balanced quanta, and then we describe some static optimizations that remove instrumentation from loads and stores when it is provably unnecessary.

6.1 Balanced Quantum Formation

For scalability, it is essential to keep the amount of work per quantum as uniform as possible. Nonuniform quanta lead to excessive waiting. Notice that there are two sources of imbalance. First, there is imbalance caused by a thread blocking for serial mode, for example to access shared data via DMP-O, or to execute a memory barrier via DMP-B. Second, there is imbalance caused by differing quantum lengths. This is the source of imbalance we strive to eliminate here. Our goal is to maximize the balance of parallel mode under the assumption that serial mode is rarely necessary.

A quantum is a certain bounded amount of dynamic work. Of course, the compiler cannot precisely estimate dynamic work nor annotate specific dynamic points in the execution. Instead we count work dynamically by setting the i parameter in `CDcommit`(i) to be the expected work done since the last call to `CDcommit`. We define *work* using a cost function over LLVM's IR which maps each opcode to its expected runtime latency. At runtime, `CDcommit`(i) subtracts i from a thread-local counter and ends the quantum when the counter reaches 0. Quantum length is deterministic because each call to `CDcommit`(i) uses a constant value for i. Note that a small change to the program can change the value for i, which can affect which deterministic program is produced by changing where quantum boundaries occur.

There is a trade-off between the precision of work counting and the overhead of instrumentation. In the following, we first explain how we use `CDcommit` to count work exactly, and then present optimizations to remove some calls to `CDcommit` to reduce overhead at the cost of some imbalance.

6.1.1 *Maximal Balance*

The simplest precise strategy is to call `CDcommit`(i) at the end of every basic block, where i is the amount of work done in that block. An equivalent strategy is to call `CDcommit`(i) at the end of every linear sequence of basic blocks, where i is the total amount of work done in that sequence. More precisely, we call `CDcommit`(i) at the end of basic block B if some successor of B has multiple predecessors; *i.e.,* if some successor of B is a merge node. In addition, we call `CDcommit` before every function call and function return so the thread-local work counter is up to date before and after each call.

6.1.2 *Minimal Balance*

At minimum, to guarantee progress, we must call `CDcommit` at some regular interval within every loop that is not provably terminating. (Otherwise, some thread could go into an infinite loop without ending its quantum and no other thread would make progress.) A conservative estimate, which we use, is to call `CDcommit`(i) before every backedge and function call. In this strategy, the true value of i depends on the specific path taken since the last call to `CDcommit`. We estimate this value by averaging the amount of work accrued over all possible incoming paths. All such averages in a function can be computed in one linear pass by processing the control-flow graph in topographic order.

6.1.3 *Heuristic Balance*

The final strategy is a hybrid of the previous two. It attempts to find a trade-off between maximal balance and minimal instrumentation.

We start with maximal balance. At each merge node N, we compare the values of i for the call to `CDcommit(`i`)` in each predecessor of N. If these values differ by a small heuristically-chosen threshold, and if none of the predecessors of N have an outgoing backedge, then we remove the call to `CDcommit` from each predecessor of N. We do this greedily until there are no more calls to remove. For the remaining calls to `CDcommit(`i`)`, i is computed as the average of work accrued over all possible incoming paths.

6.2 Static Optimizations

In this section we describe two optimizations that remove calls to `CDload` and `CDstore` when they are provably unnecessary. The first, escape analysis, is applicable to both DMP-O and DMP-B. The second, redundant call removal, is only applicable to DMP-O.

6.2.1 *Escape Analysis*

Accesses to provably thread-local data do not need to be instrumented.[2] Note that because LLVM lowers all stack-allocated scalars that do not have their address taken into virtual registers, the majority of thread-local accesses are automatically identified by LLVM with no extra work from our pass.

Although this optimization is applicable to both DMP-O and DMP-B, there are subtleties when applying it to DMP-B. For example, consider the following snippet:

```
int local;
int *p = (...) ? &local : &global;
...
```

The pointer `p` may-alias shared data, so all accesses of `*p` must be instrumented. Because `p` may-alias `local`, we must *also* instrument accesses to `local`, even though the address of `local` does not escape. If we did not instrument `local`, then direct accesses of `local` would not consult the store buffer, while indirect accesses of `local`, through `p`, would. That is incorrect behavior: We must ensure that all accesses to an object either consult the store buffer or do not consult the store buffer.

Therefore, when applied to DMP-B, our escape analysis respects an extra constraint: An object is thread-local only if all accesses of that object are through pointers that must-alias thread-local objects. We use a points-to analysis based on Data Structure Analysis [18]. DSA is unification-based, so all pointers which may-alias the same object will point at the same node in the points-to graph. This automatically satisfies our extra constraint: we simply label each node in the points-to graph as either "thread-local" or "escaping".

6.2.2 *Redundant Call Removal*

This optimization removes calls to `CDload` and `CDstore` when they are provably redundant. It is not applicable to DMP-B because we must consult the store buffer on *every* may-be-shared access.

Given two accesses A_1 and A_2 that are redundant in some way, we can remove the instrumentation from A_2 if whenever A_2 executes, then A_1 must execute earlier in the *same* quantum. In other words, A_1 dominates A_2 and there does not exist a path from A_1 to A_2 that contains a call to `CDcommit`. If A_2 is a store and A_1 is a load, then we also must instrument A_1 as if it were a store.

The two types of redundancy are:

- A_1 and A_2 are accesses of the same size that must-alias. This is a form of common-subexpression elimination.
- A_1 and A_2 are accesses that are provably spatially close, where "close" is heuristically defined. For example, they refer to different fields of the same heap object or different elements of the same array. In this case, the instrumentation on A_1 should be replaced by instrumentation that covers the full range of both accesses. This is an "access coalescing" optimization.

To see why this optimization is correct, first recall that there cannot be a quantum boundary between A_1 and A_2. Then consider three cases. In the first case, A_1 and A_2 both execute in parallel mode. The instrumentation on A_1 will verify that A_1 can execute in parallel mode; since the accesses are redundant, A_2 can also safely execute in parallel mode. In the second case, A_1 and A_2 both execute in serial mode. Again, instrumentation on A_1 subsumes instrumentation on A_2. In the third case, A_2 executes in serial mode while A_1 executes in parallel mode. It is safe to execute A_2 in serial mode without consulting the MOT since there is only one thread running. However, because A_2 is not instrumented, the MOT will not be modified even if A_2 accesses some location that is currently private to another thread. This lack of MOT change can affect future rounds. In other words, enabling this optimization can produce a *different* deterministic program than the one produced when this optimization is disabled.

7. System Issues

This section discusses special considerations for threading libraries, other external libraries, memory allocation, and C/C++ memory errors. This expands our capabilities to the full generality of C/C++.

7.1 Threading Libraries

Multithreaded programming is more than just shared memory. Operations such as thread creation, thread destruction, and mutex locking are equally important and must also happen deterministically. COREDET includes a deterministic implementation of the pthreads library. In this section we highlight key operations provided by that library.

7.1.1 *Thread Create, Exit, and Join*

Thread creation and destruction are sufficiently rare that we simply delay them until serial mode. Spawned threads are appended to the serial round-robin order immediately but do not begin executing until parallel mode of the following quantum. To exit, a thread waits for serial mode, sets an exit flag, and then exits. Thread T_0 joins on T_1 by waiting for T_1 to set its exit flag. Since the exit flag is changed only in serial mode, a join can safely complete in parallel mode.

7.1.2 *Synchronization Objects*

Synchronization objects such as mutexes, barriers, and condition variables can be implemented on top of an atomic compare-and-swap (CAS) subroutine. To avoid uncontrolled interaction with the OS scheduler, we do not invoke the system pthreads library. Instead, our synchronization objects are implemented using well known busy-wait algorithms (as in [23]) with a few modifications to ensure determinism:

DMP-O: Each operation on a synchronization object O requires exclusive ownership of O in order to perform a CAS. For pre-

[2] More generally, if an access is not involved in a data race it does not need to be instrumented, but our current infrastructure does not include a sound static race detector.

cision we use a distributed object-based MOT for synchronization objects, which means that the MOT entry for O is held in a field in O. Our MOT policy (Section 3.1) ensures that if synchronization is mostly thread-local, serialization will rarely be necessary.

DMP-B: All synchronization operations must execute in serial mode as a consequence of the following two observations: (1) Each operation requires a memory fence because execution is weakly-consistent; and (2) CAS cannot execute in parallel mode (Section 4).

7.2 External Libraries

So far we have assumed instrumentation of the whole program. However, the whole program is rarely available. C and C++ programs commonly load shared libraries, such as the system `libc`, at runtime. External libraries can execute arbitrary code that may cause interthread communication, and it is important to execute that code deterministically. We support external libraries with the following extension: Each call to an external library must execute in serial mode. Our compiler ensures this by adding instrumentation to every external function call.

Three optimizations prevent over-serialization. First, we handle indirect calls precisely. If we cannot determine all possible targets of an indirect call statically, then at runtime we compare the indirect target against a hash table of known functions. The call is serialized only if it is truly external. Second, we provide deterministic replacements for commonly used `libc` functions such as `memcpy`. Third, we do not serialize pure functions, such as those in `math.h`, since they do not access shared-memory.

We must mention two caveats. First, an external library can deadlock if it uses blocking synchronization, such as through a pipe. This is a consequence of serialization, and it is one of the reasons we provide our own implementation of the pthreads library. Second, if an external library registers an asynchronous callback to external code, we do not guarantee deterministic execution of that code since we do not control when it is scheduled.

7.3 Memory Allocation

Heap objects must be allocated at a deterministic address since addresses are visible to C. The simple solution is to treat `malloc` as an external library function, but that risks over-serializing programs with frequent heap allocation. A better solution is to compile the memory allocator using COREDET. We assume the memory allocator is data race free and observe that for data race free programs, deterministic synchronization is sufficient for full determinism [27, 30]. This means we do not need to instrument loads and stores in the allocator; we simply link the allocator with the deterministic threading library described in Section 7.1.

We selected the Hoard allocator [3] because of its high scalability, which is achieved in part by the use of a thread-local cache. This thread-local cache provides an additional benefit for DMP-O: it minimizes the number of times that an address is freed by one thread and then allocated to another thread, which reduces false-sharing in the MOT and further improves scalability.

7.4 Memory Errors

We end this section with a caveat about memory errors. C and C++ are not safe languages. COREDET is robust to most kinds of memory errors. However, in two corner cases, a memory error can invalidate our deterministic guarantee. We stress that COREDET always executes a program deterministically up to the first such error. The corner cases are memory errors that lead to unexpected accesses to: (1) some object that was labeled thread-local by our escape analysis; and (2) COREDET's runtime metadata. (Our runtime executes in the same address space as the program.) In both cases, the program is performing one of two fundamentally unsafe operations, neither of which are supported by the underlying language: making assumptions about memory layout, or writing to invalid pointers.

8. Evaluation

Our main goal in this evaluation is to show that COREDET provides determinism in arbitrary C/C++ programs without sacrificing scalability. We start by describing our experimental infrastructure, then show scalability and performance data, and end with a characterization analysis of optimizations and the configuration parameters.

8.1 Experimental Setup

We ran our experiments on an 8-core 2.4GHz Intel Xeon E5462 with 10GB of RAM, and used 64-bit Ubuntu 8.10 with address space randomization disabled. We present results that are the average of 10 runs with the highest and lowest values removed. Error bars indicate the 95% confidence interval, as even though COREDET ensures the same program outcome on identical runs, performance can still vary.

We evaluated our system on two benchmark suites: the SPLASH2 suite of parallel scientific workloads, and the PARSEC suite of more general purpose parallel programs. We used the `simlarge` inputs for PARSEC (except for the `blackscholes` and `canneal` benchmarks, where we used the `native` input to overcome short running times), and scaled inputs for SPLASH2 to run for about a minute with one thread. There were several benchmarks from these suites that we were not able to run using our infrastructure. In SPLASH2, `volrend` and `radiosity` were not 64-bit clean and we opted not to include results from `raytrace` and `cholesky` because, even with their largest inputs, they ran for less than half a second on our test machine. A few benchmarks from PARSEC use language or runtime features that our prototype implementation does not yet support: `dedup` suffered from miscompilations by LLVM; `bodytrack`, `facesim` and `ferret` use C++ exceptions; `freqmine` uses OpenMP; and `vips` was incompatible with our build infrastructure. We compiled all the benchmarks using a pre-release version of LLVM 2.6 with `-O4` optimizations, including link-time optimization. We tested the correctness of COREDET by running the `racey` deterministic stress test [15, 34] 10,000 times for each COREDET configuration. We verified that `racey` consistently produced the same output for each configuration.

Table 1 lists the benchmarks we used to evaluate our work, with a short description of each and the percentage of work (of a single-threaded execution) that is *not* designed to execute in parallel. The top section refers to SPLASH2 applications and the bottom section refers to PARSEC applications. As these benchmarks are all highly parallel, they are good choices for evaluating how COREDET impacts scalability.

8.2 Scalability and Overheads

In this section we compare the scalability and overheads of COREDET with NONDET, which is a nondeterministic executable that is compiled solely with LLVM (without the COREDET compiler transformations and runtime). For each COREDET configuration we turn on escape analysis, redundant call removal (DMP-O only), and heuristic quantum balancing, and we use optimal values for the *granularity* and *quantum size* parameters as discussed in Section 8.3.

We start by looking at the performance scalability for each configuration, shown in Figure 6. Each bar is normalized to the same configuration with 2 threads, so for example, `fmm` running with DMP-B runs 1.82x faster with 8 threads than with 2 threads. The subset of benchmarks shown in this plot includes the best-scaling (`lu`) and worst-scaling (`canneal`) benchmarks from each

			(Section 8.3) Optimal Configuration			(Section 8.2) Moore's Dividends			
Benchmark	Description	% Sequential	O	B	PB	ND	O	B	PB
barnes	n-body simulation	< 1%	16B,10k	64B,50k	64B,50k	3.55	.61	.69	**.70**
fft	fast Fourier transform	31%	4096B,200k	64B,50k	64B,50k*	1.41	**.70**	.51	.55
fmm	n-body simulation	1%	1024B,50k	64B,50k	64B,50k	2.84	**.61**	.59	.54
lu	matrix factorization	< 1%	4096B,200k	64B,200k*	64B,100k	3.21	**.70**	.28	.25
ocean	ocean simulation	< 1%	4096B,10k	64B,50k	64B,100k	1.29	.76	.82	**1.05**
radix	radix sort	< 1%	4096B,10k	64B,50k	64B,100k	3.49	**1.22**	.95	1.04
water	molecular dynamics	< 1%	16B,200k	64B,200k*	16B,200k	2.23	**.71**	.32	.50
blackscholes	options pricing	< 1%	16B,200k*	64B,100k	64B,50k	3.16	**2.75**	2.68	2.66
canneal	chip routing	17%	16B,10k	64B,10k	64B,10k	1.54	**.54**	.45	.37
fluidanimate	fluid simulation	6%	64B,50k	64B,50k	64B,50k	1.97	**.52**	.41	.36
streamcluster	online data clustering	< 1%	16B,100k*	64B,200k*	64B,200k*	2.18	**1.11**	.79	.58
swaptions	swaptions pricing	< 1%	16B,200k	64B,50k	64B,50k	3.47	**1.59**	1.39	1.30
x264	H.264 video encoding	< 1%	1024B,100k*	64B,200k	64B,100k*	2.63	**.41**	.27	.26

Table 1. Benchmarks

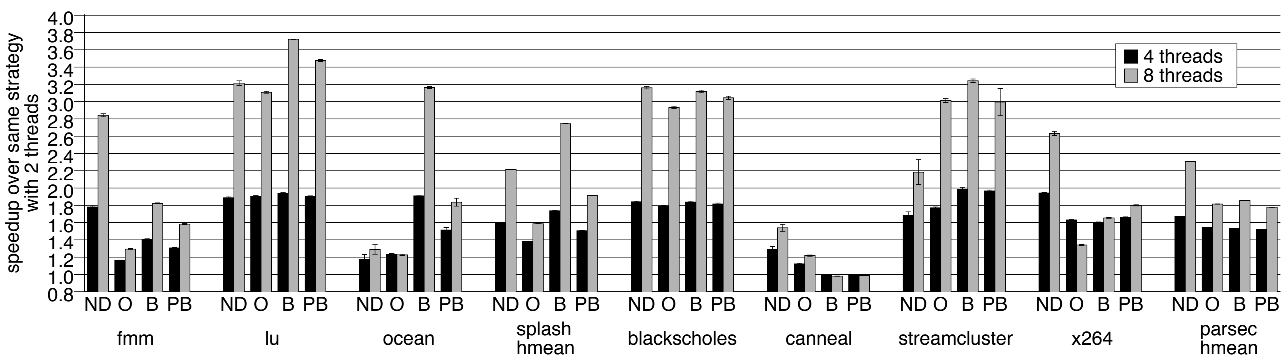

Figure 6. Scalability

of the SPLASH2 and PARSEC suites, but the harmonic mean is for all benchmarks in each suite as listed in Table 1. Overall we note that DMP-O limits scalability more than other schemes, but it still enables reasonable scaling. DMP-B generally offers the best scalability, often very close and sometimes exceeding NONDET, with a few exceptions (`fmm`, `canneal`, `x264`). Our most important goal was to avoid limiting scalability; we believe that goal has been achieved.

Figure 7 plots the performance results in a different way: Each bar shows COREDET normalized to NONDET with the same number of threads, directly showing how much overhead is incurred in each configuration. Again, we show the best and worst performers from each benchmark suite, and the means across all benchmarks in each suite. Overall, the overheads for 8 threads range from 1.1x–6x for DMP-O and 1.2x–11x for DMP-B.

Figure 7 also shows scalability of COREDET with respect to NONDET's scalability. When all bars in a group have exactly the same height (`lu`), then COREDET is scaling exactly as well as NONDET. When the bars become shorter as more threads are added (`fmm`), then COREDET is scaling more slowly than NONDET. When the bars grow as more threads are added (`ocean`), then COREDET is scaling more rapidly than NONDET. This last result is surprising, but can happen because COREDET has higher overheads than NONDET and thus more inherent parallelism; effectively, we can gain back lost overhead.

Note that Figure 7 clearly demonstrates the scalability vs. overhead tradeoff: DMP-B has higher overheads but better scalability, while DMP-O has lower overheads but worse scalability, and DMP-PB fits somewhere in between. DMP-B's distinct scalability advantage derives from its relaxed memory ordering and the fact that fences and atomic operations are relatively rare; when atomic operations are not rare (`canneal`), DMP-B's scalability suffers. Achieving similar scalability in DMP-O would require evolving the state of the MOT in a proactive and intelligent way, such that threads are very likely to own the memory locations they are updating. It can be quite difficult to discover such a sharing policy for applications with complicated sharing patterns, hence DMP-O's scalability suffers. DMP-B's scalability comes at a cost, however, as the overheads of maintaining store buffers are significantly higher than the overheads of maintaining the MOT.

Spending Moore's Dividends. We believe determinism is worth some performance loss, and since in many cases this cost is not in scalability, we can potentially pay for it with extra cores. The rightmost four columns in Table 1 show the speedup of NONDET and COREDET with 8 threads compared to NONDET with 2 threads. In five of the benchmarks we used, COREDET matched or was faster than NONDET with 2 threads, demonstrating that in some applications we can trade 4x cores for determinism. The average speedup compared to NONDET with 2 threads is 2.26x for NONDET and 0.75x for the best deterministic scheme (in bold). Hence a user upgrading from 2 to 8 cores can often choose between increased speed and determinism.

8.3 Sensitivity

The performance of COREDET is very sensitive to three parameters: the granularity (of the MOT for DMP-O, of store buffer chunks for DMP-B, and of both for DMP-PB); the quantum size; and the choice between full serial mode and reduced serial mode (Section 3.2). We measured the performance of each benchmark under a variety of parameter values. Table 1 shows the values we used

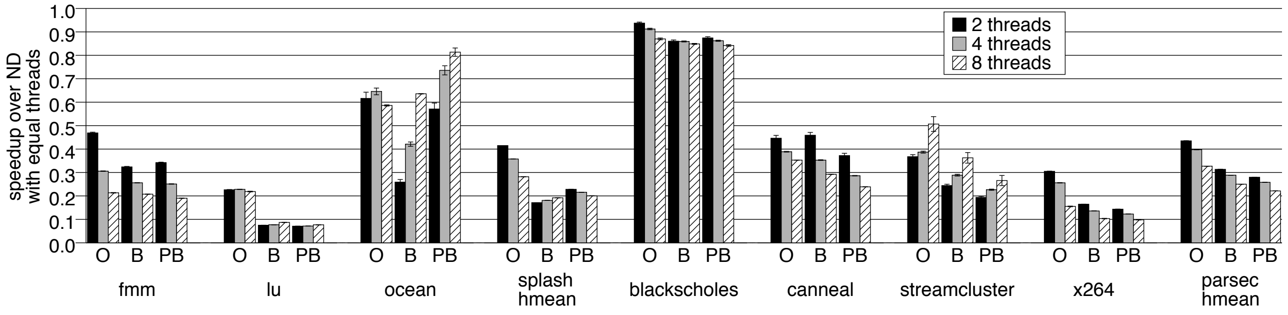

Figure 7. Overheads relative to ND with the same number of threads

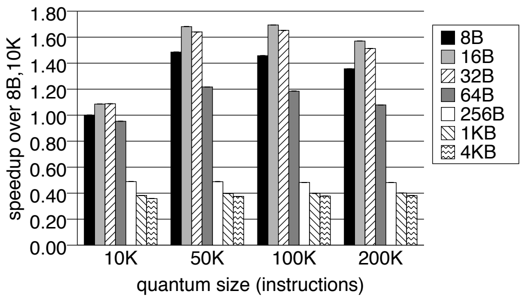

Figure 8. `streamcluster` sensitivity for DMP-O

for each benchmark in this study. In Table 1, quantum size has units of (approximately) one simple `x86` instruction, and reduced serial mode (as opposed to full serial mode) is designated by an asterisk. The variability across benchmarks shows that most benchmarks have a "sweet spot" unique to them. One visible trend is that DMP-B prefers 64B store buffer chunks, likely due to the superior space utilization compared to smaller chunks (due to specifics of our implementation we could not test chunk sizes larger than 64B).

Figure 8 examines in detail the performance of `streamcluster` executing under DMP-O with a variety of granularities and quantum sizes. For DMP-O, choosing too fine a granularity reduces the opportunity for COREDET to exploit spatial locality by "prefetching" permissions for large contiguous blocks of data. Choosing too coarse a granularity leads to false-sharing as threads try to access disjoint portions of memory mapped to a single entry in the MOT. For `streamcluster`, the latter effect dominates: tracking at too large a grain costs up to a factor of four in performance. The optimal MOT granularity for `streamcluster` is just 16B.

In addition, the choice of quantum size has a large impact on performance. Larger quanta better amortize the costs of round barriers, but some applications share data with such frequency that smaller quanta perform much better, as smaller quanta reduce the latency between when a producer and its consumer are able to run. `streamcluster` is a good example of this tradeoff. At small granularities (10k) the overheads dominate, while large granularities (200k) suffer from increased communication latency.

Finally, `streamcluster` runs best with reduced serial mode (a 20% improvement over full serial mode at the optimal configuration of 16B and 100k). Generally, we found reduced serial mode helps the most for large quantum sizes, while it occasionally hurts performance for small quantum sizes.

8.4 Characterization

Table 2 characterizes COREDET in a few ways. The data in this table was generated from executions using 8 threads. Columns 2–4 show the percentage of total execution time spent in serial mode. These numbers are highly correlated with scalability. For example, `fmm` and `canneal` both spend a relatively large percentage of time in serial mode and so have the poorest scalability, as also seen in Figure 7. Similarly, `lu` and `blackscholes` spend a small percentage of time in serial mode and have good scalability. Furthermore, these numbers show clearly that DMP-B has the best overall scalability, while DMP-O has the worst.

Columns 5 and 6 examine how much faster each benchmark runs with static optimizations turned on. Generally, static optimizations help DMP-O more often than DMP-B because redundant call removal is applicable more often than escape analysis. (Many of our scientific benchmarks make heavy use of global arrays which are not amenable to escape analysis.) However, when it does apply, escape analysis helps DMP-B more than DMP-O due to the intrinsically higher overheads of DMP-B; for example, escape analysis decreases the runtime of `swaptions` by 59% for DMP-B.

Column 7 examines how much faster DMP-B runs with parallel commit versus with serial commit, showing that our parallel commit algorithm (Section 4.2) does indeed recover parallelism. Column 8 compares parallel commit to simply committing all store buffers in parallel nondeterministically; this represents the ceiling of parallel commit performance. We observe that parallel commit is generally very close to that ceiling, with one exception (`fft`).

Columns 9 and 10 examine how much faster DMP-B runs with heuristic balance versus with maximal balance and minimal balance, respectively. We consider heuristic balance a good tradeoff as it is only rarely slower than either alternative (at most -5.2%) and often faster (up to 59%). These columns also demonstrate the importance of balance, as minimal balance is generally the worst performer.

Comparison to DMP-TM. The last column approximates the percentage of quanta suffering conflicts in a DMP-TM-like system, showing simulation results for the benchmarks reported in [11]. These results are from an HTM simulation, with 1,000-instruction quanta and eager conflict detection at a 32B granularity. A realistic STM implementation would require much larger quanta to amortize the overheads of quantum boundaries, which would cause conflict rates to climb even higher. As the rate of transactional conflicts is proportional to the amount of extra serialization introduced by TM, we observe that for most benchmarks the rate of conflicts is much higher than DMP-B's serial work (Column 3), justifying our claim that DMP-B can attain the benefits of DMP-TM without relying on speculation.

Benchmark	Serial Mode (% of execution time) O	B	PB	Static Opts (% faster) O	B	Parallel Commit (% faster vs.) Serial Commit	Ideal Commit	Heuristic Balance (% faster vs.) Maximal Balance	Minimal Balance	TM % conflicts (from [11])
barnes	56.2	8.8	36.9	6.6	0.9	1.8	−4.4	1.4	2.2	37
fft	62.6	6.5	47.9	10.7	0.0	23.9	−15.8	−4.1	3.2	25
fmm	68.3	23.6	36.9	14.6	16.8	7.3	−4.7	4.4	14.5	51
lu	22.4	2.1	6.7	24.2	0.7	1.4	−0.5	0.0	6.9	71
ocean	40.0	4.6	7.3	1.2	0.0	19.5	−5.5	0.2	14.6	28
radix	55.0	3.2	57.9	1.4	3.1	16.8	−8.3	−0.8	1.1	7
water	20.3	2.2	5.8	23.7	13.6	−0.6	−2.5	1.4	2.4	19
blackscholes	12.9	3.9	6.1	0.0	0.0	2.2	1.1	1.1	−1.1	8
canneal	70.2	96.1	98.1	0.9	0.0	0.2	−1.1	3.2	−1.6	–
fluidanimate	65.1	49.2	67.3	1.6	0.0	5.8	−0.6	0.3	−5.2	76
streamcluster	24.8	8.0	11.7	4.0	0.0	3.8	−1.0	0.3	2.6	28
swaptions	10.2	11.1	10.3	14.8	59.2	9.6	−4.2	0.8	59.7	–
x264	53.0	30.4	21.5	9.9	12.3	1.4	−1.4	0.4	−3.6	–

Table 2. Characterization

9. Related Work

Since determinism is a desirable property for parallel programs, there has been significant effort in dealing with nondeterminism. Past work focused on deterministic parallel languages, log-based deterministic replay, and testing / debugging tools. In contrast, our work focuses on providing determinism in arbitrary multithreaded programs.

There are a few recent proposals for deterministic multithreaded execution. Kendo [27] provides deterministic execution for race-free programs by imposing a deterministic order of synchronization operations. It uses the number of executed store instructions (collected via performance counters) as logical time to deterministically schedule synchronization operations. Since Kendo does not instrument loads and stores, it leads to low overhead, albeit at the cost of not automatically supporting arbitrary programs. Grace [4] is a runtime system for C/C++ programs with fork-join parallelism that executes each thread in a fork region atomically and commits them in a deterministic order; the goal is to increase safety of fork-join programs by making them behave like their sequential counterparts. Grace uses page-protection hardware and operating system processes to implement the transactional mechanism, which distinguishes it from traditional STMs.

DMP [11] is a recent proposal for a multiprocessor architecture that executes arbitrary multithreaded programs deterministically. DMP introduced the DMP-TM and DMP-O (there called DMP-ShTab) deterministic execution strategies. CoreDet is a major contribution on top of DMP. Besides all the novel compiler and runtime techniques to make determinism usable without hardware support, we propose a new execution strategy (DMP-B) and show how relaxed memory orderings can be exploited to achieve scalability comparable to DMP-TM without requiring speculation.

Stream-based programming languages, such as StreamIt [32], are deterministic because communication happens only via explicitly declared streams. Similarly, SHIM [14] is a parallel language that supports explicit communication only via deterministic message passing. These are a good match for digital signal processing (DSP) applications. NESL [6] and Data Parallel Haskell [10] are examples of data-parallel functional languages that expose sequential semantics to programmers and yield deterministic programs. Another notable class of deterministic languages are implicitly parallel languages, such as Jade [29]. With Jade, programmers write programs in a sequential, imperative language and then augment the code with information about how data is accessed. The system then extracts concurrency without violating the original sequential program semantics. More recently, Bocchinno et al., developed Deterministic Parallel Java (DPJ) [7], which is a set of extensions to Java that enable programmers to control precisely where nondeterminism is allowed in the code via a type and effect system.

While using deterministic languages is a long-term solution to the problem, the existing options are typically domain-specific and are not widely used. The majority of parallel programs being written today use mainstream languages such as C/C++ or Java, and this is likely to remain the case for the time being. Past research has dealt with nondeterminism in mainstream languages in the context of specific needs, such as log-based deterministic replay, data-race detection tools, and testing frameworks.

In log-based deterministic replay systems, a recording tool monitors the execution of a program and logs the effects of nondeterminism. A replay tool then consumes the log to reproduce the recorded behavior. DeJavu [17] is a software-only record and replay system for race-free Java programs. DeJavu records only synchronization events and enforces the recorded synchronization during replay. RecPlay [30] also records synchronization order but it includes a data-race detector as well. As it provides deterministic replay only up until the first data-race, RecPlay uses race detection to alert the user when the first race occurs. CHESS [25] is a testing tool that explores multiple synchronization orderings in order to test multithreaded programs and performs race detection for each choice. When it detects a defect, it lets the user replay the execution that led to a race.

To support arbitrary programs that might contain data-races or lock-free data-structures, a record and replay system needs to record much higher-frequency information to capture the outcome of any conflicting memory operation. This can generate very large logs because every memory operation that is subject to nondeterminism needs to be logged. Therefore, proposals for general-purpose fully deterministic replay system typically involve hardware support. Among such proposals are Instant Replay [20], Flight Data Recorder [34], and most recently DeLorean [24] and ReRun [16]. ReRun is related to CoreDet because it relies on the observation that threads do not communicate all the time. DeLorean is also related to CoreDet because it uses the notion of *chunks* (a quantum in CoreDet) to record execution at a coarse grain. In addition, one of its modes forces a specific order of events during replay to reduce log size. While CoreDet shares some of the same observations as the work just described, CoreDet is not a record and replay system and does not rely on hardware support.

The Wisconsin Wind Tunnel (WWT) [28] performs a deterministic simulation of a parallel computer. It works by dividing the program into quanta, which are executed in lock-step across all processors. The quanta provide deterministic state transitions for the simulated coherency protocol. WWT's notions of quanta and ownership-based coherency have direct analogues in DMP-O.

Other systems, unrelated to determinism, use techniques similar to ones employed by COREDET. Store buffering, as used by DMP-B, has been used before to efficiently support relaxed memory models [12, 13]. Escape, redundancy, and batching optimizations have previously been used to remove instrumentation overheads in various settings [1, 31]. Mellor-Crummey and LeBlanc describe software instruction counters that are similar to our maximal and minimal quantum balancing schemes [22].

10. Conclusions

In this paper we presented COREDET, a fully automatic compiler and runtime system for deterministic execution of arbitrary C/C++ multithreaded programs. We explored two basic approaches to enforcing determinism. The first tracks ownership of data and serializes execution whenever threads communicate. This yields sequentially consistent executions and has lower overheads, but lower scalability. The second approach uses memory versioning without any form of speculation and relaxes memory ordering, yielding higher scalability at the cost of higher overheads. We described a number of static and dynamic optimizations and evaluated the trade-offs of both approaches in detail. Our results indicate it is possible to execute arbitrary multithreaded programs deterministically without sacrificing scalability and in most cases our system can compensate for its overheads with more cores. We believe our present work is an important starting point for scalable deterministic compilers and runtime systems.

The source code for COREDET will be made available at `http://sampa.cs.washington.edu/`.

11. Acknowledgements

We thank the anonymous reviewers for their helpful feedback. We also thank the members of the Sampa group for their feedback on the manuscript. We thank Mark Oskin for his early work on some of the compiler code. This work was supported in part by NSF CAREER grant CCF-0846004 and a Microsoft Research Faculty Fellowship. Tom Bergan was partially supported by a Corin Anderson fellowship.

References

[1] A. Adl-Tabatabai, B. Lewis, V. Menon, B. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI*, 2006.

[2] S. Adve and M. Hill. Weak Ordering – A New Definition. In *ISCA*, 1990.

[3] E. Berger, K. McKinley, R. Blumofe, and P.Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *ASPLOS*, 2000.

[4] E. Berger, T. Yang, T. Liu, , and G. Novark. Grace: Safe and Efficient Concurrent Programming. In *OOPSLA*, 2009.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[6] G. Blelloch. NESL: A Nested Data-Parallel Language (Version 3.1). Technical report, Carnegie Mellon University, Pittsburgh, PA.

[7] R. Bocchino, V. 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*, 2009.

[8] H.-J. Boehm. Threads Cannot be Implemented as a Library. In *PLDI*, 2005.

[9] H.-J. Boehm and S. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, 2008.

[10] M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A Status Report. In *Workshop on Declarative Aspects of Multicore Programming (DAMP)*, 2007.

[11] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS*, 2009.

[12] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *ISCA*, 1986.

[13] M. Dubois, J. Wang, L. Barroso, K. Lee, and Y.S. Chen. Delayed Consistency and Its Effects on the Miss Rate of Parallel Programs. In *Supercomputing*, 1991.

[14] S. A. Edwards and O. Tardieu. SHIM: A Deterministic Model for Heterogeneous Embedded Systems. In *EMSOFT*, 2005.

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

[16] D. Hower and M. Hill. ReRun: Exploiting Episodes for Lightweight Memory Race Recording. In *ISCA*, 2008.

[17] J. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *SIGMETRICS SPDT*, 1998.

[18] C. Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005.

[19] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

[20] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE TOC*, 36(4), 1987.

[21] J. Manson, W. Pugh, and S. Adve. The Java Memory Model. In *POPL*, 2005.

[22] J. Mellor-Crummey and T. J. LeBlanc. A software instruction counter. In *SIGARCH Computer Architecture Notes*, 1989.

[23] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM TOCS*, 9(1), 1991.

[24] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *ISCA*, 2008.

[25] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*, 2008.

[26] S. Narayanasamy, C. Pereira, and B. Calder. Recording Shared Memory Dependencies Using Strata. In *ASPLOS*, 2006.

[27] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, 2009.

[28] S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis, and D. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. *SIGMETRICS*, 21(1), 1993.

[29] M. Rinard and M. Lam. The Design, Implementation, and Evaluation of Jade. *ACM TOPLAS*, 20(3), 1988.

[30] M. Ronsse and K. De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM TOCS*, 17(2), 1999.

[31] D. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A Low Overhead, Software-only Approach for Supporting Fine-grain Shared Memory. In *ASPLOS*, 1996.

[32] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *CC*, 2002.

[33] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

[34] M. Xu, R. Bodik, and M. Hill. A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay. In *ISCA*, 2003.