Lazy Determinism for Faster Deterministic Multithreading

Timothy Merrifield
VMware Inc.
timothym@vmware.com

Sepideh Roghanchi
University of Illinois at Chicago
srogha2@uic.edu

Joseph Devietti
University of Pennsylvania
devietti@cis.upenn.edu

Jakob Eriksson
University of Illinois at Chicago
jakob@uic.edu

Abstract

Deterministic multithreading (DMT) fundamentally requires total, deterministic ordering of synchronization operations on each synchronization variable, i.e. a partial ordering over all synchronization operations. In practice, prior DMT systems totally order all synchronization operations, regardless of synchronization variable; the result is severe performance degradation for highly concurrent applications usingfine-grained synchronization.

Motivated by this class of programs, we propose *lazy determinism* as a way to go beyond this total order bottleneck. Lazy determinism executes synchronization operations speculatively, and enforces determinism by subsequently validating the resulting order of operations. If an ordering violation is detected, part of the computation is restarted. By enforcing only the partial ordering required to guarantee determinism, lazy determinism increases the available parallelism during deterministic execution.

We implement LazyDet via a pure-software runtime system accelerated by custom Linux kernel support. Our experiments with hash table benchmarks from Synchrobench show roughly an order of magnitude improvement in the performance of lock-based data structures compared to the state of the art in eager determinism. For benchmarks from PARSEC-2, SPLASH-2, and Phoenix, we demonstrate runtime improvements of up to 2× on the programs that challenge deterministic execution environments the most.

CCS Concepts • **Software and its engineering → Multithreading**; *Concurrency control*.

ASPLOS '19, April 13–17, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304047

Keywords determinism, multi-threading, speculative execution, performance

ACM Reference Format:
Timothy Merrifield, Sepideh Roghanchi, Joseph Devietti, and Jakob Eriksson. 2019. Lazy Determinism for Faster Deterministic Multithreading. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3297858.3304047

1 Introduction

Many of today's computer systems are inherently nondeterministic. Due to timing effects, the order of operations in concurrent programs is not well defined. This means that even for a bug-free program, the output of the program may vary between executions, given identical input. Deterministic Multithreading (DMT) systems promise significantly reduced complexity for programmers writing multithreaded code. With shared memory communication and synchronization "set in stone" for a given input, the programmer is much better positioned to reproduce reported bugs, and reason about a multithreaded program's execution.

These benefits come at a performance cost, however, particularly for synchronization-intensive programs. In all existing DMT systems, both those that determinize the outcome of data races (*strong determinism* schemes) [4, 15, 19, 26, 27, 30, 32] and weak determinism schemes that ignore races [33], deterministic ordering is eagerly enforced. As a result, synchronization operations follow a deterministic *total order* regardless of variable. This introduces global coordination at every synchronization operation, even for logically independent operations such as acquiring distinct locks.

Our system, LazyDet, is thefi rst to provide *lazy determinism*, breaking through the total order bottleneck and enforcing a more scalable, partial ordering of synchronization operations. Lazy determinism executes synchronization operations speculatively, and enforces determinism by subsequently validating the resulting order of operations. If a deterministic ordering violation is detected, part of the computation is restarted. By verifying the ordering after the fact, LazyDet knows which synchronization variables were used,

and when, allowing it to avoid the *total order* of eager determinism. LazyDet preserves determinism during speculative execution by making deterministic decisions about when to initiate and terminate speculation, and by detecting and resolving conflicts deterministically.

LazyDet is thefi rst software DMT system to employ speculation, which has been conventionally deemed too expensive to be practical [5]. Our insight with LazyDet is that strong determinism enables efficient speculation, especially through cheap conflict detection performed only on locks, not memory locations as with software transactional memory (STM). While some of the existing literature on determinism has dispensed with strong determinism entirely [33] ostensibly for performance reasons, LazyDet surprisingly shows that by enabling speculative execution, strong determinism can in some cases outperform its weak counterpart.

The primary contributions of this paper are as follows:

- We describe thefi rst fully general DMT system that removes the total order synchronization bottleneck.
- Wefi nd that the thread isolation that underlies strong determinism is also a key enabler of lazy determinism. This allows our strong DMT system to match and even outperform a weak DMT system for thefi rst time.
- We detail a high-performance, software implementation of LazyDet that runs on commodity multicores.
- We apply LazyDet to a challenging new set of synchronization-intensive workloads, where LazyDet performs ≈ 10× better than existing systems.

Below, we give a brief background and motivation for lazy determinism in §2, followed by a system description in §3. We offer additional context for the relaxed memory consistency model provided by LazyDet in §4, and a detailed performance evaluation in §5. Finally, we discuss related work in §6, discuss limitations in §7, and offer our conclusions in §8.

2 Background and Motivation

The key to deterministic multithreading is either restricting or deterministically ordering all "communication" between concurrent threads of execution, while still providing a usable memory model for multi-threaded programming. Here, communication consists of any reads of values previously written by other threads. Communication may be restricted through thread isolation techniques (in the case of *strong determinism*), and the order of operations is regulated by a *deterministic logical clock* (DLC).

LazyDet adds lazy determinism to Consequence [30], an existing deterministic execution environment. We briefly review the operation of Consequence below. Consequence uses the virtual memory subsystem to provide thread-level isolation—each thread maintains its own page table, and any modifications are kept strictly local until explicit communication in the form of `commit` and `update` operations are performed against the central "version list" (see Conversion [31]). Changes to local memory are communicated in this fashion only as a result of synchronization operations, which are eagerly ordered by an instruction-counter based DLC. The thread that has the lowest logical clock value (intuitively: arrivesfi rst, in deterministic logical time) is always next in line to execute a synchronization operation. Note that operations other than synchronization execute independently—only synchronization operations are ordered by the DLC.

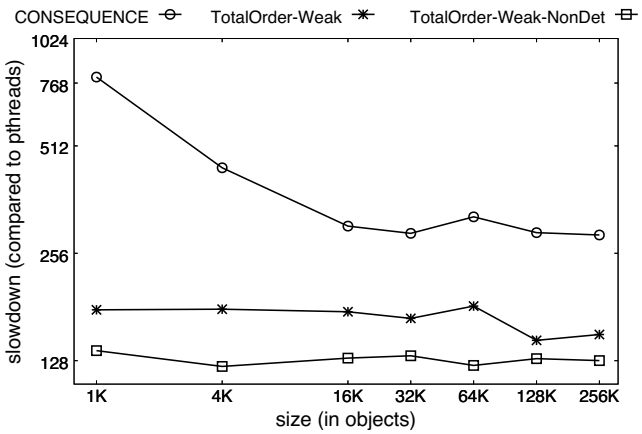


Figure 1. The results of a hash table experiment running under Consequence, and two eager variants.

Fundamentally, deterministic multithreading requires synchronization operations *on the same variable* [28] to occur in order of increasing DLC values. Like Consequence, all prior work on deterministic multithreading has enforced this constraint eagerly, allowing a thread to perform a synchronization operation only when it has the globally minimum DLC value. This, however, results in a total ordering across operations on *all* synchronization variables.

In general, it is not possible to predict future synchronization operations by other threads. Thus, any eager solution must totally order all synchronization operations in order to maintain determinism.

2.1 The Cost of Totally Ordered Synchronization

While the state-of-the-art eager DMT system Consequence [30] achieves good performance on benchmarks that use primarily coarse-grained synchronization, wefi nd that the total ordering of synchronization operations results in tremendous performance degradation onfi ne-grained synchronization workloads. Figure 1 shows the performance of a hash table data structure from Synchrobench [21], running on top of Consequence, in terms of slow-down versus normal pthreads execution. The x-axis is the maximum number of objects inserted into the hash table for a given experiment. In this example, deterministic execution with Consequence imposes a factor of ≈ 300× slowdown versus nondeterministic execution with pthreads. Figure 1 also

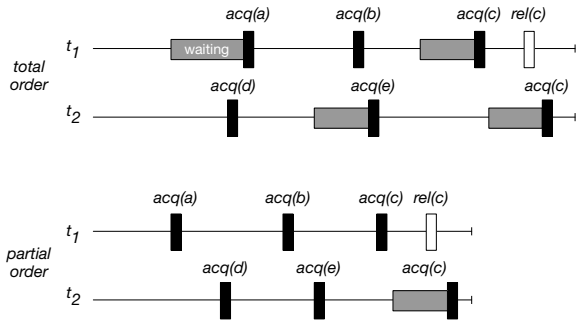


Figure 2. An example execution of a synchronization-heavy workload with (top) Kendo-style totally ordered synchronization [33] and (bottom) LazyDet's partial order determinism.

shows two alternative schemes that provide weaker guarantees. Here, Consequence-Weak disables strong determinism, and Consequence-Weak-Nondet uses a nondeterministic DLC (with no determinism guarantees). However, both Consequence-Weak and Consequence-Weak-Nondet still enforce a total order on all synchronization operations. This total order is expensive: even under Consequence-Weak-Nondet, which enforces a total but non-deterministic order (see §5 for details), the hash table is 128× slower than under normal, un-ordered operation.

2.2 The Performance Potential of Lazy Determinism

However, we observe that much of the time, ordering of synchronization operations on individual variables, a *partial order*, is deterministic even without enforcement. This suggests a lazy solution to enforcing deterministic ordering.

LazyDet demonstrates, for thefi rst time, that totally ordered synchronization is not necessary for determinism. Instead, LazyDet uses lazy determinism enforcement to impose only a partial order on synchronization. This avoids the expensive coordination that prior schemes must necessarily perform on every synchronization operation.

Figure 2 illustrates how avoiding such coordination can result in lower latency when synchronization is frequent. In the top half of thefi gure, totally ordered synchronization requires thread t_1 to wait to acquire lock a because it does not have the global minimum DLC. However, with partial ordering, threads can acquire distinct locks without coordination with others. Only t_2's acquisition of lock c incurs any waiting, as c is currently held by t_1.

2.3 Relationship between Strong, and Lazy, Determinism

Both strong determinism and lazy determinism require thread isolation in order to provide their determinism guarantees. In the case of strong determinism, thread isolation delays inter-thread visibility of writes, to ensure determinism in the face of race conditions. In the case of lazy determinism, thread isolation provides the same effect, but to enable speculative lock acquisition.

In principle, one could also design a lazy, weak deterministic execution runtime by enabling thread isolation only when executing speculatively. This alternative design falls outside the scope of this paper.

3 System Description

Below, we describe the design of LazyDet, thefi rst DMT system that breaks the total synchronization order barrier. As described in 2, LazyDet builds on Consequence. However, LazyDet avoids the total synchronization order bottleneck in Consequence, by executing lock acquisitions speculatively. When LazyDet encounters a lock acquisition in user code, it must decide whether to acquire the lock conventionally (which implies waiting for the current thread to have the minimum DLC, committing local changes and updating changes by others, and entering a critical section), *or* to acquire the lock speculatively and continue execution without delay, and without publishing its changes to local memory. System calls during speculative execution require special handling, see §3.5.

Acquiring a lock speculatively results in a new memory consistency model, a variant of a Data-Race-Free memory model [1] that we call *Deterministic Data-Race-Free* (DDRF)[1]. In essence, DDRF says that changes *must* be visible between threads that touch the same synchronization variable, and *may* be visible at any other time, as long as such visibility is provided in a deterministic fashion. In §4, we describe the DDRF memory consistency model more formally, and compare it to several models from related work.

3.1 Speculative Order Elision

Figure 3 illustrates the speculative operation of LazyDet. Speculation always begins at a lock acquisition.[2] If thread i makes a decision to speculate, it takes a snapshot of its current state to support roll-back (see §3.3), and the current value of the DLC is stored in $BEGIN_i$. §3.4 describes how the speculation decision is made in more detail. Speculative execution then proceeds through one or more lock acquisitions where each lock acquisition triggers a decision about whether to continue speculating, to terminate speculation, or to upgrade the speculation run (see §3.5). If the decision is to continue, speculative lock acquisition merely results in a record in a thread-local log, L_i, for thread i indicating that the lock was used—no coordination with other threads happens yet. The same is true for release—the record is updated

[1]For readers unfamiliar with the term data-race-free, it implies that memory consistency guarantees are only provided for data race free programs. DDRF additionally provides determinism guarantees even for racy programs.
[2]In LazyDet, condition variable operations and barriers, as they necessitate inter-thread communication, cause speculation to commit if possible or rollback otherwise.

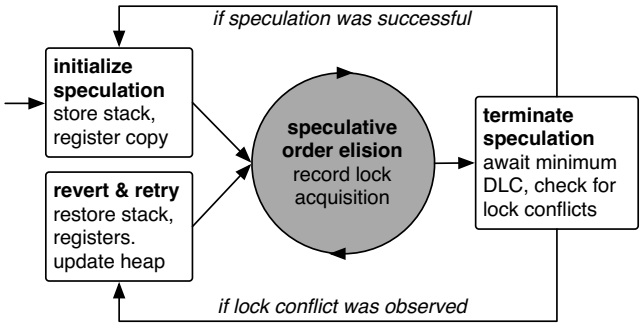


Figure 3. Action diagram for LazyDet. Speculation begins at a lock acquisition, and may proceed through several, until a deterministic decision is made to terminate speculation, either on lock acquisition, or on other calls that do not permit speculation. Upon termination, a check is done for locking conflicts. If a locking conflict is detected, the thread is restored and restarted, after updating the heap.

to show that the lock is no longer (speculatively) held, but again, other threads are not (yet) made aware of this. Once the decision is made to terminate the speculation run, it is committed in its entirety, as described below.

3.2 Committing a Speculation Run

Once speculation terminates, the thread either successfully commits all of its writes to shared memory, or reverts to its state prior to the start of speculation. To ensure progress, the next critical section executes without speculation.

The speculative commit process begins with the thread waiting until it has the minimum DLC, followed by conflict detection. Conflict detection confirms that none of the locks recorded in the local log L_i were acquired by another thread since speculation began (at DLC $BEGIN_i$). For this purpose, a global map G_l maintains the DLC of the most recent acquisition of lock l. The global map and other lock-related metadata are allocated when the lock is initialized. This map is updated following a successful commit, or any non-speculative lock acquisition. Note that LazyDet performs conflict detection purely on locks, not on the data addresses accessed by each thread (as with TM systems), since lock-level conflict detection is sufficient for determinism and memory consistency. LazyDet's versioned memory ensures a deterministic outcome, even in the presence of data races.

If the speculation run passes conflict detection, the remainder of the commit process proceeds as in Consequence.

3.3 Reverting and Restarting Execution

To support reverting execution following a failed speculation run, each thread maintains a copy of its stack and register contents from before the start of speculation. To revert, stack and register contents are restored from their copies. However, the heap and DLC are not restored: instead, the heap is updated to the most recently committed version, and the DLC is left unchanged. Intuitively, since all speculation decisions are made deterministically there is no need to restore the DLC, and leaving it unchanged helps maintain similar DLC values between threads. Naturally, the heap must be updated to reflect the global state of shared variables at the start of the new run.

3.4 Adaptive Speculation

The decision to speculatively elide DLC ordering should be an informed one. Repeated failed speculation attempts on contended locks can severely deteriorate program performance, a fact that has not gone unnoticed in the transactional memory (TM) community [20].

In order to make an informed, yet efficient, decision we store an array of per-thread metadata (a single 64-bit word for each thread) with each lock variable in the program.[3] The metadata represents a bit vector that records the last 64 speculation successes or failures for this lock, and is updated each time a speculation run is terminated. During speculation, on lock acquisition we use a *popcnt* instruction to count the set bits (successful speculation runs) in the current thread's metadata word and use the ratio of set bits to total bits to inform our decision making. Our current threshold requires an 85% success rate to begin speculating. For locks below that threshold, we retry every 20 attempts to account for program phase changes. The deterministic schedules produced by LazyDet are potentially sensitive to these parameters. In our evaluation, we tuned the parameters empirically for the hash table microbenchmark (§5.1), then applied the same parameters to all of our workloads. We anticipate that higher performance could be achieved with per-workload tuning.

3.5 Irrevocable Speculation

LazyDet supports deterministic irrevocability [43] to support I/O and other system calls that are inherently incompatible with speculation. Speculation runs cannot terminate during a critical section: work performed during the run must be committed, and this can only be done (while preserving correctness) at the end of a critical section. Thus, we cannot simply terminate speculation before the system call.

Rather than terminate the speculation, LazyDet *upgrades* a speculation run to "irrevocable" status when a system call is encountered. The system call can then be performed, and execution continues in irrevocable mode until no further locks are held; this is thefi rst feasible termination point.

[3]The per-thread aspect of the metadata is necessary to preserve determinism. During speculation the metadata is read and used to make decisions - and these decisions must be deterministic. If the metadata were shared, it could potentially be updated non-deterministically by another committing thread and cause different speculation decisions to be made across different executions.

An irrevocable speculation run bypasses the conflict detection step upon termination. Instead, conflict detection is performed during the *upgrade* process, after which point the thread blocks others from committing until the run can be terminated, to ensure there are no conflicts for the now irrevocable run. If a conflict is detected during the upgrade operation, the speculation run is reverted. In §5, irrevocable speculation is demonstrated to provide a 4× speedup on the *ferret* benchmark, over the alternative of reverting the speculation run when a system call is encountered, and then restarting without speculation.

Next, we introduce Deterministic, Data-Race Free (DDRF) consistency, a new memory consistency model that encompasses this behavior. While DDRF only *requires* data exchange along happens-before edges, it *allows* for deterministic data exchange at arbitrary, but deterministic, times.

4 Deterministic DRF Memory Consistency

LazyDet inherits the use of Conversion [32] for thread isolation and version control of main memory. Conversion maintains a single central "version list" to/from which threads commit their changes, and retrieve the most recent version of global memory. Combining speculative, partial-order deterministic synchronization with the shared version list of Conversion results in a new memory consistency model, that we call *Deterministic Data-Race-Free* (DDRF).

DDRF is a variant of a Data-Race-Free memory model [1]. A DRF model requires visibility only along *happens-before edges*, which arise either via 1) program order, 2) synchronization order between matched synchronization operations like a release and a subsequent acquire of the same lock or 3) the transitive closure of the two. DDRF extends DRF by providing a determinism guarantee, which holds even in the presence of data races (DRF provides no semantics for races). While DDRF requires visibility along happens-before edges, it (like DRF) does not rule out visibility at other program points, so long as determinism is preserved.

Other deterministic execution systems have adopted similar relaxed consistency models, such as the DRF model in RCDC (called DMP-HB) [19] or the Deterministic Lazy Release Consistency (DLRC) model from RFDet [26]. All of these consistency models are compatible with the memory models of mainstream languages like C/C++ [3, 11], and thus LazyDet supports code written in these languages. LazyDet's memory isolation mechanism inherits a limitation from RFDet [26] that, due to word tearing in the presence of silent stores, makes it incompatible with the Java memory model's restrictions on out-of-thin-air values.

DDRF is weaker than Total Store Order (TSO), the consistency model adopted by Consequence [30], as the example in Figure 4 shows. This program is a variant of the standard store buffer example from the consistency model literature, but with added synchronization. Under TSO, the program in Figure 4 should execute in a sequentially consistent manner because the lock acquires and releases act as full memory fences [39]. Thus, it is impossible under TSO for both loads to return zero (though, without any synchronization, such a result is possible under TSO).

initially, $x == y == 0$

Thread 1	Thread 2
acquire(A)	acquire(B)
store $1 \rightarrow x$	store $1 \rightarrow y$
release(A)	release(B)
acquire(A)	acquire(B)
load $r1 \leftarrow y$	load $r2 \leftarrow x$
release(A)	release(B)

Figure 4. A simple program that, under TSO, can never have both loads return zero. Under LazyDet's DDRF memory consistency model, however, such an outcome is possible.

In DDRF it is possible for the program in Figure 4 to have both loads return zero. Such an outcome can arise because stores need only be made visible along happens-before edges, and there are no such edges between the two threads in Figure 4. Thus, neither store becomes visible to either load, and both loads return zero.

4.1 Comparing relaxed consistency models

Despite their similarities, there are significant differences between the relaxed memory consistency models DDRF, DMP-HB and DLRC in their semantics for data races. The original DRF0 model [1] does not provide semantics for races: it only stipulates the conditions under which visibility is guaranteed and sequential consistency is preserved. [19] does not explain the semantics of races except to say that a deterministic outcome is guaranteed. While it is hard to state precisely DMP-HB's consistency guarantees, we believe they are equivalent to our DDRF model. In DDRF, we introduce a new *visibility order*, a partial order over arbitrary memory operations of an execution. Visibility order can be affected by *arbitrary, deterministic* program events, *e.g.*, the number of locations written by a thread. In DDRF, the value seen by a racy load l will arise from either a store s such that s happens-before l, or another store s' such that s' is ordered before l in visibility order. Because visibility order is deterministic, data races in DDRF always result in a deterministic outcome. Because visibility order is arbitrary, it is difficult to say a priori what the outcomes of a program can be. The definition of visibility order is deliberately general: thisfl exible definition of visibility order provides greater scope for optimization, just as with nondeterministic consistency models, as we discuss next.

RFDet's DLRC model allows a load to see the value of a store *iff* a happens-before edge exists from the store to the

initially, $x == 0$

Thread 1	Thread 2
store $1 \rightarrow x$	
release(A)	
...	...
	acquire(B)
	load $r1 \leftarrow x$

Figure 5. Under DLRC, $T2$'s load can never return 1, while under LazyDet's DDRF model it can. "$\cdots$" indicates a sequence of arbitrary, non-synchronization operations.

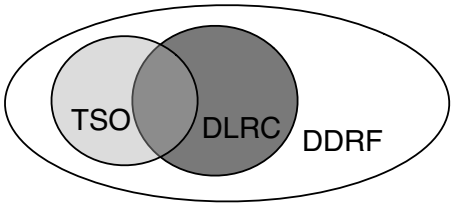


Figure 6. The relative strength of various consistency models. DLRC is incomparable to TSO, and DDRF admits more executions than either.

load; without such an edge, the store must not be visible to the load. Because of the strong restrictions imposed by the bi-conditional, DLRC's model is actually stronger than DDRF in that DLRC allows fewer executions than DDRF allows.[4] Figure 5 illustrates the relative strength of these models with an example program. The load by $T2$ can *never* return 1 in DLRC because there is never a happens-before edge from $T1$ to $T2$. Under DDRF, $T2$'s load can return either 0 or 1, as the threads' executions can induce a visibility order edge between $T1$'s store and $T2$'s load on some executions and not on others, albeit deterministically.

DLRC is not strictly weaker than TSO, as revisiting Figure 4 under DLRC reveals. With TSO, the loads in Figure 4 can never both return zero. Under DLRC, because there are no happens-before edges between the threads, both loads not only can return zero: they *must* return zero. Thus, the executions of this program allowed by TSO and DLRC are disjoint, and neither consistency model is strictly stronger or weaker.

Figure 6 illustrates the relative strength of TSO, DLRC and DDRF in terms of the set of executions allowed under each model. More relaxed models allow larger sets of executions than stronger models. As we show in §5, the extrafl exibility allowed by the DDRF model has significant systems implications, allowing LazyDet to achieve much better scalability than prior systems. In particular, DLRC's restrictions on when stores are visible forces it to retain many versions of memory while DDRF is free to coalesce these into a single version to save space and time.

[4]This is contrary to [26], which states that "DLRC is most similar to DMP-HB, but is more relaxed" [page 2].

4.2 Scalability Advantage of DDRF over DLRC

As a result of the central version list in LazyDet, any update from the version list includes all prior commits, creating a *visibility order* edge between updates and commits to the version list. Thus, even if no happens-before path exists between two threads, modifications will propagate between them due to the shared version list.

DDRF allows communication between threads, outside of happens-before, as long as the execution remains deterministic. For a program with l locks and t threads, DLRC requires the deterministic execution runtime to maintain $l + t$ versions of main memory, while DDRF requires only t versions.

5 Performance Evaluation

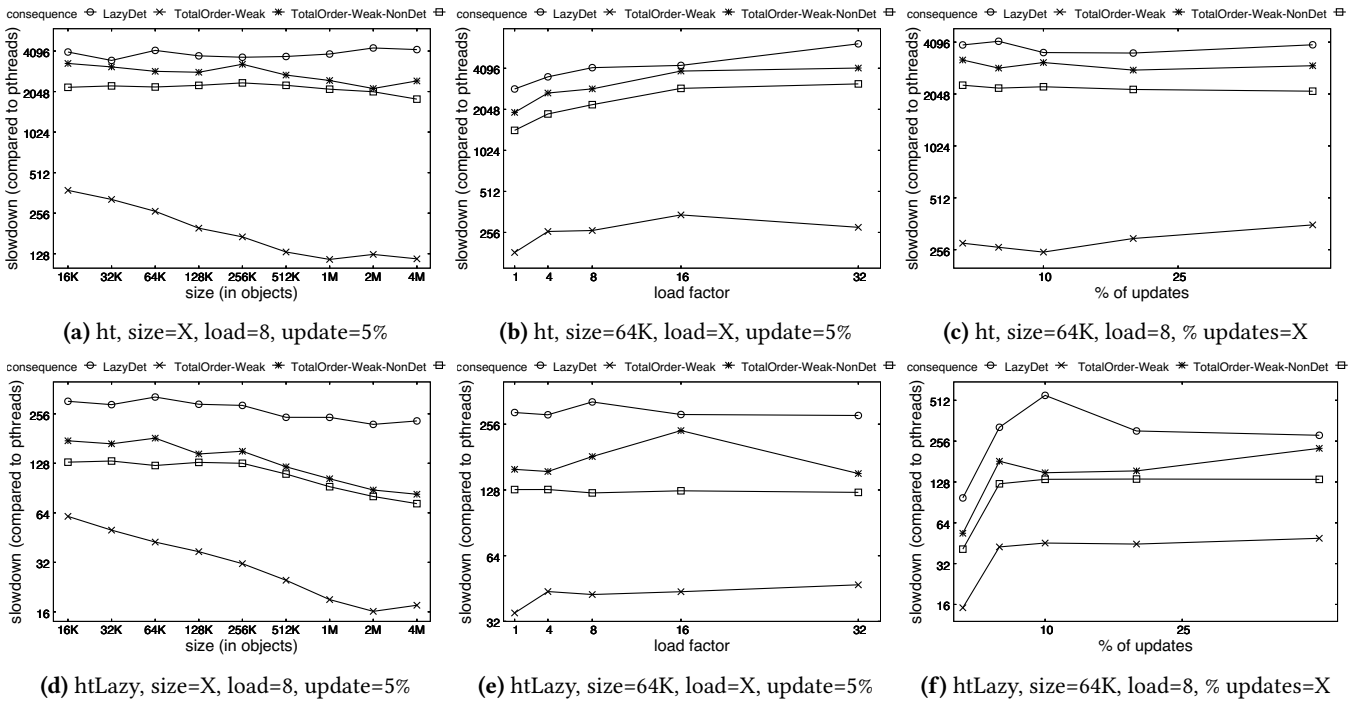
Below, we evaluate the performance of LazyDet, when running a variety of programs. First (§5.1), we evaluate a hash table microbenchmark, which measures in detail the impact of speculative deterministic execution on a highly concurrent data structure. Here, we demonstrate an order-of-magnitude throughput improvement with LazyDet. Following that, (§5.2) we summarize the performance of LazyDet on a wide variety of applications from the PARSEC [7], SPLASH-2 [44] and Phoenix [36] benchmark suites, where we achieve up to 2× improvement in performance.

5.1 Hash Table Microbenchmark

To better understand the impact of speculative deterministic execution on a highly concurrent data structure, wefirst evaluate LazyDet using the lock-based hash table found in Synchrobench [21]. For synchronization purposes, each bucket in this hash table is chained with either (1) hand-over-hand locking (*ht*) or (2) a lazy list-based set [23] (*htLazy*).

Figure 7 shows the runtime of both variants, normalized to Pthreads performance. In other words, we plot "slowdown" vs pthreads, thus lower is better. Starting from the left-most plot, we vary (1) the size of the hash table, (2) the load factor (number of items per bucket or maximum length of the chain), and (3) the percentage of updates performed during the experiment.

We compare LazyDet against other DMT systems that lack speculation, namely Consequence, as well as variants of LazyDet that trade determinism for performance. TotalOrder-Weak disables strong determinism, and thus speculation. TotalOrder-Weak-Nondet disables both strong determinism and the total ordering based on the DLC: instead of enforcing a specific order, it simply acquires a spin-lock. This still imposes a total ordering of lock acquisitions, just not



(a) ht, size=X, load=8, update=5% **(b)** ht, size=64K, load=X, update=5% **(c)** ht, size=64K, load=8, % updates=X

(d) htLazy, size=X, load=8, update=5% **(e)** htLazy, size=64K, load=X, update=5% **(f)** htLazy, size=64K, load=8, % updates=X

Figure 7. A DMT worst case scenario: Performance of DMT systems on a hash-table-based stress test from Synchrobench [21], with 32 threads. Variants are *ht*, which uses a linked list for chaining with hand-over-hand locking, and *htlazy*, which uses a lazy list-based set [23] for chaining. Results are shown as slowdown vs. pthreads (lower is better). Note the log y-axis.

deterministically across executions. Essentially, TotalOrder-Weak-Nondet helps us simulate the performance that could be achieved with a perfect logical clock.

At a high level, Figure 7 shows that LazyDet achieves roughly an *order of magnitude* increase in performance over Consequence, the current state of the art. Regarding the different data structure design choices, all DMT systems do better with *htLazy* than *ht* because it requires fewer lock acquisitions than the hand-over-hand locking approach. Regarding the configuration parameters, LazyDet does better as we increase the size of the data structure because the likelihood of a conflict is reduced. If we increase the load factor for *ht* performance decreases for all DMT systems, because the number of items in each linked list increases and thus we need to acquire more locks.

Overall, we can see from the results in Figure 7 that lazy determinism enables dramatic performance improvements over eager determinism for this type of synchronization-intensive, highly concurrent workload.

5.2 Application Benchmarks

We evaluate LazyDet against programs from the PARSEC-2, SPLASH-2 and Phoenix benchmark suites. Some of the programs are excluded due to being problematic for deterministic systems, either due to the use of atomic instructions or ad-hoc synchronization (see §A). For the *barnes* program from SPLASH-2, we made a slight modification to replace a singlefl ag variable with a condition variable to allow it to work with LazyDet.

Our testbed is a system with four 2.00GHz Intel Xeon E7-4820 8-core processors and 256GB of main memory, running Linux 2.6.37 with the kernel patches needed to run Consequence. For these experiments, Hyper-threading was turned off, and the frequency scaling governor was set to *performance*. Experiments were run 5 times per thread count, and we report the mean and standard deviation (error bars).

5.3 Identifying Where Lazy Determinism Could Make a Difference

Table 1 shows lock statistics for a number of benchmark programs, running with up to 8 threads. We show the number of lock variables, and the lock acquisitions per-variable by percentiles. The benchmarks highlighted in gray were selected for deeper analysis, based on the number of lock variables, lock acquisitions performed and distribution across threads. Among these, those with many lock variables are the benchmarks where LazyDet's lazy determinism holds the most promise. Benchmarks that synchronize frequently, but on a small number of lock variables (e.g. *reverse_index*), cannot be significantly improved through speculation. Here, the goal of LazyDet is simply to not significantly hurt performance. Clearly for benchmarks that don't synchronize with locks

program	# lock variables	# lock acquisitions	lock acquisitions per lock variable (percentile)				pthreads runtime (s)
			50th	75th	95th	100th (max)	
barnes	5222	94175	1	5	70	335	0.3
ocean_cp	15	6487	1	8	5326	5326	0.6
ferret	1004	532112	15	163	781	269993	2.5
water_nsquared	3447	11523	3	4	7	118	0.6
reverse_index	61	17774	1	2	9	17654	1.9
water_spatial	10	306	8	32	117	117	0.5
dedup	2244	246862	37	111	166	88806	6.5
radix	16	216	6	12	104	104	3.5
streamcluster	2	161	2	159	159	159	1.5
fft	3	63	8	53	53	53	0.8
blackscholes	1	2	2	2	2	2	0.4
swaptions	1	2	2	2	2	2	3.6
linear_regression	1	2	2	2	2	2	1.2
word_count	1	2	2	2	2	2	2.1
matrix_multiply	1	2	2	2	2	2	4.9
pca	1	2	2	2	2	2	6.3
string_match	1	2	2	2	2	2	1.8
lu_cb	0	0	0	0	0	0	0.2
lu_ncb	0	0	0	0	0	0	0.1

Table 1. Lock statistics for PARSEC, SPLASH-2, and Phoenix benchmark programs. The percentiles show lock acquisitions per lock variable. Programs like *ferret* and *dedup*, which have a large number of lock variables, are ideal for lazy determinism.

(e.g. *blackscholes*), the speculation techniques of LazyDet are not relevant. These receive only a brief evaluation below.

Figure 8 shows the overall performance of LazyDet when compared to Consequence. We did not compare directly against DThreads. However, in [30], Consequence was consistently faster than DThreads. We also include implementations of weak determinism (TotalOrder-Weak, similar to [33]) and weak determinism with nondeterministic synchronization (TotalOrder-Weak-Nondet), derived from the LazyDet code base. For TotalOrder-Weak-Nondet, synchronization is still totally ordered: we implement this as a simple lock acquisition, instead of using the DLC.

Here we show the best runtime of each library normalized to the best runtime of (nondeterministic) pthreads. The group on the left shows applications where speculation on lock variables may have some impact, either positive or negative. For *barnes, ferret, dedup* and *water_nsquared* - the class of programs where speculation may help - the average runtime of LazyDet is 53% of Consequence. For *water_nsquared*, LazyDet even outperforms TotalOrder-Weak. Overall, these results demonstrate that speculative deterministic execution offers substantial performance improvements on several benchmarks, in some cases doubling performance compared to Consequence.

In addition to runtime performance, Figure 10 shows the CPU utilization results for the lock-based programs running with 16 threads. The results show that LazyDet's improvement in runtime does not imply an increase in CPU usage.

A noticeable performance regression occurs in *radix*, where LazyDet is regressed relative to Consequence by 18%. This is due to a high failure rate combined with expensive reverts and a small number of lock acquisitions per-thread (less than 20 in the 32 thread experiment), which doesn't give LazyDet enough time to learn to avoid speculation. It should be noted that even in the presence of applications that use highly contended locks such as *reverse_index*, the regression is a modest 6% due to LazyDet's adoption of Consequence's coarsening optimization [30].

5.4 Speculation Performance Factors

Figure 11 evaluates the performance of various aspects of LazyDet's speculation engine: LazyDet-NoCoarsening limits its speculation to one critical section. LazyDet-NoIrrevocable disables upgrading a speculative run to irrevocable, when encountering code that cannot be executed speculatively (see §3.5). LazyDet-NoPerLockStats replaces per-lock speculation statistics with per-thread statistics.

Different optimizations are critical for different programs. For example, with *ferret* the performance depends on coarsening several critical sections as well as using irrevocable speculation. The reason for this is that one thread performs a high frequency of lock acquisitions with little work between

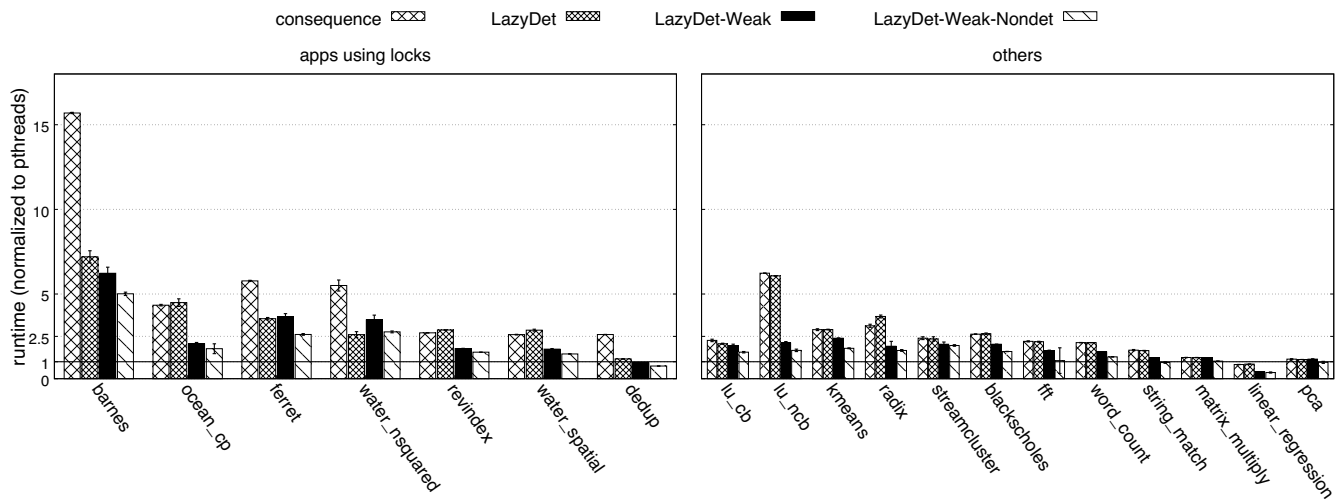


Figure 8. Performance results comparing the best runtime of each library normalized to pthreads runtime (any thread count). Lazy determinism with LazyDet achieves up to 2× speedup vs. eager determinism with Consequence. (lower is better)

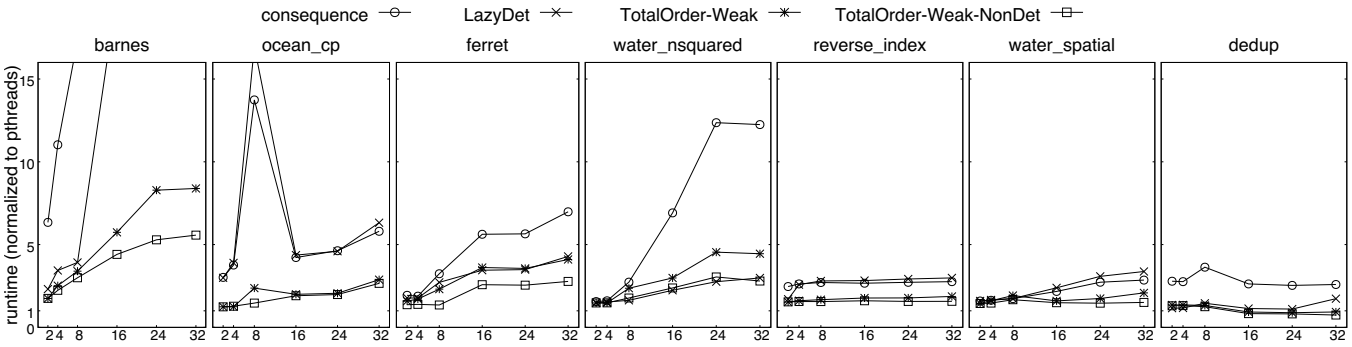


Figure 9. Scalability results for relevant benchmarks, normalized to pthreads runtime. For most benchmarks tested, LazyDet improves substantially on Consequence scalability. (lower is better)

to amortize the cost. Further, system calls (e.g. mmap/munmap) are invoked while holding locks, leading to the need to upgrade the speculation to irrevocable. We can also see that keeping per-lock statistics provides a modest performance improvement on *ocean_cp*, while it slightly degraded performance for *barnes* and *water_nsquared* which maintain thousands of locks with largely uniform access patterns.

Table 2 shows speculation statistics for our benchmark programs. For *barnes*, the statistics paint a picture as to why the performance does not scale beyond 8 threads (see Figure 9). First, the mean speculation length (in number of critical sections) is reduced from 19.28 to 5.86 when moving from 8 to 16 threads. Second, the percentage of critical sections executed speculatively drops from 98.88% to 68.18%. This indicates that lock-set conflicts between threads both reduced the ability to coarsen and led to failed speculation and falling back to traditional lock acquisition.

Finally, Figure 12 shows a scatter-plot of revert cost vs. change set size, as experienced in the programs from Table 2. The mean revert time is approximately 11,000 cycles, and the black line shows a linear regression of the relationship between the size of the change set and the revert time.

In summary, LazyDet offers substantial performance improvements to a wide range of applications, but is particularly helpful for synchronization-heavy, yet highly parallel programs.

6 Related Work

Research in deterministic concurrency has led to the development of several types of systems that support deterministic execution, including hardware architectures [15, 17–19, 25, 38], runtime systems [14, 27, 30, 32, 33], programming languages [8–10, 22, 37, 40], compilers [4, 16, 26] and entire operating systems [2, 24, 41].

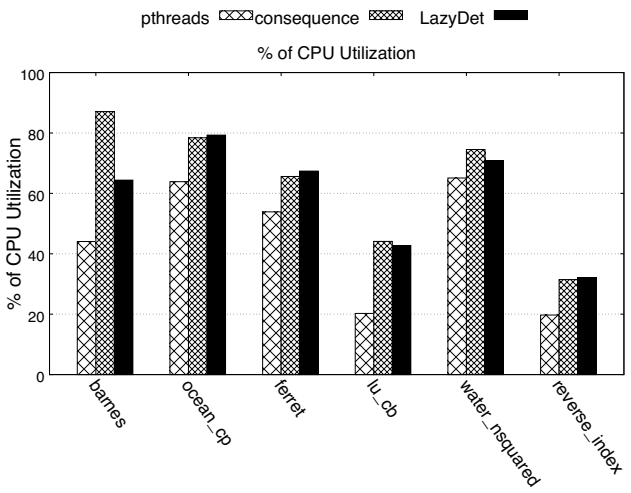


Figure 10. CPU utilization numbers for lock-based programs. LazyDet does not significantly impact CPU usage relative to Consequence.

program	threads	% spec. acquisitions	% spec. success	mean spec. length (in CS)
barnes	8	97.0%	82.5%	19.3
	16	58.5%	50.3%	5.2
	32	50.2%	38.4%	3.8
ocean_cp	8	13.8%	7.41%	1.0
	16	13.1%	5.27%	1.0
	32	12.6%	3.5%	1.0
ferret	8	99.9%	99.8%	38.4
	16	100.0%	99.9%	40.5
	32	100.0%	99.9%	40.4
water_nsquared	8	99.7%	91.5%	56.1
	16	99.5%	91.2%	31.3
	32	99.2%	91.0%	24.1
reverse_index	8	10.5%	13.7%	4.2
	16	7.7%	4.9%	1.0
	32	0.0%	0.0%	N/A
water_spatial	8	45.5%	12.6%	1.0
	16	41.4%	10.5%	1.0
	32	40.1%	7.2%	1.0
dedup	8	51.8%	69.4%	3.3
	16	54.1%	62.9%	2.2
	32	58.1%	61.5%	2.3

Table 2. Speculation statistics for a number of benchmark programs. Speculation length (# of critical sections) and success correlates with benchmark program performance.

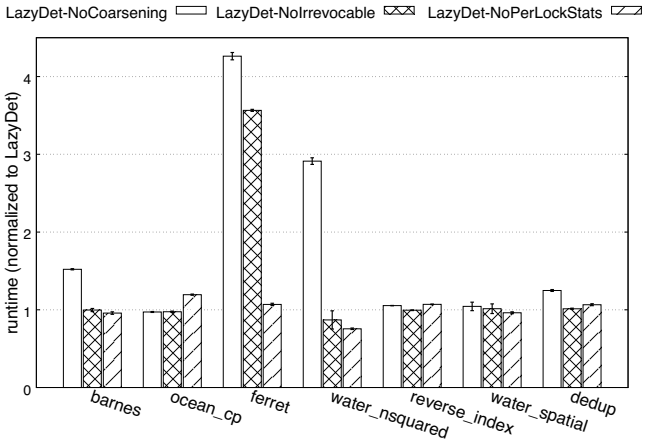


Figure 11. Performance of benchmark programs with certain speculation features disabled. The y-axis shows runtime normalized to the runtime of LazyDet with all speculation features enabled. (lower is better)

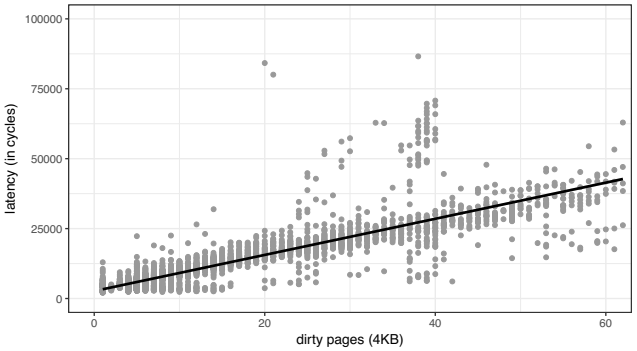


Figure 12. Scatter-plot of revert cost vs. change set size. Black line shows linear regression. Mean revert cost is approximately 11,000 cycles.

These various types of deterministic multithreading (DMT) systems differ primarily in the determinism guarantees they provide and the type of deterministic logical clock (DLC) they employ. All DMT systems impose a "SyncOrder determinism" constraint that ensures a deterministic order of synchronization operations and, optionally, a stronger "Dataflow determinism" constraint that ensures a deterministic value for every load [28]. Most DMT systems impose both constraints, though the Kendo system [33] demonstrated that the Dataflow constraint can be relaxed to improve performance.

A common trait among all of these systems, including recent work such as [26, 30, 32], is that they all impose a total order on synchronization operations. In further acknowledgment of this bottleneck, related work on "stable multithreading" has resorted to relaxing deterministic synchronization guarantees [12, 13] to improve performance.

Like LazyDet, several previous DMT systems isolate threads between synchronization operations using the virtual memory subsystem [26, 27, 30, 32]. Others use compiler support [4, 26], or avoid this problem entirely by assuming that programs are data-race free [33].

In terms of specific DLC mechanisms, many schemes use a deterministic instruction counter [15, 17, 19, 26, 30, 33], while others count synchronization operations [27, 32]. In principle, counts of basic blocks, function calls and many other deterministic program features could be used as the basis for a DLC. The choice of DLC determines the ordering of synchronization operations — thus a poor ordering can result in long delays as threads wait their turn. Here, the instruction counter solution provides the highest accuracy, but also incurs the largest overhead.

6.1 Speculative Execution and Relaxed Memory Models

LazyDet uses speculation to surmount the total order bottleneck that limits previous DMT systems. Speculation (in the form of hardware transactional memory) was employed in DMP [17], one of the earliest DMT designs. Follow-on work to DMP [4] dismissed speculation as overly complex and instead achieved performance via relaxed memory consistency models, while still utilizing totally ordered synchronization operations. The Consequence system [30] argued that relaxed consistency buys relatively little given the restrictions of totally ordered synchronization, demonstrating that strong memory models like TSO can match the performance of weaker models.

While more relaxed memory consistency can in principle offer performance gains for determinism, just as it does for nondeterministic parallelism, these gains cannot be realized so long as synchronization remains totally ordered. In some restricted contexts, it is possible to reason about conflicts a priori and prove their absence, *e.g.*, the Deterministic Parallel Java [9, 10, 42] type-and-effect system can do so for data-parallel fork-join programs. LazyDet is thefi rst deterministic system to move past the total order bottleneck for arbitrary programs.

6.2 Transactional Memory

In many ways, the speculation engine in LazyDet has similarities to a software transactional memory (STM) system. LazyDet uses, in STM parlance, lazy conflict detection of committed critical sections,fl attened nesting, and irrevocable speculation.

The main difference between HTM/STM and our approach is that we are detecting conflicts at the level of *synchronization variables*, not memory accesses. In traditional nondeterministic systems this technique would be of little use because critical sections protected by distinct locks can already execute in parallel. However, DMT systems have been hamstrung by the total order bottleneck until LazyDet.

Another difference is that strong determinism makes strong isolation inherent to deterministic multithreading systems. This means that the mixed-mode access problem commonly discussed in the STM literature does not arise with LazyDet or other strong determinism systems.

The speculation engine in LazyDet has similarities to Speculative Lock Elision (SLE) [34] and Speculative Synchronization (SS) [29]. These prior schemes also attempt to automatically speculate past lock acquires. However, a crucial distinction with this prior work is that LazyDet doesn't allow parallelism between critical sections protected by the same lock – the main purpose of SLE/SS. Using SLE/SS techniques to unlock additional parallelism is an intriguing direction for future work in deterministic parallelism. Sophisticated forms of conflict detection, like Dependence-Aware TM [35], might additionally prove useful for reducing reverts.

7 Limitations

Despite the performance benefits that speculation brings, many avenues for improvement remain in DMT systems. Better support for atomics is one natural extension for LazyDet. Determinism imposes high overheads on these mechanisms, which are chosen by developers explicitly for their speed. Allowing speculative execution of atomics, perhaps detecting conflicts only on locations accessed by the atomics, should improve performance significantly.

While LazyDet adapts its speculation to run programs with arbitrary system calls, these system calls may compromise LazyDet's determinism guarantee. Enforcing determinism for a wide range of I/O operations is beyond the scope of LazyDet, but an intriguing path for future work.

8 Conclusion

In this work we have shown that deterministic multithreading need not be limited by the total order bottleneck on synchronization that has restricted previous systems. Our system, LazyDet, demonstrates that speculation is an effective mechanism for breaking through this bottleneck, while preserving strong determinism guarantees. LazyDet also shows that, counterintuitively, strong determinism can outperform weak determinism for some programs that synchronize frequently, because strong determinism provides a useful foundation for speculation that makes it possible to contain and rollback the effects of misspeculations.

9 Acknowledgments

This work was made possible through National Science Foundation grants CNS-1320235 and CNS-1703425.

A Incompatible Benchmarks

As Table 3 shows, some benchmarks from PARSEC [6] and SPLASH-2 [44] are incompatible with LazyDet, as well as many other DMT systems [26, 27, 30]. This is in part due

benchmark suite	program	incompatibility reason
PARSEC 2	bodytrack	shared stack variables
	canneal	atomic instructions
	fluidanimate	shared stack variables
	facesim	shared stack variables
	x264	shared stack variables
Splash 2	cholesky	ad-hoc synchronization
	barnes*	ad-hoc synchronization
	radiosity	ad-hoc synchronization
	volrend	ad-hoc synchronization

Table 3. A list of benchmarks that are incompatible with current DMT systems, and why. *Manually modified in our evaluation to use pthreads synchronization instead.

to the choice to rely on synchronization operations to enforce memory visibility. This design choice breaks ad-hoc synchronization such asfl ag-based synchronization: a thread polling on afl ag never performs a synchronization operation and thus never receives any updates from remote threads that set thefl ag. Additionally, the use of atomic instructions requires compiler support to instrument the atomic instruction so that the DMT system can perform DLC operations and memory updates; otherwise, the atomic instruction may operate on a thread-private copy of memory which violates atomicity and memory ordering semantics.

Note that DMT systems, including LazyDet, preserve determinism even when ad-hoc synchronization or atomic operations are broken. The resulting deadlocks or program crashes are repeatable and thus much easier to diagnose than in a nondeterministic environment. We document these issues here to facilitate future work addressing these issues.

References

[1] Sarita V. Adve and Mark D. Hill. 1990. Weak ordering - a new definition. 2–14.

[2] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2010. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*.

[3] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

[4] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 2010. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*. Pittsburgh, Pennsylvania, USA, 53. https://doi.org/10.1145/1736020.1736029

[5] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. 2010. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. *ACM SIGARCH Computer Architecture News* 38, 1 (2010), 53–64.

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. *The PARSEC Benchmark Suite: Characterization and Architectural Implications*. Technical Report TR-811-08. Princeton University.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT (PACT '08)*.

[8] Robert Bocchino, Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. 2011. Safe nondeterminism in a deterministic-by-default parallel language. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '11*. Austin, Texas, USA, 535. https://doi.org/10.1145/1926385.1926447

[9] Robert Bocchino, Mohsen Vakilian, Vikram Adve, Danny Dig, Sarita Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, and Hyojin Sung. 2009. A Type and Effect System for Deterministic Parallel Java. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '09*. Orlando, Florida, USA, 97. https://doi.org/10.1145/1640089.1640097

[10] Robert L. Bocchino and Vikram S. Adve. 2011. Types, regions, and effects for safe programming with object-oriented parallel frameworks. In *Proceedings of the 25th European conference on Object-oriented programming (ECOOP'11)*. Berlin, Heidelberg, 306–332. http://dl.acm.org/citation.cfm?id=2032497.2032519

[11] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08*. Tucson, AZ, USA, 68. https://doi.org/10.1145/1375581.1375591

[12] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. 2013. Parrot: A Practical Runtime for Deterministic, Stable, and Reliable Threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 388–405. https://doi.org/10.1145/2517349.2522735

[13] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. 2011. Efficient Deterministic Multithreading Through Schedule Relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. ACM, New York, NY, USA, 337–351. https://doi.org/10.1145/2043556.2043588

[14] Derek R. Hower and Mark D. Hill. 2011. Hobbes: CVS for Shared Memory. In *Workshop on Determinism and Correctness in Parallel Programming*.

[15] Derek R. Hower, Polina Dudnik, David A. Wood, and Mark D. Hill. 2011. Calvin: Deterministic or Not? Free Will to Choose. In *Proceedings of the 17th International Symposium on High-Performance Computer Architecture (HPCA)*.

[16] Joseph Devietti, Dan Grossman, and Luis Ceze. 2012. The Case For Merging Execution- and Language-Level Determinism with MELD.

[17] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: Deterministic Shared Memory Multiprocessing. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS '09)*. Washington, DC, USA, 85. https://doi.org/10.1145/1508244.1508255

[18] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2010. DMP: Deterministic Shared Memory Multiprocessing. *IEEE Micro* 30, 1 (Jan. 2010), 40–49. https://doi.org/10.1145/1508244.1508255

[19] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. 2011. RCDC: A Relaxed Consistency Deterministic Computer. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*.

[20] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. 2014. Adaptive Integration of Hardware and Software Lock Elision Techniques. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '14)*. ACM, New York, NY, USA,

188–197. https://doi.org/10.1145/2612669.2612696

[21] Vincent Gramoli. 2015. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2688500.2688501

[22] Guy Blelloch. 1992. *NESL: A Nested Data-Parallel Language*. Technical Report CMU-CS-92-103. Carnegie Mellon University, Pittsburgh, PA.

[23] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. 2006. A Lazy Concurrent List-based Set Algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS'05)*. Springer-Verlag, Berlin, Heidelberg, 3–16. https://doi.org/10.1007/11795490_3

[24] Nicholas Hunt, Tom Bergan, Luis Ceze, and Steven D. Gribble. 2013. DDOS: taming nondeterminism in distributed systems. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, Vol. 48. 499–508. https://doi.org/10.1145/2451116.2451170

[25] Hadi Jooybar, Wilson W.L. Fung, Mike O'Connor, Joseph Devietti, and Tor M. Aamodt. 2013. GPUDet: a deterministic GPU architecture. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '13)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2451116.2451118

[26] Kai Lu, Xu Zhou, Tom Bergan, and Xiaoping Wang. 2014. Efficient Deterministic Multithreading Without Global Barriers. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[27] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. New York, NY, USA, 327–336. https://doi.org/10.1145/2043556.2043587

[28] Li Lu and Michael L. Scott. 2011. Toward a Formal Semantic Framework for Deterministic Parallel Programming. In *Proceedings of the 25th International Conference on Distributed Computing (DISC'11)*. Springer-Verlag, Berlin, Heidelberg, 460–474. http://dl.acm.org/citation.cfm?id=2075029.2075086

[29] José F. Martínez and Josep Torrellas. 2002. Speculative Synchronization: Applying Thread-level Speculation to Explicitly Parallel Applications. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. ACM, New York, NY, USA, 18–29. https://doi.org/10.1145/605397.605400

[30] Timothy Merrifield, Joseph Devietti, and Jakob Eriksson. 2015. High-performance Determinism with Total Store Order Consistency. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 31, 13 pages. https://doi.org/10.1145/2741948.2741960

[31] Timothy Merrifield and Jakob Eriksson. [n. d.]. Conversion, Multi-Version Concurrency Control for Main-Memory Segments. In *Proceedings of the 8th ACM european conference on Computer Systems (EuroSys '13)*.

[32] Timothy Merrifield and Jakob Eriksson. 2013. Conversion: multi-version concurrency control for main memory segments. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 127–139. https://doi.org/10.1145/2465351.2465365

[33] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: Efficient Deterministic Multithreading in Software. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems - ASPLOS '09*. Washington, DC, USA, 97. https://doi.org/10.1145/1508244.1508256

[34] Ravi Rajwar and James R. Goodman. 2001. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture (MICRO 34)*. Washington, DC, USA, 294–305. http://dl.acm.org/citation.cfm?id=563998.564036

[35] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. 2008. Dependence-aware Transactional Memory for Increased Concurrency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, Washington, DC, USA, 246–257. https://doi.org/10.1109/MICRO.2008.4771795

[36] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA*. 13–24.

[37] Martin C. Rinard and Monica S. Lam. 1998. The design, implementation, and evaluation of Jade. *ACM Transactions on Programming Languages and Systems* 20, 3 (May 1998), 483–545. https://doi.org/10.1145/291889.291893

[38] Cedomir Segulja and Tarek S. Abdelrahman. 2012. Architectural support for synchronization-free deterministic parallel programming. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/HPCA.2012.6169038

[39] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture* 6, 3 (May 2011), 1–212. https://doi.org/10.2200/S00346ED1V01Y201104CAC016

[40] Stephen Heumann and Vikram Adve. 2012. Tasks with Effects: A Model for Disciplined Concurrent Programming.

[41] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven Gribble. 2010. Deterministic process groups in dOS. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*.

[42] Mohsen Vakilian, Danny Dig, Robert Bocchino, Jeffrey Overbey, Vikram Adve, and Ralph Johnson. 2009. Inferring Method Effect Summaries for Nested Heap Regions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. Washington, DC, USA, 421–432. https://doi.org/10.1109/ASE.2009.68

[43] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. 2008. Irrevocable Transactions and Their Applications. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '08)*. ACM, New York, NY, USA, 285–296. https://doi.org/10.1145/1378533.1378584

[44] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture (ISCA '95)*. ACM, New York, NY, USA, 24–36. https://doi.org/10.1145/223982.223990