High-Performance Determinism with Total Store Order Consistency

Timothy Merrifield
University of Illinois at Chicago
tmerri4@uic.edu

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

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

Abstract

We present CONSEQUENCE, a deterministic multi-threading library. CONSEQUENCE achieves deterministic execution via store buffering and strict ordering of synchronization operations. To ensure high performance under a wide variety of conditions, the ordering of synch operations is based on a deterministic clock [25], and store buffering is implemented using version-controlled memory [23].

Recent work on deterministic concurrency [14, 19] has proposed relaxing the consistency model beyond total store order (TSO). Through novel optimizations, CONSEQUENCE achieves the same or better performance on the Phoenix, PARSEC and SPLASH-2 benchmark suites, while retaining TSO memory consistency. Across 19 benchmark programs, CONSEQUENCE incurs a worst-case slowdown of 3.9× vs. pthreads, with 14 out of 19 programs at or below 2.5×. We believe this performance improvement takes parallel programming one step closer to "determinism by default."

1. Introduction

With multi-core processors comes the need for parallel programs. Unfortunately, writing parallel programs is hard. On top of the difficulties of writing correct sequential programs, parallelism brings nondeterminism which undermines our ability to debug, test and understand our programs.

Research into deterministic concurrency seeks to alleviate these problems. While some initial proposals focused on the design of deterministic multi-core architectures [12–14], subsequent work has demonstrated practical pure-software implementations of determinism [19, 21, 23].

One of the central techniques for improving performance in both hardware and software deterministic systems has been relaxing the memory consistency model. While initial proposals adopted strong consistency models like sequential consistency [13] and total store order (TSO) [5], subsequent work relies on extremely relaxed consistency models like DRF0 [14] and lazy release consistency (LRC) [19] to reduce inter-thread communication and thus improve performance. However, all deterministic execution systems, relaxed consistency or not, require global ordering of inter-thread communication, a likely bottleneck in any deterministic concurrency system. We argue that while this bottleneck remains, relaxing the consistency model is of limited benefit. Moreover, the adoption of an extremely relaxed consistency model is not free. Supporting LRC, as [19] does, suffers from high space overheads that hinder scalability. The programmability challenges inherent in relaxed consistency are also well-known, and can result in unintuitive behavior [1, 4]. Finally, some current hardware memory consistency models like x86 are not as relaxed as LRC. Moving to a weaker consistency model breaks compatibility with existing binaries. In contrast, a system that offers consistency guarantees compatible with modern architectures can support legacy binaries simply by replacing the pthreads library.

CONSEQUENCE demonstrates that a deterministic implementation of TSO, a relatively strong consistency model that is compatible with all modern hardware platforms, can perform well across a range of workloads. The primary contributions of this paper are:

- We describe several TSO-compatible optimizations that target the same scenarios that relaxed consistency optimizes, reducing the scope for further improvements from relaxed consistency.
- We demonstrate deterministic adaptation to program behavior as a means of further improving performance.
- We describe novel implementations of deterministic synchronization operations that are flexible and increase the parallelism of prior techniques.
- Finally, we demonstrate a deterministic implementation of TSO that achieves a 2.8x and 2.2x improvement over DThreads [21] and DWC [23] (respectively) on the five most challenging benchmark programs.

EuroSys'15, April 21–24, 2015, Bordeaux, France.

ACM 978-1-4503-3238-5/15/04…$15.00.
http://dx.doi.org/10.1145/2741948.2741960

While determinism can simplify parallel programming, using complicated memory consistency models to make determinism fast under-cuts these programmability gains. CONSEQUENCE erases this tension by combining determinism and strong memory consistency.

2. CONSEQUENCE: Background, Design and its Motivation

Current deterministic execution systems [19, 21, 23] allow an arbitrary multi-threaded program written in a conventional parallel programming language (like C or Java) to execute in a deterministic manner (the *singleton* definition of determinism [22]). The program's output and succession of internal states become solely a function of the program's explicit inputs and are unaffected by the non-deterministic interleaving of shared memory operations or scheduler policy.

CONSEQUENCE is intended as a high-performance, deterministic, drop-in replacement library for pthreads. Any pthreads-compatible program may be linked with CONSEQUENCE to create a deterministic, yet efficient version of the same. Determinism is assured for all programs, even those with data races. However, the use of atomic instructions or ad-hoc synchronization methods such as spinning on flag variables may (deterministically) result in incorrect behavior (§2.7). Further, in order to ensure correctness memory conflicts are merged at a byte granularity. In the presence of data races, this can result in outcomes not possible in non-deterministic execution [19]. In these aspects CONSEQUENCE offers the same determinism and progress guarantees as other recent deterministic execution systems [19, 21]. In contrast with some prior work [25], CONSEQUENCE does not assume that programs are race-free.

We now briefly summarize the design of CONSEQUENCE. To achieve determinism, CONSEQUENCE uses a deterministic, instruction-count based logical clock (§2.1), and memory isolation using page table manipulation and deterministic, byte-level merging at synchronization points (§2.4). All pthreads synchronization operations are replaced with deterministic equivalents, which ensure a strict ordering of these operations (§2.6) based on the logical clock. CONSEQUENCE also supports ad-hoc synchronization through periodically (and deterministically) synchronizing memory in the absence of explicit synchronization operations (§2.7).

Below, we provide background on deterministic concurrency, and describe and motivate each of the design choices made in CONSEQUENCE.

All deterministic execution schemes must provide two key components: 1) a *deterministic logical clock* to enforce a deterministic order of synchronization operations and 2) a *deterministic memory consistency model* that ensures loads return deterministic values even in the presence of unsynchronized accesses. Together, these two components are sufficient to guarantee the determinism of arbitrary programs. Below, we present and motivate the primary decisions that guided the design of CONSEQUENCE, while reviewing the alternative designs used in prior work.

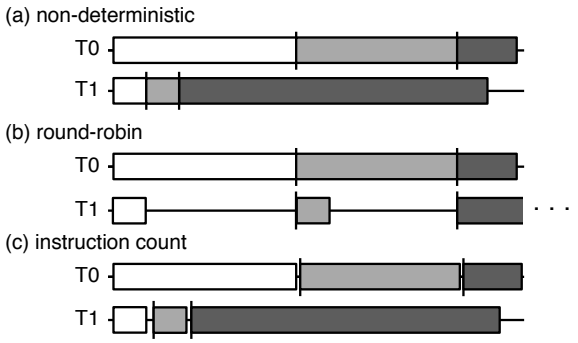

Figure 1: Effect of mismatched rate of synchronization operation. Vertical bars–sync. ops, boxes–work performed. (a) Mismatch causes no waiting with non-deterministic execution, but (b) can incur extensive waiting with round-robin deterministic logical clocks. (c) An instruction-count based logical clock [25] incurs waits close to those of a non-deterministic system.

2.1 Deterministic Logical Clocks

A deterministic logical clock is a mechanism that provides a deterministic total ordering for synchronization operations. A total order is necessary because synchronization operations must be globally coordinated in the general case when the system has no knowledge of what synchronization threads may attempt to perform next. The deterministic logical clock must ensure that, e.g., if two threads try to acquire the same lock, the same thread always wins.

The simplest logical clock is a round-robin policy and has been proven to work well in many prior schemes [5, 12, 13, 21, 23]. The round-robin policy works well when threads perform synchronization operations at the same rate. However, a thread that performs synchronization operations frequently can end up spending most of its time waiting for a thread that synchronizes rarely (Figure 1b).

A more efficient policy was proposed in Kendo [25] where the number of user instructions retired determines the deterministic ordering. The number of retired instructions may be determined using either performance counters [14, 25] or compiler instrumentation [19]. Under this design, a deterministic total ordering is obtained by allowing only the thread with the global minimum instruction count (GMIC) to perform a synchronization operation.

If all instructions required the same amount of time to execute, the Kendo approach would produce a very good ordering in the sense that threads would never have to wait long to perform a synchronization operation (Figure 1c). However, instruction latencies can vary greatly depending on the particular instruction being executed or non-deterministic processor state (e.g cache). Compounding this issue, the library implementations of deterministic synchronization op-

erations can contain non-deterministic code such as system calls or acquisition of spin locks for internal purposes. To avoid non-determinism, an obvious solution is to disable the instruction counting whenever such code may be executed. Both instruction latency variance and clock disabling can lead to *clock skew*, where the logical clock becomes increasingly disconnected from real time.

In CONSEQUENCE, we use the retired instruction count for ordering based on hardware performance counters. Threads pass a single *global token* between themselves and the token can only be acquired by the GMIC thread. We mitigate clock skew by allowing clocks to deterministically "fast-forward" as described in §3.5. While we have observed a small degree of nondeterminism in our performance counter measurements (consistent with other work [30]) our logical clock is sound in the presence of deterministic performance counters and could also be constructed from lightweight compiler-based instruction counting [19].

2.2 Deterministic Memory Consistency

A deterministic memory consistency model ensures that loads from shared memory always return the same value across program runs even in the presence of unsynchronized accesses, i.e., data races. While most loads are well-synchronized, and thus rendered deterministic by deterministic synchronization operations (§2.1), unsynchronized loads require additional effort. While [25] provides deterministic memory consistency by assuming data-race-free programs, most deterministic consistency models are built around the idea of temporarily isolating each thread in its own memory space. When threads are isolated, a multi-threaded program is effectively converted into a collection of single-threaded programs, each of which are inherently deterministic. Of course, threads must periodically be allowed to communicate with each other. The strength of the memory consistency model determines when a thread's updates, accumulated in isolation, must be made visible to other threads: stronger models require updates to be shared more frequently and more widely than weaker models. The key design decision for deterministic memory consistency is the semantics of a memory fence, the mechanism that controls how updates propagate to remote threads. We refer to the process of making updates visible as a *commit* operation to distinguish it from the hardware's memory fence instructions. While commit operations are not inherently deterministic, they are rendered so by enforcing a commit ordering with a deterministic logical clock (§2.1).

2.3 Total Store Order is Enough

Relaxing the consistency model improves the performance of determinism by making commits cheaper. Consistency models like sequential consistency and total store order require that stores become visible in a total order that all threads agree on: commits must thus be a global operation visible to all threads (Figure 2). Relaxed consistency models

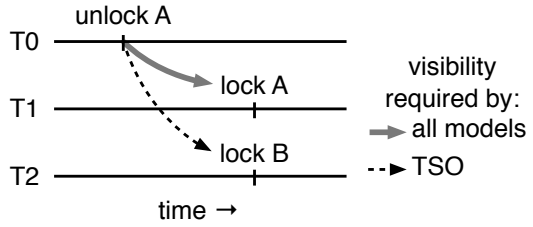

Figure 2: TSO requires that commits are global, while more relaxed models allow local commits that are visible to a subset of threads.

like DRF0 [14] and LRC [19] allow commits to be "point-to-point" with respect to a given synchronization object, i.e., the commit performed when releasing a lock need be visible only to the thread that subsequently acquires that lock.

While an LRC-based system may in principle offer better performance than a TSO-based system, the adoption of such an extremely relaxed[1] consistency model entails several additional costs. One concern is a space leak that arises from making commits visible only via a particular synchronization object. In [19], if a thread modifies some data and releases lock A, those modifications must be recorded until some other thread acquires lock A. This causes space usage to scale with the number of lock objects. In the case that no other thread ever acquires lock A, space will be permanently leaked. This space overhead is not just a theoretical concern: in [19] large space overheads for one LRC-based system restricted the evaluation of some programs to just four threads.

In addition to its space overheads, weak consistency models like LRC are difficult for both humans [1] and automated tools [4, 9] to understand. LRC is more relaxed than the TSO consistency models of x86 and SPARC, and thus legacy binaries on those platforms can break when moved to LRC.

CONSEQUENCE adopts the TSO consistency model, offering a familiar programming environment and compatibility with binaries compiled for any modern hardware platform. In our evaluation (§5), we demonstrate that efficient determinism is achievable without sacrificing TSO consistency.

2.4 Implementing Commits

While the memory consistency model does somewhat constrain the possible implementations of the commit operation, many alternative designs are still possible for a given consistency model. For instance, several deterministic execution systems implement the TSO consistency model by dividing program execution into chunks consisting of a fixed number of instructions (typically 10,000-100,000) and placing a commit at the end of each chunk [5, 12]. Chunks are terminated early if they contain a synchronization operation. However, memory models like TSO do not require commits at places other than synchronization operations. Thus,

[1] It is not obvious how further relaxations beyond LRC would be possible without breaking programming language semantics.

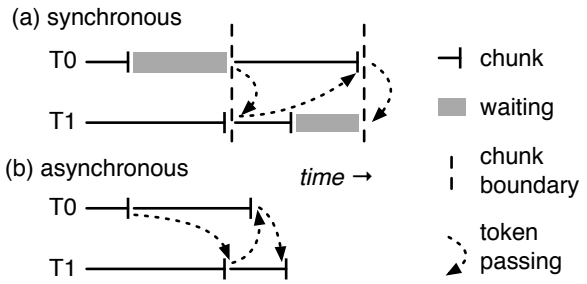

Figure 3: Synchronous versus asynchronous commit.

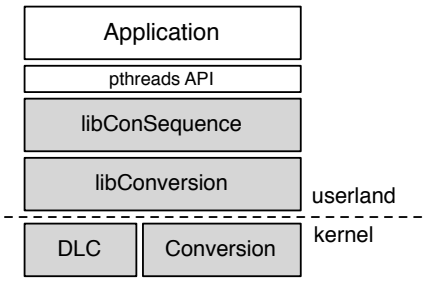

Figure 4: CONSEQUENCE architecture diagram. Shaded components indicate our system. CONSEQUENCE is implemented primarily in the form of a runtime library, but is supported by several key components implemented in the Linux kernel.

an even more efficient TSO-based system was proposed in [21] where chunks are as large as the regions between synchronization operations, i.e., commits occur only at synchronization operations. This approach better amortizes the costs of a commit.

Moreover, in all of these systems [5, 12, 21] commits are *synchronous* operations that require coordination among all threads in the program. This synchronous approach (Figure 3a) can lead to excessive waiting if threads do not naturally want to commit simultaneously.

An alternative *asynchronous* approach was proposed by [23]. Here, instead of a shared workspace, threads share a list of changes, and each thread applies these changes to their own workspace when appropriate. This allows threads to commit independently (Figure 3b), increasing parallelism. This optimization does not break TSO because updates still appear in a total order that all threads agree on. Indeed, in modern TSO processors memory fences are implemented in a similarly asynchronous fashion.

With CONSEQUENCE we build upon the approach of [23]. However, we observe that if the number of instructions between synchronization operations is small the fixed costs of committing cannot be effectively amortized. To counteract this effect CONSEQUENCE performs adaptive coarsening of these chunks, dynamically and deterministically selecting a target chunk size to optimize performance. Coarsening does not affect TSO consistency because a fence in TSO does not require that updates be made visible immediately, merely that writes made after the fence appear no sooner than writes made before it. Coalescing multiple commits and making them visible all at once is thus valid under TSO. We provide a more detailed discussion of adaptive coarsening in §3.

2.5 Implementing Thread Isolation

Many possible implementations of thread isolation have been explored by prior systems. For example, compiler instrumentation [5, 19] can be used to buffer a thread's writes into per-thread store buffers. This approach introduces considerable overhead for each write and requires recompilation of all programs and libraries. Hardware store buffers [12, 14, 18] have also been proposed to avoid these overheads. Virtual memory protection hardware on existing systems can also be used to intercept writes on a per-page basis using the `mprotect()` system call [21] or kernel modifications [23]. Using virtual memory techniques requires the use of processes instead of threads.

CONSEQUENCE's implementation of thread isolation relies on CONVERSION [23], a kernel-implemented version control system for main memory segments. CONVERSION allows a CONSEQUENCE thread (actually a process) to operate on a *local copy* of a memory segment in complete isolation, until it explicitly retrieves remote changes, or commits its local changes. CONSEQUENCE provides deterministic memory consistency for only the globals and heap segments.[2]

When a thread $T0$ begins a chunk, its globals and heap segments are updated to the latest available version of memory and each page is write-protected. Writes will trigger a copy-on-write page fault and a reference to the new page will be stored in a thread-local dirty list. On commit, a new version is created that contains the modified pages from the dirty list. If another thread $T1$ has committed since $T0$ began its chunk, it will have to update its page table to reflect those modifications. If there is a page-level conflict (i.e., $T0$ and $T1$ wrote to the same page), a basic byte-granularity merging mechanism enforces a last-writer-wins policy.

2.6 Deterministic Synchronization

The pthreads API defines synchronization operations that are used to ensure the correctness of parallel programs. Any usable determinism system must implement a deterministic version of this API. The main primitives provided are *mutual exclusion*, *condition variables*, and *barriers*. Additional primitives such as thread *create*, *join* and *exit* are also typically handled by deterministic systems. Previous work has varied greatly in its support and implementation of these primitives. In [21], the mutual exclusion implementation replaces all locks with a single global lock. This obviously negates any performance benefits gained by fine-grained locking code.

[2] Shared memory segments mapped by the user will not behave deterministically.

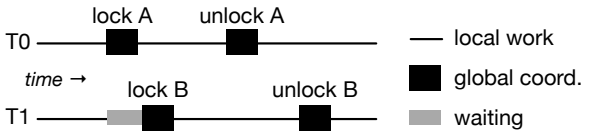

Figure 5: Global coordination is required only for synchronization operations, not for critical sections.

CONSEQUENCE provides efficient implementations of the common pthreads synchronization primitives listed above. Each synchronization primitive is implemented using both the deterministic logical clock (§2.1) and commit (§2.4) mechanisms. While highly relaxed consistency models like LRC can make commits into local operations, clock operations fundamentally require global coordination. Thus, in CONSEQUENCE, critical sections protected by different locks constitute local work that can execute concurrently, though the lock acquires and releases themselves require global coordination. In Figure 5, locking A and B invokes global coordination phases which must be serialized. However, between the lock and unlock operations, threads T0 and T1 can run concurrently.

CONSEQUENCE provides the first *blocking* implementation of a deterministic `mutex_lock()`, as opposed to the polling implementation in [25]. Our barrier implementation enables threads to concurrently retrieve and commit changes during the global coordination phase, a great improvement over an otherwise-serialized process. The implementation of our synchronization operations is described in detail in §4.

2.7 Ad Hoc Synchronization and Atomic Operations

For deterministic execution systems that perform commits at explicit synchronization operations [19, 21, 23], it can be difficult or impossible to support programs that communicate implicitly through shared memory. Take for example a thread $T0$ spinning on a synchronization variable $flag$ that will be set by another thread $T1$. If $T1$ commits its modification to $flag$ after $T0$'s chunk containing the ad hoc synchronization loop begins, $T0$'s chunk will never finish. This infinite loop occurs because no event will cause $T0$ to update its view of memory and see the update to $flag$ by $T1$.

One way to support this type of synchronization in CONSEQUENCE would be to set a limit on the number of instructions that can be executed within a chunk. Once that bound is hit, the thread must perform a commit operation. This would force $T0$ to break out of the spin loop and eventually see the changes made by $T1$.

However, the choice of a per-chunk instruction limit is application specific. For some programs, termination of a chunk early can severely degrade performance. For example, we found that some benchmarks required per-chunk instruction limits to be set to 1 billion instructions to achieve equivalent performance to an execution with the limit disabled. Of course, with higher instruction limits comes higher latency of communication with ad hoc synchronization. For now, we provide this mechanism in CONSEQUENCE but we conduct our evaluation (§5) with it disabled and leave efficient support of such programs as future work.

Atomic operations are another type of synchronization that is difficult for CONSEQUENCE to support. Because of the thread isolation we provide (§2.5), atomic writes will occur to thread-local memory and will lose the atomicity guarantees the programmer had expected. Atomic writes will be treated like any other write and are thus subject to the last-writer-wins merging policy. This can cause correctly synchronized programs to (deterministically) produce incorrect results when using CONSEQUENCE.

While CONSEQUENCE doesn't currently support atomic operations, we believe this can be easily resolved by using binary instrumentation. By replacing an atomic instruction with a CONSEQUENCE operation that acquires the token, performs the operation, and commits; we regain the atomicity.

3. Deterministic Adaptation and Other Means of Performance Improvement

A key insight behind the performance of CONSEQUENCE is that deterministic execution does not preclude adaptation on the part of the runtime system. Philosophically, a program running under varying conditions cannot be fully deterministic: while the output may be completely deterministic, the time to completion generally is not. Accordingly, we may adapt to the behavior of the running program, as long as the determinism of program state is preserved.

In principle, a great number of changes to the underlying execution environment are possible without affecting determinism, either in response to environmental changes or in response to (deterministic) program behaviors. Below, we discuss two such adaptations, followed by three other optimizations used in CONSEQUENCE.

3.1 Adaptive Coarsening

Fig. 6 illustrates the operation of CONSEQUENCE for a critical section. In the local work phase, all changes are made to the local copy. When a synchronization operation is reached, the thread enters the global coordination phase. It sleeps until it has the global minimum instruction count (GMIC) and can acquire the token, then retrieves any remote updates, and commits its own changes before acquiring the lock. It then releases the token and begins its critical section. The critical section constitutes another local work chunk, which is followed by another global coordination phase.

From this example, it is clear that though the work performed in the global coordination phase may be highly optimized, each deterministic synchronization operation will incur substantial overhead vs. its nondeterministic equivalent. This problem becomes particularly severe when the time between synchronization operations is brief, such that the local work is dwarfed by the global coordination surrounding it.

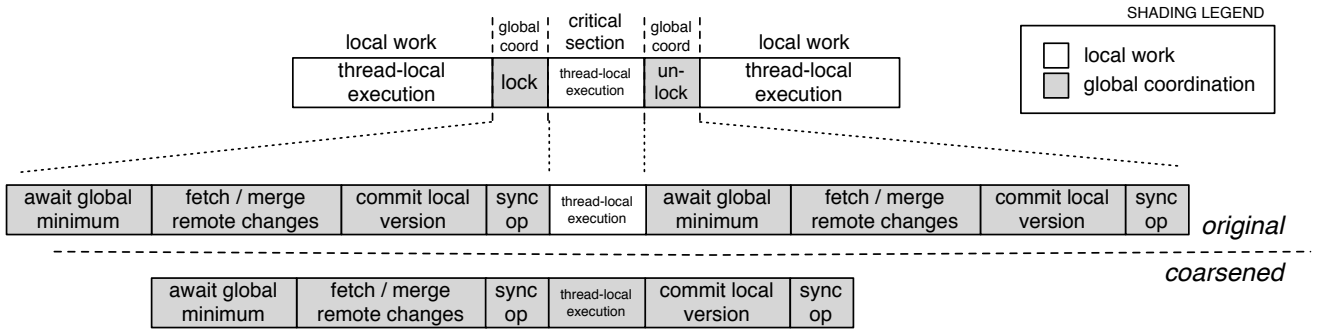

Figure 6: Program execution under CONSEQUENCE is divided into phases of local work and global coordination. In this example, a critical section is surrounded by two phases of global coordination, for the lock and unlock operations respectively. The lower part of the figure shows the result of coarsening on this example.

To address this problem, we propose to adaptively reduce the number of global coordination phases, through a technique we call coarsening. In essence, coarsening combines several global coordination phases into a single phase, which includes the intervening local work chunks. The lower part of Fig. 6 illustrates the effect of coarsening on a single critical section. Here, the coordination work required is substantially reduced, at the cost of moving what used to be (parallel) local work into the (serialized) global coordination phase. As discussed in more detail below, coarsening is not limited to pairs—any number of local work chunks and their surrounding coordination may be coarsened into a single, longer global coordination phase.

While this technique sounds straightforward, several challenges arise. First, in order to maintain TSO consistency, other threads are blocked at their next synchronization operation until the coarsened chunk has completed. This means that an overly ambitious coarsening policy, producing excessively long coarsened chunks, can significantly harm the overall system by causing excessive waiting of other threads. Second, the choice of when to begin a coarsened chunk must be an informed one. If CONSEQUENCE chooses to coarsen and the next chunk is very long (which cannot be known ahead of time) the result would be a net loss in performance. Finally, these decisions need to be done deterministically.

Our approach is to use an adaptive coarsening policy that estimates, for each synchronization variable, the next chunk length using an exponentially weighted moving average of earlier chunk lengths. One such estimate is maintained per lock, for use with coarsening $lock$ operations, and a thread-local estimate is maintained for use with coarsening $unlock$ operations. Coarsening proceeds until either (a) the estimated total coarsened chunk length exceeds a maximum threshold, or (b) a conditional variable or barrier operation is encountered.

We deterministically adapt the maximum chunk length at runtime, using a multiplicative increase, multiplicative decrease policy. Each time a thread $T1$ enters the global coordination phase, $T1$ compares its thread id to the id of the thread $T2$ that previously entered the global coordination phase. If $T1 == T2$, the maximum chunk length is doubled. Conversely, if $T1 \neq T2$, the maximum chunk length is halved. This allows each thread to individually adapt to current conditions. Because all decisions are based on chunk lengths and the token order, both of which are deterministic, coarsening maintains determinism.

3.2 Adaptive Counter Overflows

A thread's logical clock can be updated primarily in one of two ways: (1) when a thread finishes a chunk and the performance counter is read and (2) when a performance counter overflows, triggering an interrupt. Regarding the latter, the choice of overflow frequency is a trade-off between sequential overhead and logical clock accuracy. With a low frequency of overflows, a thread that is not the GMIC may wait longer to receive the notification that they are the new global minimum. However, if the frequency is too high, the cost of exception handling can become unwieldy (see [25] for further discussion).

Our solution to this problem is based on the observation that overflow frequency does not impact determinism at all. A lower or higher frequency may impact when a thread is notified in real time, but it has no bearing in logical time. Therefore, we design our logical clock module to adapt our overflow frequency using three simple rules. First, at the beginning of each chunk, the overflow is set to a conservative base value of 5,000 retired instructions. Second, if we are the GMIC, then at the beginning of each chunk and at each overflow we check to see if the next lowest logical clock is currently waiting to become the GMIC. If so, we set the performance counter to overflow just when our clock exceeds theirs. Third, if there is not a thread that we must notify then we double the number of instructions that will occur before the next overflow.

3.3 Thread Reuse for Fork-Join Programs

In order to support fork-join concurrency it is imperative for CONSEQUENCE to provide fast thread creation and tear-

```
1  void mutexLock(lock_t* l){
2   clockPause();
3   while(true){
4    waitToken();
5    if (lockAcq(l)){;
6     convCommitAndUpdateMem();
7     break;
8    }
9    else{
10    clockDepart();
11    insert(l->waitQueue, _tid);
12    releaseToken();
13    waitForRelease(l);
14   }
15  }
16  releaseToken();
17  clockResume();
18 }
```

Figure 7: mutexLock() implementation.

down. However, the use of processes with CONVERSION memory makes this a challenging goal. When a new thread is spawned the call is intercepted by CONSEQUENCE and instead a new process is forked (§2.5). Because CONVERSION memory is mapped as a private segment, each populated page-table entry in the CONVERSION segment must be copied into the child's page-table. Depending on the application, this can be a large number of entries to copy and adds a significant amount of latency to thread creation. To help mitigate this issue, CONSEQUENCE keeps a pool of threads that have recently finished executing. When spawning a new thread, if a thread is waiting in the pool that thread is reused, eliminating an expensive fork operation. The newly spawned thread will still have to update its view of memory to reflect what has been committed since it began waiting in the pool. However, this is typically a much cheaper operation than forking a new process.

3.4 User Space Reading of Performance Counters

Our deterministic logical clock module resides primarily in the kernel. One advantage of this approach is that performance counter overflows that identify a new GMIC can notify waiting threads directly from kernel space using shared memory, avoiding costly signals to user space for each overflow. There is added cost to this design however, as CONSEQUENCE would require system calls at the end of each chunk to read the counters and determine (and potentially notify) a newly-appointed GMIC thread. To avoid the system call latency for short chunks, we allow user space reading of the performance counters when executing a coarsened chunk.

3.5 Fast Forward

A thread may wait on a conditional variable or lock for an indefinite amount of time, causing its logical clock to become further and further behind the rest of the threads in the system. When the thread is finally woken and acquires the token, it may be the GMIC thread for a long time to come. To combat this, CONSEQUENCE "fast forwards" a thread's logical clock to the value of the logical clock of the last thread to release the token if that thread had a larger clock.[3]

```
1  void waitToken(){
2   while(!isGMIC() || token!=NULL){}
3   token=_tid;
4  }
5
6  void releaseToken(){
7   token=NULL;
8  }
```

Figure 8: A simplified implementation of token acquisition and release.

4. Synchronization Primitives

CONSEQUENCE supports deterministic versions of mutual exclusion, conditional variables and barriers. As in previous systems [19, 25], CONSEQUENCE uses the GMIC to deterministically order synchronization along with thread creation, exit and join events. Once a thread becomes the GMIC, it is eligible to acquire the *global token* which is required to perform any deterministic event. The token is useful as both an abstraction for maintaining determinism as well as a means of relaxing the traditional GMIC invariant, necessary to perform the coarsening optimization (see §3).

4.1 Mutual Exclusion

In Kendo [25], in order to ensure progress for others and to avoid introducing deadlocks, a GMIC thread that failed to acquire a lock would repeatedly increment their logical clock by some value until they were no longer the GMIC. This approach suffers from two problems: 1) the choice of a sensible value to add to the clock while polling requires program-specific tuning and 2) many polling requests to check whether there is a new GMIC thread to notify adds needless latency. A better approach would allow the GMIC thread to block and wait for the lock to be released while continuing to ensure progress for others. We accomplish this by adding the ability for a thread to remove itself from GMIC consideration through the $clockDepart()$ function.

The $mutexLock()$ implementation (shown in Figure 7) begins by pausing the clock and acquiring the token via the $waitToken()$ function (shown in Figure 8). If the lock is available (Figure 7, line 6), the thread commits its changes to memory and begins executing its critical section. However in the case of a held lock (Figure 7, line 10) the thread will remove itself from consideration for the GMIC and add itself to the lock's queue of waiters.

[3] Kendo [25] employed a similar mechanism to avoid excessive logical clock increments in their locking algorithm.

```
1  void mutexUnlock(lock_t* l){
2   clockPause();
3   waitToken();
4   lockRelease(l)
5   if (!queueEmpty(l->waitQueue)){
6    int tid=remove(l->waitQueue);
7    wakeupThread(tid);
8   }
9   convCommitAndUpdateMem();
10  releaseToken();
11  clockResume();
12 }
```

Figure 9: mutexUnlock() implementation.

Figure 9 shows the implementation of $mutexUnlock()$. After pausing the clock and acquiring the token, we check to see if there are any threads waiting for the lock (Figure 9, line 5). If so, we remove the thread from the wait queue and invoke $wakeupThread()$ which activates the thread using a (non-deterministic) conditional variable and adds the thread back into consideration for the GMIC. [4]

Note that, unlike in Kendo [25], $mutexUnlock()$ must acquire the token, as it performs a commit. Kendo assumes that applications are data-race-free and thus enforces no isolation between threads.

The techniques described above are also used to support deterministic conditional variables.

4.2 Barriers

CONSEQUENCE takes advantage of barrier semantics and CONVERSION's parallel commit feature to improve performance. This is not to be confused with prior synchronous deterministic systems, which used *internal* barriers to perform commits from different threads in parallel [5, 18].

In the case of multiple concurrent committers, CONVERSION may commit pages of memory in parallel through a two phase commit process. The first phase is done in serial, and determines the order in which changes to each page will be committed. In the second phase, pages are then merged and committed in parallel. The work done in phase two is several times larger than that of phase one, leading to better performance through parallelism. See [23] for more details.

To guarantee a deterministic ordering for our barrier, threads hold the token during phase one. After completing phase two in parallel, each thread waits at a non-deterministic (pthreads) barrier until all threads have finished committing. All threads then perform an update to get the latest version of memory.

[4] Our simplified code in Figure 9 does not handle one case of potential non-determinism. If the newly activated thread is the GMIC, then we must pass the token to them directly to avoid potential non-determinism with the thread that was the GMIC thread prior to activation.

5. Evaluation

Below, we study the performance of CONSEQUENCE in broad strokes, followed by several detail studies. Our results were produced on a computer with four 2.00GHz Intel Xeon E7-4820 8-core processors and 256GB of main memory, running Linux 2.6.37 with the CONVERSION and CONSEQUENCE kernel patches. For these experiments, Hyper-threading was turned off, and the frequency scaling governor was set to *performance*. Experiments were run 10 times per thread count, and mean deviation was within 20% for all benchmarks other than *linear_regression*, where the execution times can be below 500ms.

Figure 10 illustrates our main result: the runtime of two different variations of CONSEQUENCE, as well as DThreads [21] and DThreads with CONVERSION (DWC) [23]. [5] DThreads and DWC both use round-robin ordering, commits at synchronization operations and virtual memory-based isolation through either `mprotect()` (DThreads) or specialized kernel support (DWC). For each library we present results from 19 benchmarks[6] from the Phoenix, Parsec and SPLASH-2 benchmark suites, normalized to pthreads runtime.

To produce this plot, we measured the performance of each library (as well as pthreads) using 2–32 threads, and retained the corresponding best result. The plot shows the best library runtime, divided by the best pthreads runtime, for each benchmark.

Here, CONSEQUENCE-RR uses round-robin ordering based on synchronization operations (similar to that used in DThreads and DWC), while CONSEQUENCE-IC uses GMIC ordering (see §2.1). We see a maximum slow-down vs. pthreads of 3.9× using CONSEQUENCE with the deterministic GMIC ordering (compared to 12.5× with DThreads and 11.0× with DWC). The maximum slow-down is arguably the number to watch, as many of the benchmarks are "embarrassingly parallel" to start with and offer little or no insight into the performance of these libraries. While DThreads and/or DWC do outperform CONSEQUENCE-IC on some benchmarks, this difference is within the mean variation for all but two programs: *linear_regression* and *pca*. Further, all cases where a CONSEQUENCE variant is not the top performer occur for programs where the max slow-down is less than 2.0× pthreads across all libraries.

An interesting result occurs on the *reverse_index* and *dedup* benchmarks, where CONSEQUENCE is negatively impacted by its more sophisticated and flexible locking algorithm. Both DThreads and DWC treat each lock as a single global lock, which works well for programs where there is only a single lock and critical sections are short.

[5] Unfortunately a comparison with the relaxed consistency system RFDet [19] was not possible as the current implementation is provided without deterministic synchronization.

[6] Other benchmark programs from Parsec, Phoenix and SPLASH-2 are problematic for deterministic execution. This is mainly caused by the use of ad-hoc synchronization techniques. See [21] for details.

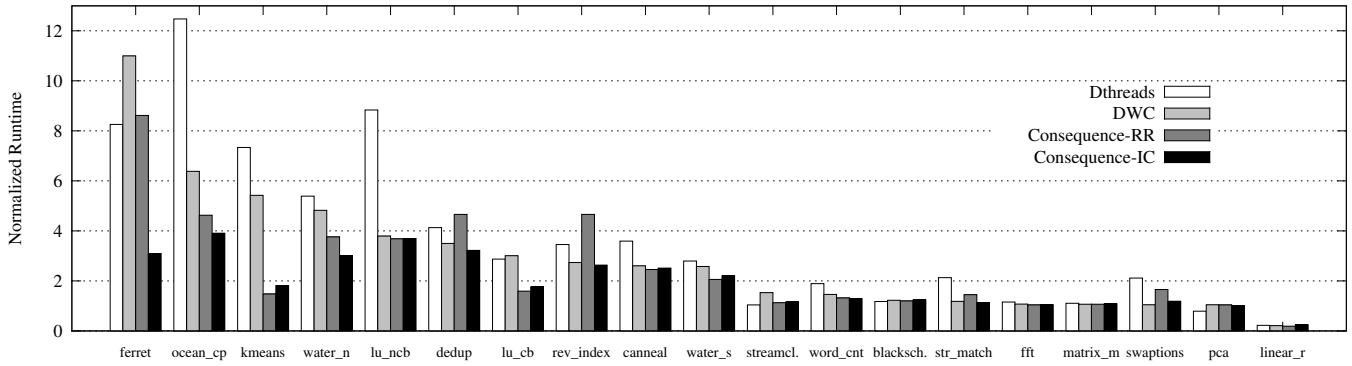

Figure 10: CONSEQUENCE, DThreads and DWC runtime normalized to pthreads runtime. Main result shown as CONSEQUENCE-IC. CONSEQUENCE-IC achieves an average of 2.8× and 2.2× improvement over DThreads and DWC (respectively) on the five most challenging benchmark programs.

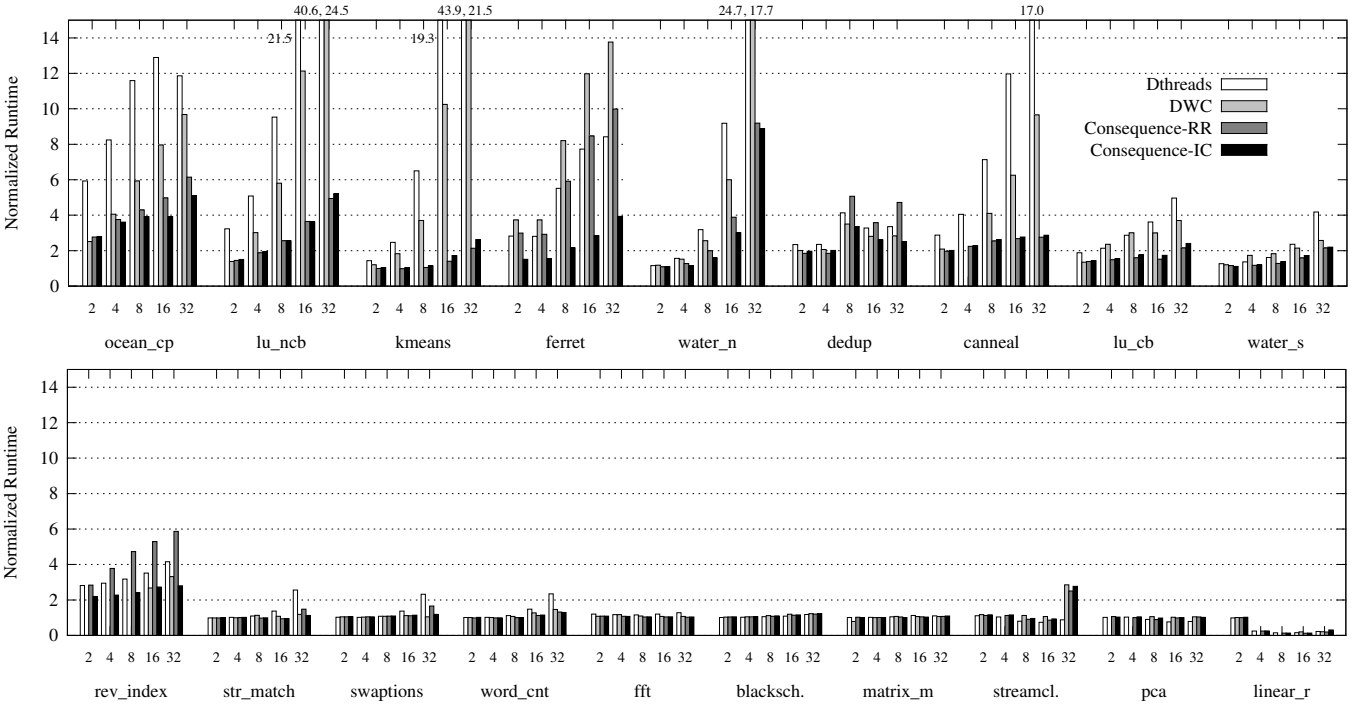

Figure 11: Performance when varying the number of threads for CONSEQUENCE, DThreads and DWC.

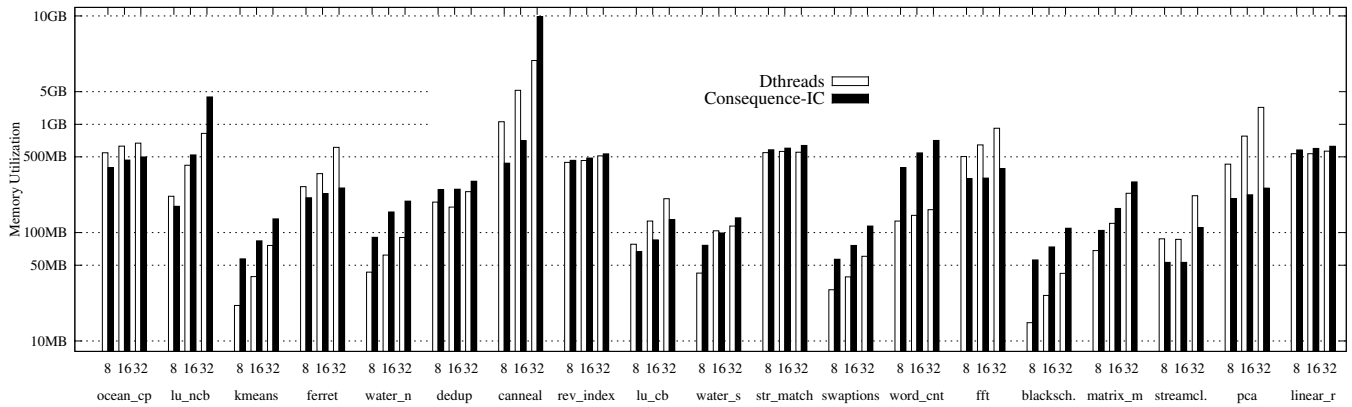

Figure 12: Peak memory usage for CONSEQUENCE and DThreads.

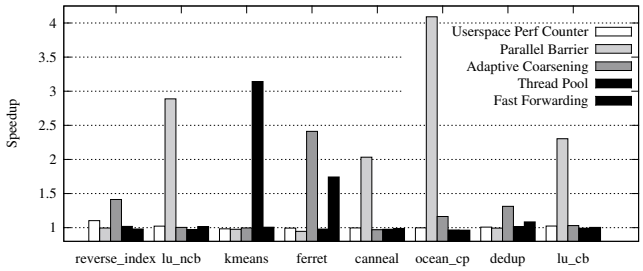

Figure 13: Speedup (higher is better) of various optimizations on a select group of benchmarks.

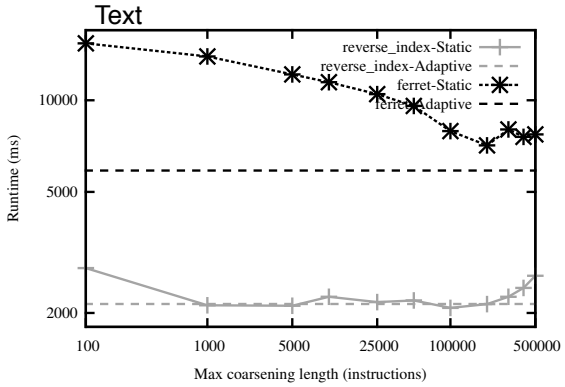

Figure 14: Comparison of Adaptive and Static Coarsening for *reverse_index* and *ferret*.

This causes CONSEQUENCE-RR performance to suffer, yet CONSEQUENCE-IC still manages to provide the best performance by generating a better deterministic ordering.

Figures 11–12 analyze the scalability of CONSEQUENCE with respect to thread count, in terms of runtime and memory usage. These reveal a severe DThreads and DWC scalability problem for six benchmarks: *ocean_cp, lu_ncb, ferret, kmeans, water_nsquared* and *canneal*. CONSEQUENCE also exhibits scaling difficulties, albeit much less severe.

On the *water_nsquared* benchmark with 32 threads, CONSEQUENCE experiences a drastic and unexpected performance hit. This occurs because each thread performs many fine-grained lock acquisitions with short critical sections, which become coarsened with CONSEQUENCE. Because a thread holds on to the token throughout the entire coarsened chunk, other threads may be blocked, leading to the scalability issue seen here.

In terms of memory usage (see Figure 12), DThreads and CONSEQUENCE appear evenly matched. Two notable exceptions are *canneal* and *lu_ncb* at high thread counts. This is caused by a high volume of page allocation/freeing such that the single-threaded CONVERSION garbage collector cannot keep up. A multi-threaded collector would solve this issue.

5.1 Sources of Performance Improvement

As described in §3, CONSEQUENCE includes a number of performance optimizations, aimed at reducing the cost of common concurrency patterns. To better understand how these various parts contribute to the performance of CONSEQUENCE, we evaluate CONSEQUENCE-IC with and without each of these optimizations.

Figure 13 shows the performance improvement (higher is better) contributed by each of five major optimizations, on eight of the most difficult benchmarks. While all optimizations demonstrate some amount of improvement, we find that user space performance counter readings contribute very little to the overall performance. For *ferret*, being one of the hardest of the benchmark programs to perform well on, both the adaptive coarsening and fast forward optimizations provide major speed improvements. For *ocean_cp, lu_ncb, canneal* and *lu_cb* the parallel barrier provides the main source of our speedup. Note that these numbers represent the performance difference in CONSEQUENCE with and without each optimization, not a performance improvement with respect to pthreads or DThreads.

Adaptive Coarsening stands out as one of the most successful optimizations, which merits additional study. Figure 14, shows the effect of coarsening level (x-axis) on the runtime (y-axis, lower is better) of the *reverse_index* and *ferret* benchmarks. It is clear that the coarsening level has a significant effect of runtime performance, even when set statically. With adaptive coarsening, each thread selects its own coarsening level, allowing adaptive coarsening to outperform even the best statically chosen coarsening level.

5.2 What is Holding CONSEQUENCE Back?

Figure 15 provides a breakdown of where CONSEQUENCE spends time for a selection of benchmarks. The benchmarks can be divided between "embarrassingly parallel" (*string_match*), barrier-heavy (*ocean_cp, lu_cb, lu_ncb, canneal, water_nsquared and water_spatial*) and those that experience other determinism-related overhead (*kmeans, ferret, dedup and reverse_index*). For *ferret*, the first thread spawned is presented on its own (*ferret_1*) because it exhibits a radically different synchronization pattern than the rest (*ferret_n*).

For the barrier-heavy programs (e.g. canneal), DWC typically has a much higher execution time (see Figure 11) and a large percentage of that time is spent waiting on others. This is because the DWC barrier commits are done serially and the amount of memory that must be committed is typically high. In CONSEQUENCE-IC, much of the commit work is done in parallel so the wait time (shown as barrier_wait[7]) is greatly reduced. With less wait time, the remaining overhead is spread between copy-on-write page faults and commit operations in CONVERSION. For *canneal* and *lu_ncb* the time spent in CONVERSION is higher because threads tend to write

[7] We separate barrier_wait and determ_wait time for CONSEQUENCE because the time spent waiting at the barrier is not impacted by deterministic ordering.

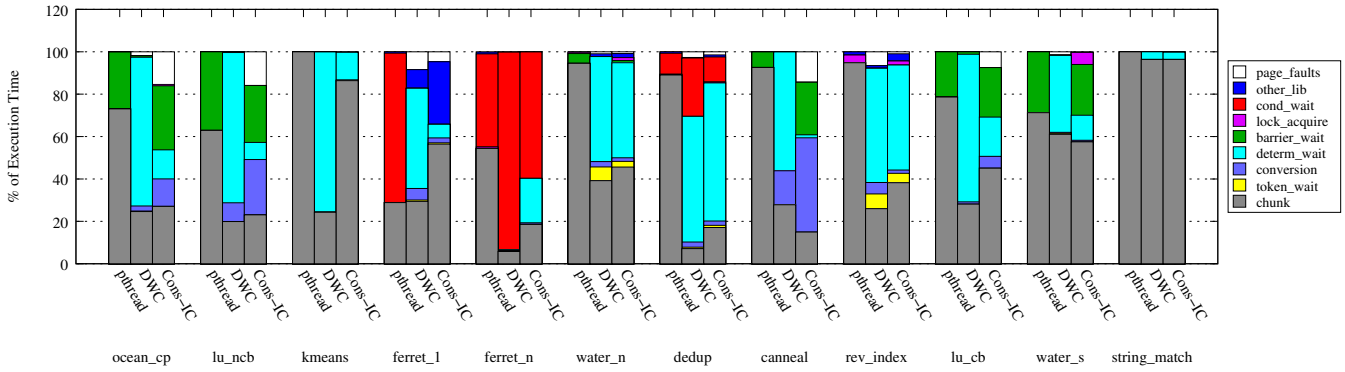

Figure 15: Breakdown of the time spent in each benchmark with pthreads, DWC and CONSEQUENCE-IC using 8 threads.

to the same page in isolation, leading to a larger number of byte-granularity merges.

The *ferret* program provides an interesting challenge for deterministic execution. The first phase of the pipeline (shown as *ferret_1*) performs a high volume of lock acquisitions with short chunk sizes, while the rest of the threads oscillate between executing longer chunks and waiting at conditional variables. For good deterministic performance, it is important to (1) provide an ordering that allows the first thread to perform synchronization unimpeded, and (2) reduce the cost of each synchronization operation. CONSEQUENCE-IC provides (1) through GMIC ordering and (2) through adaptive coarsening. This results in more time spent executing chunks and less time spent waiting or performing page faults.[8]. CONSEQUENCE-IC still experiences a large amount of general library overhead on *ferret_1* which is mainly caused by reading the performance counters and other work done between chunks.

5.3 Memory Propagation for Relaxed Models

For some benchmark programs like *canneal* and *lu_ncb*, the amount of memory that must be propagated between threads can degrade CONSEQUENCE performance. Therefore, it is worth investigating whether relaxed memory models can perhaps alleviate this burden on deterministic execution.

An LRC-based memory model can reduce total memory propagation by allowing commits to be "point to point." More specifically, memory is propagated between threads using the *happens-before* relation. An operation a is said to have happened before b if (1) a occurs before b within the same thread of execution, or (2) a and b are release and acquire operations on the same synchronization variable.

To evaluate the reduction in memory propagation that a relaxed memory model can offer, we modified CONSEQUENCE to track the happens-before relation. This was achieved by adding a vector clock to each thread, synchronization variable and committed page. For each acquire operation (lock, conditional wait, thread creation/join and barrier wait)[19], we compute which pages would need to be propagated along happens before edges.

Figure 16 compares the total number of pages propagated under TSO (CONSEQUENCE) and the expected number for an LRC-based system across 12 benchmarks that perform at least 10K page updates. While the LRC system can reduce total memory propagation, the reduction is just 21% when averaged across all benchmarks considered. For some benchmarks like *canneal*, the use of barriers limits the gains from using a relaxed consistency model.

6. Discussion and Future Work

While CONSEQUENCE makes significant improvements over prior systems, there are some types of workloads that CONSEQUENCE (and all current TSO deterministic systems) will struggle to support efficiently. One such class of programs is those that use fine-grained locking with relatively short chunk sizes. In CONSEQUENCE, each lock and unlock will be totally ordered and will require a global commit operation. This significantly impacts scalability, as was seen in the *water_nsquared* benchmark program.

This is one class of programs where relaxed consistency can make an improvement over TSO. In an LRC system, the lock acquisitions and releases must still be totally ordered but the commit operations can be done in parallel for distinct locks. Therefore, even if the total amount of memory that must be propagated between threads for the two consistency models is roughly the same, the LRC system may exhibit better scalability. In our future work, we look to better support these types of workloads while maintaining TSO.

7. Related Work

The Kendo [25] and Deterministic Shared Memory Multiprocessing (DMP) [13] systems first showed how to provide determinism for general multi-threaded programs: Kendo was a pure-software system that leveraged performance counters and DMP used hardware support to provide determinism even for programs with data races. Follow-on work

[8] Coarsening can reduce the total number of page faults if two chunks in a coarsened chunk write to the same page.

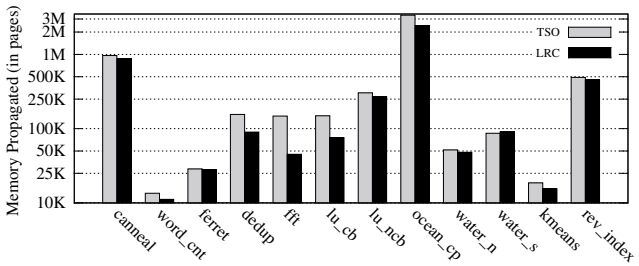

Figure 16: Total pages propagated under TSO (CONSEQUENCE) and the expected number for an LRC-based system. (log scale y)

has shown a variety of ways to optimize the performance overheads of determinism, e.g., through compiler optimizations [5], relaxed memory consistency [2, 11, 12, 14], existing hardware support for virtual memory [3, 7, 21, 29], and eliminating the synchronous implementation of commits present in prior determinism systems [23]. Segulja and Abdelrahman [27] measure the performance cost of enforcing a deterministic logical clock and find it to be less than 2x across a range of benchmarks and runtime perturbations, showing that determinism needn't be fundamentally expensive. Recently, the RFDet system [19] demonstrated the performance benefits of memory consistency optimizations with its deterministic implementation of LRC [20].

There is also a large body of work on programming languages that enforce deterministic parallelism. These languages ensure determinism by construction, e.g., via data-parallel functional programming models [15], annotations to identify opportunities for parallelism in sequential code [26], stream-based programming models [31], or type-and-effect systems for imperative languages [8]. Blelloch et al. [28] describe the *deterministic reservations* programming discipline for scheduling potentially conflicting parallel operations in a deterministic way, showing good speedups on a range of parallel algorithms. The Deterministic Galois system [24] shows how to enforce deterministic reservations automatically, guaranteeing deterministic results for all Galois programs. While programs written in these deterministic languages often show good performance and scalability, CONSEQUENCE provides determinism for legacy binary programs and does not require that programs be rewritten or even recompiled.

Another vein of work investigates limiting the nondeterminism of multi-threaded programs through *stable multi-threading*. Instead of forcing every execution of a particular input to deterministically follow the same schedule, a small set of schedules are found that any input can nondeterministically follow. The flexibility to nondeterministically choose a schedule at runtime typically allows for higher performance. Early work on stable multi-threading relied on sophisticated program analysis [6, 16, 17] to discover the set of permitted schedules. More recently, the Parrot [10] system eschews such analysis in place of programmer annotations that identify where schedule flexibility is needed. Parrot's limited nondeterminism has been shown to amplify the power of verification techniques like model checking. While CONSEQUENCE does not require programmer annotations to achieve good performance, the authors of [10] observe that stable multi-threading and determinism are not mutually exclusive. In future work we hope to better understand what trade-off exists between stability and determinism.

8. Conclusion

With CONSEQUENCE we demonstrate that highly relaxed memory consistency is not necessary for high-performance deterministic execution; CONSEQUENCE achieves similarly good performance while retaining the stronger TSO consistency model. The performance benefits of relaxed consistency, which allow memory fences to be implemented as local operations, are attenuated by the fact that deterministic synchronization still requires global coordination. We identify performance optimizations for the TSO consistency model that reduce the cost of commits and that exploit adaptivity to program behavior while maintaining determinism. We plan to release the source code for our system to enable future researchers to duplicate and build upon our results.

9. Acknowledgments

This work was supported by the National Science Foundation under grants CNS-1320235 and XPS-1337174.

References

[1] Sarita Adve. Data races are evil with no exceptions. *Communications of the ACM*, 53(11):84, 2010.

[2] Amittai Aviram, Bryan Ford, and Yu Zhang. Workspace consistency: A programming model for shared memory parallelism. In *Workshop on Determinism and Correctness in Parallel Programming*. 2011.

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

[4] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2011.

[5] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. 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)*. 2010.

[6] Tom Bergan, Luis Ceze, and Dan Grossman. Input-covering schedules for multithreaded programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. 2013.

[7] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: safe multithreaded programming for C/C++. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '09*. 2009.

[8] Robert Bocchino, Mohsen Vakilian, Vikram Adve, Danny Dig, Sarita Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, and Hyojin Sung. 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*. 2009.

[9] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. 2007.

[10] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013.

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

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

[13] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: deterministic shared memory multiprocessing. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS '09)*. 2009.

[14] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. RCDC: a relaxed consistency deterministic computer. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. 2011.

[15] Guy Blelloch. NESL: a nested data-parallel language. Technical Report CMU-CS-92-103, Carnegie Mellon University, Pittsburgh, PA, 1992.

[16] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. 2010.

[17] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011.

[18] Hadi Jooybar, Wilson W.L. Fung, Mike O'Connor, Joseph Devietti, and Tor M. Aamodt. GPUDet: a deterministic GPU architecture. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*. 2013.

[19] Kai Lu, Xu Zhou, Tom Bergan, and Xiaoping Wang. Efficient deterministic multithreading without global barriers. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2014.

[20] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: distributed shared memory on standard workstations and operating systems. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*. 1994.

[21] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011.

[22] Li Lu and Michael L. Scott. Toward a formal semantic framework for deterministic parallel programming. In *Proceedings of the 25th International Conference on Distributed Computing*. 2011.

[23] Timothy Merrifield and Jakob Eriksson. Conversion: multi-version concurrency control for main memory segments. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 2013.

[24] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. Deterministic galois: On-demand, portable and parameterless. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. 2014.

[25] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems - ASPLOS '09*. 2009.

[26] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483545, 1998.

[27] Cedomir Segulja and Tarek S. Abdelrahman. What is the cost of weak determinism? In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. 2014.

[28] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: The problem based benchmark suite. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. 2012.

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

[30] V.M. Weaver and S.A. McKee. Can hardware performance counters be trusted? In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. 2008.

[31] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: a language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*. 2002.