What is the Cost of Weak Determinism?

Cedomir Segulja and Tarek S. Abdelrahman
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
seguljac@eecg.toronto.edu, tsa@eecg.toronto.edu

ABSTRACT

We analyze the fundamental performance impact of enforcing a fixed order of synchronization operations to achieve weak deterministic execution. Our analysis is in three parts, performed on a real system using the SPLASH-2 and PARSEC benchmarks. First, we quantify the impact of various sources of nondeterminism on execution of data-race-free programs. We find that thread synchronization is the prevalent source of nondeterminism, sometimes affecting program output. Second, we divorce the implementation overhead of a system imposing a specific synchronization order from the impact of enforcing this order. We show that this fundamental cost of determinism is small (slowdown of 4% on average and 32% in the worst case) and we identify application characteristics responsible for this cost. Finally, we evaluate this cost under perturbed execution conditions. We find that demanding determinism when threads face such conditions can cause almost 2x slowdown.

Categories and Subject Descriptors
D.1.3 [Programming Techniques]: Concurrent Programming—Parallel Programming; D.2.8 [Software Engineering]: Metrics—Performance Measures; D.3.4 [Programming Languages]: Processors—Run-time Environments

Keywords
determinism; deterministic execution; multithreading

1. INTRODUCTION

One of the main reasons why parallel programs are demanding to write, debug, and test is the nondeterminism of program output and/or the way this output is produced [19, 31]. Recently, there has been a stream of systems that suppress nondeterminism by forcing threads to access shared resources, i.e., communicate, in a deterministic manner. They propose different thread synchronization orders, or schedules, and also incur different overheads (Table 1).

<table>
<thead>
<tr>
<th>System</th>
<th>Schedule</th>
<th>Weak vs. Strong</th>
<th>Max. Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grace</td>
<td>serial</td>
<td>weak</td>
<td>3.6x</td>
</tr>
<tr>
<td>Dthreads</td>
<td>round-robin</td>
<td>strong</td>
<td>4x</td>
</tr>
<tr>
<td>Conversion</td>
<td>round-robin</td>
<td>strong</td>
<td>5x</td>
</tr>
<tr>
<td>Parrot</td>
<td>round-robin</td>
<td>weak</td>
<td>3.8x</td>
</tr>
<tr>
<td>Kendo</td>
<td>dynamic</td>
<td>weak</td>
<td>1.6x</td>
</tr>
<tr>
<td>RCDC</td>
<td>dynamic</td>
<td>strong</td>
<td>1.7x</td>
</tr>
<tr>
<td>RFDet</td>
<td>dynamic</td>
<td>strong</td>
<td>2.6x</td>
</tr>
<tr>
<td>DMP</td>
<td>hybrid</td>
<td>strong</td>
<td>1.7x</td>
</tr>
<tr>
<td>CoreDet</td>
<td>hybrid</td>
<td>strong</td>
<td>10x</td>
</tr>
<tr>
<td>Calvin</td>
<td>hybrid</td>
<td>strong</td>
<td>1.7x</td>
</tr>
</tbody>
</table>

Table 1: State-of-the-art deterministic systems. The overheads, reported in [5] and in the respective papers, show that the authors themselves find that determinism comes at a cost.

Some of the above systems provide determinism only for data-race-free programs, which is commonly referred to as weak determinism [33]. Others secure determinism even in the presence of data races and this stronger guarantee is therefore named strong determinism [33]. In addition to this difference, these systems come in both hardware [14, 16, 22] and software [7, 33, 4, 25, 13, 29, 26] flavors, and enforce determinism using a variety of techniques, including speculation [7, 14], buffering [14, 25, 13, 29, 26] and version control [29]. This makes it difficult to understand if one system is better than another due to the schedule being enforced or due to differences in implementations. In other words, it is unclear whether the performance degradation under deterministic systems stems from forcing the execution to be deterministic, or from the enforcement mechanisms being used. Thus, we pose the question: implementation overhead set aside, what is the fundamental cost of determinism?

In this paper, we seek to answer this question in the context of data-race-free programs, that is, for weak determinism. Guaranteeing deterministic thread communication for this class of programs reduces to ensuring that synchronization operations are executed deterministically. In the rest of this paper, we use the term “determinism” to simply refer to weak determinism, unless noted otherwise.

We run a data-race-free version of SPLASH-2 [10] and PARSEC [8] benchmark suites on a dual chip 16-core Xeon E5-2660 machine and empirically study the cost of making synchronization deterministic under a given schedule. We use a schedule-record-replay framework to divorce the implementation overhead of generating a schedule from the fundamental impact of enforcing the schedule. The framework also allows us to evaluate the performance of deterministic execution across different execution environments by perturbing the execution in a controlled manner.
Our study has three parts. In the first, we quantify to what extent does nondeterminism occur in program execution. In the process, we identify previously unreported sources of nondeterminism, like lazy binding, initialization of C++ static variables and compiler vectorization. Our analysis confirms the common belief that thread synchronization is the most significant contributor to nondeterminism in data-race-free programs.

In the second part of our study, we evaluate the performance impact of enforcing a schedule on program execution. By employing our schedule-record-replay framework, we find a lower bound on the execution time that can be achieved with any deterministic execution that enforces the same schedule. The analysis reveals that the cost of deterministic execution is smaller than previously thought: only 4% on average and never greater than 32% increase in the run time compared to a nondeterministic execution. We further pinpoint program characteristics that are relevant for identifying whether or not a program will incur significant performance degradation when executed with a schedule.

In the last part of our study, and again using the same framework, we evaluate the cost of enforcing a schedule across multiple execution environments (e.g., with the presence of context switches, background processes and execution on asymmetric architectures). We find that insisting on the same schedule when threads face uneven conditions like unbalanced system loads or an asymmetric architecture can slow down execution by as much as 2x.

While the technique we use to measure the cost of determinism applies only to data-race-free programs, we believe that the insights that our work provides are also of interest to systems that supply determinism for arbitrary parallel programs (i.e., strong determinism). First, for both weak and strong determinism, the order in which threads execute synchronization operations (e.g., lock acquire/release, condition variable signal/wait) must be resolved deterministically. Second, although strongly deterministic systems must additionally deal with data races, recent advances in this area have demonstrated that strong determinism requires no extra program serialization in addition to the serialization caused by deterministically ordering synchronization operations [29, 26].

Contributions. To the best of our knowledge, our study of data-race-free benchmarks is the first to:

- Quantify the impact of various sources of nondeterminism on program execution on a real machine.
- Determine the fundamental performance cost of enforcing weak determinism and correlate it to application characteristics.
- Evaluate the fundamental cost of requiring the same deterministic execution across a range of environments.

The rest of this paper is organized as follows. Section 2 defines determinism, discusses sources of nondeterminism, and summarizes the existing approaches of enforcing deterministic thread synchronization. Section 3 describes our schedule-record-replay framework. Section 4 details the platform, benchmarks, and the metrics we use in our study. Section 5 presents and analyzes our empirical results. Section 6 presents related work, and Section 7 concludes.

2. BACKGROUND

2.1 What is Determinism?

Determinism is a program property that requires observing the same output whenever the program runs with the same input. However, depending on what one assumes program output and program input to mean, many flavours of determinism arise. These flavours apply to both data-race-free and arbitrary parallel programs, i.e., they are orthogonal to the notion of weakly vs. strongly deterministic implementations. Here we highlight common definitions of determinism and state the one we adopt in this paper.

2.1.1 Program Output

One often considers only the end program result to be the output (external determinism [19]). However, while debugging, the intermediate values or even instruction operands might be considered as the program output. Determinism is then defined more rigorously as the repeatability of the sequence of instructions each thread executes, along with the values of operands used by each instruction (internal determinism [19]). Between these two extremes, other formulations are possible; an excellent discussion of plausible definitions of determinism is given by Lu and Scott [27]. In particular, the executions are SyncOrder deterministic if they, in addition to agreeing in the final result, contain the same synchronization operations, executed by the same threads in the same order.

For both data-race-free and arbitrary parallel programs, SyncOrder determinism allows for some differences between executions (changes in the addresses of variables via ASLR as described in Section 2.2.2 or benign data changes due to data races), as long as they have no effect on synchronization order and program result. However, it still limits the number of possible thread interleavings and hence greatly increases test coverage [13].

All existing deterministic systems provide determinism of at least SyncOrder semantics, and they deterministically resolve the order of synchronization operations. Hence, when assessing the lower bound on the cost of determinism, we adopt SyncOrder determinism and require threads to synchronize in a deterministic manner.

2.1.2 Program Input

A natural definition of the input includes supplied program arguments and file/network input. However, certain software and hardware parameters may also influence the program output (e.g., process memory layout or the size of hardware structures). Practical approaches to determinism can be classified in three groups according to how they cope with these parameters. One class considers these parameters to be program input as well, by silently assuming they do not vary across executions. Another class forces these parameters to be constant (e.g., system-level determinism [2, 6] by providing a custom OS). Finally, some approaches actually do guarantee determinism when certain system parameters vary (e.g., unbounded determinism [22] by providing determinism across micro-architectural variations).

We believe that it is more useful to provide determinism across a wide range of system setting. Thus, we consider system parameters to be a source of nondeterminism rather than a program input.
2.2 What Causes Nondeterminism?

Here we describe causes for executions of the same program not to be identical in the strict sense of internal determinism. Most of these sources are already known, however, we do identify some sources not previously reported and that we discovered while running our benchmarks.

While each of the sources we discuss instigates nondeterminism on its own, it is important to note that these sources do interact and hence are not completely orthogonal.

2.2.1 Thread Communication

One well-known source that leads to nondeterminism is thread communication, i.e., accesses to shared resources by different threads. Small variations in the execution environment, such as the state of hardware caches, TLBs or the presence of other applications, may make accesses to shared resources by different threads occur in different orders across multiple program runs. This, in turn, can cause threads to execute different instruction sequences and possibly affect program output.

Prior work has identified accesses to shared program variables [19, 51] and shared library resources (e.g., variables and I/O objects) [29, 4] as potentially occurring in different orders, leading to nondeterminism. However, we have also discovered accesses to other shared resources, not introduced by programmers and library writers, as giving rise to nondeterminism. These accesses are manifested through subtle compiler/run-time interactions. For example, the runtime symbol resolution (also known as lazy binding) [17], speeds up program load time, but it also causes the first thread to reach a given shared library call to do the extra work of symbol resolution. Since the “first” thread can be different across program runs, so can the number of instructions executed by each thread, making executions not internally deterministic. C++ local static variables initialized with a non-compile-time constant also cause the first thread to reach the corresponding function to do the extra work, again making executions nondeterministic.

2.2.2 Process Address Space Layout Variations

Address space layout randomization (ASLR), changes in Linux environment variables, modification of shared libraries, preloading additional libraries (e.g., to debug segmentation faults using {libSegFault.so}), cause changes in the addresses of program instruction and data across multiple runs, making them not identical by the strict definition of internal determinism.

Additional variability may arise when the address variations are coupled with code that inspects addresses. For example, compiler auto-vectorization, which takes advantage of the vector units available in modern processors (e.g., SSE/AVX on x86 and AltiVec/VMX on PowerPC), is implemented via loop peeling and/or loop versioning to dynamically at run-time guarantee aligned accesses [9]. These transformations result in code in which the total number of executed instructions depends on the input data alignment, and hence, could change due to the address space layout variability. Similarly, certain library (or compiler built-in) functions (e.g., memcpy and memset), follow different code paths based on input alignment in order to use the widest data type transfers supported by the hardware. Finally, the application code itself can be dependent on memory addresses.

2.2.3 System and Library Dependences

Certain library and system calls are nondeterministic. Some are such by construction (e.g., rand or gettimeofday), others through their semantics. For example, Linux’s read system call is allowed to return smaller number of bytes than requested [14]. Different runs of the same program that must read a certain number of bytes may execute a different number of read calls.

2.3 How to Achieve Determinism?

The bulk of work on determinism seeks to ensure that threads communicate in a deterministic manner [14, 55, 4, 22, 16, 26, 13, 29, 26]. For data-race-free programs, this problem reduces to guaranteeing that synchronization operations are executed deterministically. While data races add complexity to a deterministic system, it has been recently shown that race access can be made deterministic without adding any program serialization in addition to the one caused by deterministically ordering synchronization operations [29, 26].

Thread synchronization order is described formally using the notion of a schedule. A schedule is a directed acyclic graph consisting of nodes that represent executed synchronization operations of a multi-threaded program. The edges of a schedule denote the causal partial relationship between synchronization operations, i.e., the Lamport’s happened-before relation [24].

The schedules generated and enforced by the state-of-the-art deterministic systems (Table 1) are best described by the notion of a turn. At any given time during execution, it is only one thread’s turn and only while a thread has the turn can it execute synchronization operations. Depending on when a thread passes its turn, the following schedules emerge.

In the serial schedule [7] a thread holds on to its turn until the end of its execution, effectively imitating an execution in which threads are executed synchronously in program order (i.e., as function calls). In the round-robin schedule [25, 4] a thread passes its turn after each executed synchronization operation. In the dynamic schedule [33, 16, 26] the turn is passed on each tick of a per-thread deterministic logical clock. The original proposal by Oliszewski et al. [33] uses store instructions to maintain logical clocks, i.e., a thread passes its turn upon executing one memory store. The hybrid schedule [14, 4, 22] uses the logical clock to break down the execution of each thread into quanta and enforces the serial schedule during each quantum – a thread passes its turn only when it reaches the end of one quantum. A thread’s quantum ends when it executes N ticks, where N is an input parameter of this schedule. Additionally, a thread also could end its quantum after executing a synchronization operation – this variation is known as the “reduced serial mode” [4].

While primarily introduced to handle programs with data races, we note that the hybrid schedule can be seen as a generalization of the serial schedule (when N = ∞), the round-robin schedule (when N = N and a quantum also ends upon executing a synchronization operation) and the dynamic schedule (when N = 1). We utilize this aspect when building a framework for finding the cost of enforcing these schedules.

It is important to note that the notion of a turn is conceptual, and many executions satisfy the same schedule. For example, the implementation of the serial schedule executes
threads speculatively to regain parallelism while respecting the serial schedule [7]. The systems using the dynamic schedule avoid the overhead of turn passing and force a thread to wait when executing a synchronization operation until its logical clock becomes the lowest across all threads [33]. Nonetheless, the existing systems suffer overheads not fundamental to deterministic synchronization (e.g., rollbacks in case of speculation, or forcing a total order of synchronization operations). While we use the concept of a turn, we ensure that threads wait only when necessary to conform to the partial order of synchronization dictated by a given schedule.

3. OUR FRAMEWORK

We use a schedule-record-replay framework (Figure 1) to evaluate the cost of making synchronization operations in data-race-free programs deterministic under a given schedule. When evaluating the performance of a program under a specific schedule, we execute it twice: the first time to generate the schedule, and the second time to enforce the schedule. The key concern is to impose the schedule as efficiently as possible, in order to obtain the best execution time achievable with that schedule.

The scheduler has a "knob" that allows us to select the scheduling algorithm to be evaluated. During this run, the recorder captures the scheduled order of synchronization operations and dumps it to a file. In the second run, the replacer imposes the saved recorded schedule, guaranteeing the same deterministic execution as in the first run, but without the scheduler's overhead.

The replacer should force threads to wait only when absolutely necessary for the schedule to be respected and it should do so with little overhead. If we can achieve that, then the run time of the second run is close-to-best time that can be attained with any system enforcing the same schedule.

The framework also has a perturber component, as shown in Figure 1. The perturber, similar to the scheduler, has a knob that allows us to choose the execution environment in which the performance of a given schedule is measured. We discuss each component of this framework in turn.

3.1 Scheduler

The scheduler intercepts Pthreads synchronization operations and orders them by implementing the schedules described in Section 2.3 Using the insight that the hybrid schedule can be seen as a generalization of all the other schedules, our scheduler implements the hybrid schedule with parameterizable quantum duration, and by varying the quantum duration we get the serial, round-robin, and the dynamic schedule.

We use the hardware performance counters and Linux’s perf_event API to implement logical clocks. For the dynamic schedule, we implemented two versions of logical clocks: one that uses only the number of store instructions [33], and one that uses the number of all instructions [16, 20]. We refer to the resulting schedules as dynamic-S and dynamic-A, respectively. For the hybrid schedule, we use the number of all instructions to implement logical clocks, set the phase duration to 100,000 instructions, and do not use the reduced serial phase. These settings have been found previously to on average give the best performance with the CoreDet [4] implementation of the hybrid schedule [25].

The determinism of logical clocks, necessary to ensure a deterministic schedule, can be compromised when there are other sources of nondeterminism besides thread synchronization (e.g., lazy binding, auto-vectorization with ASLR, and nondeterminism in library calls). Thus, to ensure correctness of the dynamic and hybrid schedule, we pause logical clocks during the execution of nondeterministic library functions and disable lazy binding and auto-vectorization. This is done only during the record run, and not during the replay run.

The use of logical clocks that tick on each instruction can compromise the determinism of schedules. This is because performance counters report slightly different number of instructions even for identical instruction sequences, as reported in [39]. However, our framework is unaffected by this issue. We use the scheduler only during the recording run, and then enforce the recorded order with the replacer, making the replayed execution deterministic. This is one advantage of our schedule-record-replay framework that decouples schedule generation from schedule enforcement. Indeed, we verify this deterministic execution using a series of microbenchmarks and a data-race-free version of RACEY [21].

3.2 Recorder

The recorder captures the scheduled order of synchronization operations. Due to space considerations, we focus our description on critical sections, but our framework supports the other commonly used synchronization primitives (barriers and condition variables) as well.

When a critical section is executed, the relevant information (the identifiers of the thread executing the critical section and of the lock that protects the critical section) are atomically appended to an array shared by all threads. Although this creates an over-constrained, total order of synchronization operations, the schedule (i.e., the partial order sufficient to guarantee repeatability) is easily reconstructed based on the thread and lock identifiers: a synchronization operation A must occur before a synchronization B if and only if A precedes B in the shared array and A and B have
the same lock or thread identifier. The potential application slowdown during the recording run is not an issue, since the recorder is used solely to capture a schedule, while it is the task of the replayer to efficiently execute this schedule.

3.3 Replayer

During replay, the schedule is represented with multiple arrays, one per thread. Each byte of a thread’s array corresponds to one critical section that the thread will execute and its value is the id of the next thread to enter a critical section protected by the same lock. This encoding supports programs with up to 256 threads. A thread traverses its array during the execution, consuming one byte for each executed critical section. Thus, laying out the elements of a thread’s array in the order in which the thread will enter critical sections (which is known from the recording run) enables good cache behavior.

We use the replay lock and unlock operations, shown in Figure 2 that replace corresponding Pthreads routines and are inspired by ticket locks [25]. We hijack the Pthreads mutex structure (pthread_mutex_t), and use it to store the identifier of the next thread to acquire the lock: a thread can enter a critical section only if the value currently stored in the mutex structure equals the thread identifier. The initial value of the mutex structure is set during the mutex initialization routine (pthread_mutex_init) and then updated with the identifier of the next thread to acquire the lock (retrieved from the aforementioned thread’s array) upon exiting a critical section.

```c
int pthread_mutex_lock(pthread_mutex_t *mutex){
  // The mutex structure holds the identifier of the next thread to acquire the lock.
  volatile char *nextThreadID = (char*) mutex;
  char thisThreadID = get_threadID();
  // Wait until the previous thread releases the lock.
  while (*nextThreadID != thisThreadID);
  return 0;
}

int pthread_mutex_unlock(pthread_mutex_t *mutex){
  char *nextThreadID = (char*) mutex;
  // Allow the next thread to acquire the lock by updating the mutex structure.
  *nextThreadID = get_nextThreadID();
  return 0;
}
```

Figure 2: Replay lock and unlock operations.

3.3.1 Optimizations

The replayer’s lock routine requires identifying a running thread (the get_threadID macro). Library routines such as gettimeofday or pthread_self, or the use of the thread-local storage, would provide the thread identifier, albeit at the price of a function call. This overhead can be avoided by carefully allocating the stack of each thread. Placing stacks in a consecutive memory region aligned to a constituent size (\(nT hreads\times nThreads\)) boundary allows to obtain a unique number in \([0, nT hreads - 1]\) for each thread by just manipulating the stack pointer (only 3 register-to-register x86 instructions), and we use this number as the thread identifier.

Inspired by Linux’s current macro [12], we place the pointer to the current element of a thread’s array at the bottom of the thread stack. This allows a fast implementation of the get_nextThreadID macro: after the address of this pointer is calculated by manipulating the stack pointer, the identifier of the next thread to acquire the mutex is accessed by simply incrementing and dereferencing this pointer.

Hence, with the exception of a few assembly instructions that manipulate the stack pointer and make memory accesses only to thread-private data and are of high spatial locality, and aside from the store during the unlock operation, a replayed run includes only the fundamental operation necessary to enforce a schedule: waiting during a lock acquire for the immediately preceding thread to leave the critical section. We verify that this implementation of the replayer indeed incurs negligible overhead in Section 5.2.1.

3.4 Perturber

One benefit of determinism is that once the program is deployed in the field, it continues to behave as tested, i.e., “deployed determinism has the potential to make testing more valuable, as execution in the field will better resemble in-house testing” [15]. This however, makes efficient deterministic execution especially challenging, since programs are run under various conditions (e.g., OS-caused perturbations such as context switches and page faults, dynamic CPU frequency scaling) and on various systems (e.g., symmetric vs. asymmetric architectures). The perturber imitates various execution environments programs face in the field.

3.4.1 Small Perturbations

The perturber simulates small execution perturbations such as context switches, thread migrations and page faults via Linux signals and wait periods. To simulate a perturbation in a thread, a signal is sent to this target thread which mimics the magnitude of the perturbation by just waiting a certain time period. We determined the smallest possible perturbation delay in the target thread execution using a micro-benchmark as about a 5µs. This allows us to imitate the first-order effects of context switches and thread migrations (which are typically on the order microseconds [3]) and page faults (milliseconds [29]). Hence, we randomly insert two types of delays (10µs and 1ms delays) so that they occur with the following frequency: one 10µs delay for each 1ms of execution time and one 1ms delay for each 100ms of execution time. The perturber implements two scenarios: balanced perturbations (same delays in each thread) and unbalanced perturbations (the frequency of delays varies over threads, ranging from the above frequency to 1/8 of it).

3.4.2 Background Processes

To account for the presence of other processes, the perturber spawns a number of background threads which repeatedly execute a work-sleep pattern in which they first spin in an empty loop (keeping the processor busy) and then sleep (allowing for other threads to run). By varying the duration of the work vs. sleep phases we control the impact that the background threads have on the core utilization and other running threads. Again the perturber implements two scenarios: a low-intensity balanced one in which there is one background thread running on each core and each thread spends just 5% of its cycles working, and an intense unbalanced one where there is only one background thread but it spends 50% of its cycles working.
3.4.3 Dynamic Voltage and Frequency Scaling (DVFS)

The perturber uses Linux’s cpufreq system to explore the performance of deterministic execution under different DVFS policies. First, by utilizing Linux’s ondemand governor, the perturber implements a scenario in which power-performance balance is required, and the OS dynamically adjusts the CPU frequency. Second, by using the userspace governor, the perturber sets the frequency of some processor cores to the lowest value (1.2 GHz) while periodically switching the frequencies of the other cores between 1.2 and 2.2 GHz. This simulates the per-core DVFS approach [24].

3.4.4 Non-Uniform Memory Access

To explore the effects that a non-uniform memory access (NUMA) architecture has on deterministic execution, the perturber spreads the threads across the two nodes of our NUMA machine in a round robin manner. Otherwise, the threads are contained in a single NUMA node, experiencing uniform memory accesses.

3.4.5 Asymmetric Architectures

We employ the aforementioned userspace governor to create asymmetry. The perturber implements a scenario where half of the threads are running on the cores that are more powerful than the others (2.2 GHz vs. 1.2 GHz) and a scenario where only one thread is running on a powerful core.

4. EXPERIMENTAL METHODOLOGY

4.1 Benchmarks

We use the SPLASH-2 [10] and PARSEC [8] benchmarks released with the PARSEC 3.0 suite. From this suite of 27 benchmarks, we exclude two benchmarks from all of our experiments: freecine, which does not have a Pthreads version (and our framework currently only works with Pthreads), and canneal, which employs a lock-free synchronization strategy, and conversion to a version that uses Pthreads synchronization primitives would result in code bearing little similarity to the original canneal code.

We tested all the other benchmarks for data races using ThreadSanitizer [39], and found races in 16 out of 25 benchmarks. Our examination of the code revealed that races were used when it is perceived that proper synchronization is unnecessary or that it would decrease performance. Additionally, some of the races were undoubtedly program bugs: even with the assumption of sequentially consistent memory model in the presence of data races (which none of the modern programming languages, compilers, and architectures provide) these data races lead to undesired program behaviour.

We fixed all the reported data races by adding synchronization and compared the performance of the original benchmarks to the data-race-free ones. An increase in the execution time was noticed only in three cases: barnes (11%), radiosity (5%), and raytrace_parsec (8%). This confirms that there is little to be gained by introducing “benign” race accesses, as previously hypothesized by Boehm [11]. For the rest of this study, we use the data-race-free version of benchmarks as a baseline.

Unless stated otherwise, all the benchmarks were compiled with GCC 4.6 using default compilation settings of PARSEC 3.0 with added the -march=native flag to obtain binaries optimized for our machine. The input used is simlarge: decent parallel speedup was obtained for this input while the running time of a single experiment was short enough in order to run all the experiments in a reasonable amount of time.

4.2 Platform

The machine we use has two Xeon E5-2660 chips. A single chip consists of 8 cores and operates at a frequency between 1.2 GHz and 3.0 GHz. Each core supports two-way hyper-threading which yields a total of 32 hardware threads. There is 16 GB of main memory per chip, and cores have faster access to 16 GB of their chip and slower access to 16 GB of the other chip, i.e., this is a NUMA system. The operating system is Ubuntu 12.04, kernel version 3.2.

When quantifying sources of nondeterminism, we run the benchmarks to utilize all 32 hardware threads and use the “default” OS settings: ASLR is enabled, the OS controls thread scheduling, and the processor frequency varies based on the workload. On the other hand, when evaluating the cost of making synchronization deterministic, we execute both nondeterministic and deterministic runs in a more controlled environment. This way, any difference we observe across runs can be confidently attributed to differences between the schedules and not to other factors.

For the controlled environment, we disable hyper-threading and ASLR, fix the frequency to 2.2 GHz, and explicitly pin application threads to cores (one thread per core). We then configure the benchmarks to run on 8 cores since this allows us to evaluate the effect of a NUMA system on deterministic execution (Section 3.4.4). Two exceptions are dedup and ferret that for an 8-core configuration spawn much more than 8 threads and our one-thread-per-core approach is not possible. Thus, we run dedup and ferret in a 4- and 3-core configuration, respectively. The number of threads in the x264 benchmark is proportional to the number of input frames and much larger than 8; hence, we exclude this benchmark when evaluating the cost of determinism.

4.3 Metrics

4.3.1 Execution Variability

To quantify the extent to which executions are not identical in the strict sense of internal determinism, we utilize the hardware performance counters to detect changes in the executed instruction streams. We record the total number of repeatable hardware events for each thread over 10 program runs. These events are repeatable in the sense that their number is always the same for the runs with identical instruction traces. Hence, changes in the recorded number of events undoubtedly indicate the presence of nondeterminism.

The events we monitor are store instructions, conditional branches, and not taken branches. These were the only repeatable events correctly accounted for by the performance counters on our platform. We define the variability of a thread as the range (across multiple executions) of the number of events in that thread, normalized by the average number of events in all the threads. For example, in a program that on average triggers 50 events, a thread’s variability of 10% indicates that the difference in the number of events occurred in that thread between multiple runs was as much as 5 events. The execution variability is simply the sum of per-thread variabilities.
where \( \text{variability} \) and \( e_{t,r} \) denote variability of the thread \( t \) and the number of events that occurred in that thread during program run \( r \), respectively.

It is possible for executions with different instruction traces to have the same number of events, and hence, our approach can fail to detect nondeterminism. The fine-grain nature of events we monitor makes this unlikely, but we do recognize that our results are a lower bound on variability. While we could, for example, collect a complete per-thread execution trace, this would significantly perturb execution.

### 4.3.2 Deterministic Slowdown

We quantify the performance impact of imposing a certain schedule by comparing the execution time of a replay run enforcing the schedule with that of a nondeterministic run. We repeat both the replay and nondeterministic run at least 10 times and as many times as needed to get the standard error of the average execution time of each run below 0.5%. We then report the ratio of the average execution time of the deterministic replay run to the nondeterministic run. We refer to this value as the deterministic slowdown. Using bootstrap [13], we established that each reported deterministic slowdown is accurate to ±1.5%, with 95% confidence.

For fairness, we use the same controlled environment for both the deterministic and nondeterministic runs, as described in Section 4.2. Further, since the replay uses spin-based synchronization, we use scalable spinning synchronization [49] in the nondeterministic run as well. This improves the nondeterministic execution time across all the benchmarks (by as much as 2x for streamcluster). Finally, when quantifying the impact of a schedule under varying system conditions, we ensure that the same conditions are applied during both the replay and nondeterministic run.

## 5. EXPERIMENTAL EVALUATION

### 5.1 Quantifying Nondeterminism

Table 2 shows the average number of the events triggered in each benchmark across multiple runs, and the resulting execution variability (in the “all” column). Additionally, the benchmarks for which changes in the output occurred at least once have the corresponding variability cell grayed. This clearly demonstrates that the nondeterminism in multi-threaded programs execution is real, significant, and that it sometimes even leads to a different program output.

We further break the variability down to the sources discussed in Section 4.2. The “sync.”, “lazy.” and “C++” columns show the variability due to thread communication stemming from respectively: (i) program synchronization, (ii) lazy binding, and (iii) initialization of C++ local static variables. The “vec.” column displays the variability due to the ASLR and the compiler auto-vectorization. Lastly, the “lib.” column presents the variability occurred during library calls, caused by a combination of (i) thread communication that occurs during the execution of library calls, (ii) the ASLR, and (iii) nondeterministic timing functions.

### Table 2: Variability in SPLASH-2 and PARSEC benchmarks.

The variability for each column was obtained by examining 10 runs in which only the target source of nondeterminism was contributing to execution variability. This was done by either removing other sources of nondeterminism: i.e., by enforcing a fixed synchronization order using the record-replay framework, by disabling lazy binding and auto-vectorization, and by transforming local static variables into global ones, or by isolating the nondeterminism inside library calls (e.g., printfs, scanf, file I/O, and memory allocation) by pausing the performance counters during their execution.

It is important to note that the presented variability breakdown is not exact: the total variability is not equal to the sum of individual variabilities. This is because the individual variabilities do interact with each other and they are also collected on different sets of runs. Nonetheless, the breakdown reflects the extent to which the executions experienced variability due to each particular source of nondeterminism.

We make the following observations:

- Not surprisingly, program synchronization is the most significant contributor to nondeterminism, and it sometimes affects program output.

- Other sources of nondeterminism result in much less variability but do materialize. This has important implications for existing deterministic systems that rely on determinism of logical clocks. These systems typically handle nondeterminism during library calls, but assume none of the additional nondeterminism we discovered in this study. Consequently, the combination of auto-vectorization and the ASLR can compromise guarantees of strongly deterministic systems. For weakly deterministic systems, lazy binding and static initialization, in addition to vectorization, can undermine their deterministic guarantees.
5.2 The Cost of Determinism

5.2.1 Replayer’s Efficiency

Ideally, the replayer should impact execution only by forcing a thread to wait before executing a synchronization operation until the preceding scheduled synchronizations have finished. Yet, our replayer has three sources of overhead. First, although the replayer’s macros (get_threadID and get_nextThreadID) are highly optimized, they are of non-zero latency. Second, having each thread traverse its (potentially large) array that represents synchronization operations could possibly disturb the memory subsystem and slow down other memory accesses. Finally, a write to a lock during replayer’s unlock operation can potentially cause unnecessary communication between processors executing the threads not directly involved in the transfer of the lock. The last issue can be avoided by having a per-thread dedicated variable in each lock, as done with array-based queue locks [25]. However, modifying the replayer in this way brought no performance improvement. Thus, we conclude that this issue does not have a visible performance impact on our benchmarks.

The impact of the first two overhead sources should ideally be evaluated by comparing our replayer to an ideal replayer with no overhead, but this is not possible. Thus, we compare a nondeterministic run to a nondeterministic run in which the synchronization operations have been modified to execute the get_threadID and get_nextThreadID macros, but not to enforce any particular schedule. Since no schedule is enforced, the array elements that are being accessed during the modified nondeterministic run may not correspond to the synchronization operations being executed. Nonetheless, the underlying effect is still present: executing synchronization operations includes the additional latency and causes each thread to sequentially traverse a dedicated memory region.

The result of this comparison is shown in Figure 3, which displays the execution time of the nondeterministic run including replayer’s overhead normalized w.r.t. the execution time of the unmodified nondeterministic run (the error bars show 95% confidence interval). As seen, the execution times are nearly identical. Thus, we consider the execution time obtained by our framework during the replay run to provide a lower bound on the execution time that can be achieved with any deterministic execution enforcing the same schedule.

![Figure 3: The efficiency of the replayer.](image-url)

5.2.2 Deterministic Slowdown

The “Deterministic Slowdown” column of Table 3 shows the deterministic slowdown for each schedule. We make the following observations:

- The performance of a significant number of benchmarks (10 out of 24) is unaffected by the schedule being enforced and is the same in both the nondeterministic and deterministic case.
- Enforcing the serial schedule comes with a significant cost of up to 7.71x slowdown. Moreover, this schedule degrades the performance of certain benchmarks to that extent that their deterministic execution is slower than the single-threaded execution, i.e., the deterministic slowdown is higher than the parallel speedup in these cases.
- The round-robin schedule on average incurs modest slowdown of 1.66x, although it can also result in a significant performance degradation of more than 5x.
- The dynamic schedules induce a small slowdown, both on average and in the worst case. The dynamic-A schedule, which uses the number of all instructions to implement logical clocks, performs exceptionally well: the slowdown never raises above 1.32x, and it is only 1.04x on average.
- The hybrid schedule, similar to the dynamic schedules, generates only a small average slowdown of 1.18x. However, it can result in up to 2.69x slowdown.

This demonstrates that, at least for our set of benchmarks and our platform, and implementation overhead set aside, the fundamental cost of weak determinism is small.

5.2.3 Analysis

The slowdown during a deterministic run can occur due to the change in the time threads spend waiting while performing synchronization (wait time), and/or the time threads spend working (work time). The increase in the wait time is caused by a schedule forcing threads to wait longer than in the nondeterministic case, while the change in the work time may occur due to a schedule forcing a different control path, leading to an execution with different amount of work, or to an execution with the same amount of work but changed efficiency (e.g., improved/deteriorated data locality).

Figure 4 display the work time under schedules normalized w.r.t. the work time of the nondeterministic run, showing that these are very close, with the exception of dedup. Profiling reveals that the amount of work is identical in both the deterministic and nondeterministic run of dedup, but the execution efficiency changes. The compress stage of the dedup pipeline, where threads allocate and access a large number of memory buffers, experiences significantly less page faults under the nondeterministic and the round-robin runs compared to the other schedules. A schedule in this case determines the order in which threads allocate memory and this evidently influences Linux’s page allocation policy. Aside from this peculiarity, the work time does not increase under deterministic runs. Thus, we focus on the increase in the wait time.

Intuition suggests that the lock frequency should relate to the wait time during deterministic runs, since more/less
synchronization gives more/less opportunity for a schedule to force threads to wait. However, Table 3 shows this is not always the case: synchronization-intense *barnes* experiences only a small slowdown of 1.11x and only under the serial schedule, while the synchronization-light *ferret* suffers a performance degradation under all schedules. Thus, we analyze two application characteristics that can better relate to deterministic performance: the lock span and clock imbalance.

**Lock span.** The lock span is obtained during nondeterministic execution as the time between earliest and latest lock acquires, relative to the total run-time (Figure 3). More precisely, the lock span is calculated by measuring the time elapsed from the earliest to latest acquire of each lock, finding the maximum over all the locks and normalizing this time by the total execution time. We hypothesize that a low lock span translates to a low deterministic slowdown. In the worst case, the earliest and the latest acquire of a lock in the nondeterministic run get reordered during a deterministic run. In this case, the increase in the wait time will be roughly equal to the time difference between these two uses of the lock in the nondeterministic run. Hence, applications with small time differences between any two lock acquires should experience little to no increase in wait time.

Two extensions are required in order to properly account for benchmarks that suffer load imbalance or use barriers. First, for the imbalanced benchmarks, a potential increase in the wait time of a “shorter” thread translates to increase in the program execution time only when this thread is forced to wait for long enough to become the “longest” thread. Thus, when calculating the locks span, we decrease the time elapsed between the earliest and latest use of a lock by

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Deterministic Slowdown</th>
<th>Parallel Speedup</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>serial round-robin dynamic-S dynamic-A hybrid</td>
<td>lock freq.</td>
<td>lock span</td>
</tr>
<tr>
<td>barnes</td>
<td>1.11 0.97 0.95 0.95 0.99</td>
<td>6.87</td>
<td>2.7510² 0.14</td>
</tr>
<tr>
<td>cholesky</td>
<td>3.41 2.40 1.08 1.05 1.09</td>
<td>3.98</td>
<td>5.7010³ 0.90</td>
</tr>
<tr>
<td>ft</td>
<td>4.36 1.02 1.02 1.02 1.02</td>
<td>7.21</td>
<td>1.16 0.50</td>
</tr>
<tr>
<td>fmm</td>
<td>6.34 1.33 1.16 1.13 1.19</td>
<td>6.68</td>
<td>2.6710² 0.77</td>
</tr>
<tr>
<td>lu_cb</td>
<td>1.00 1.00 1.00 1.00 1.00</td>
<td>7.32</td>
<td>1.4410³ 0.00</td>
</tr>
<tr>
<td>lu_ncb</td>
<td>1.01 1.00 1.01 1.01 1.01</td>
<td>4.04</td>
<td>5.7410⁴ 0.00</td>
</tr>
<tr>
<td>ocean_cp</td>
<td>1.00 1.00 1.00 1.00 1.00</td>
<td>7.41</td>
<td>4.5910² 0.00</td>
</tr>
<tr>
<td>ocean_ncp</td>
<td>1.00 1.00 1.00 1.00 1.00</td>
<td>7.83</td>
<td>2.6710² 0.00</td>
</tr>
<tr>
<td>radiosity</td>
<td>7.58 3.03 1.09 1.07 2.69</td>
<td>7.00</td>
<td>8.4510² 1.00</td>
</tr>
<tr>
<td>radix</td>
<td>1.00 1.00 1.00 1.00 1.00</td>
<td>7.70</td>
<td>2.6310³ 0.00</td>
</tr>
<tr>
<td>raytrace</td>
<td>7.71 2.93 1.08 1.03 1.88</td>
<td>7.71</td>
<td>3.6710¹ 1.00</td>
</tr>
<tr>
<td>volrend</td>
<td>6.12 1.91 1.08 1.02 1.67</td>
<td>6.37</td>
<td>5.9710¹ 0.75</td>
</tr>
<tr>
<td>water_resquared</td>
<td>1.00 1.00 1.00 1.00 1.00</td>
<td>6.85</td>
<td>3.14 0.00</td>
</tr>
<tr>
<td>water.spatial</td>
<td>1.00 1.00 1.00 1.00 1.00</td>
<td>6.44</td>
<td>8.2010⁻³ 0.00</td>
</tr>
</tbody>
</table>

Table 3: Deterministic slowdowns, parallel speedup of the nondeterministic run w.r.t. the single-threaded run, the number of lock acquires per million cycles and the program characteristics that impact the performance under schedules.

Figure 4: The total time threads spent working under schedules, normalized w.r.t. the work time in the nondeterministic run.
the maximum observed difference between the run times of threads. Second, when barriers are used to divide program into phases, the use of a lock in different program phases can never be reordered by a schedule, so it can not contribute to the execution time increase. Hence, we calculate the lock span as the sum of per-phase lock spans.

The lock span values are shown in Table 3. The 10 benchmarks that have lock span lower than 0.001 show no slowdown under deterministic runs. The barnes benchmark, which has a low lock span of 0.14, experiences only 1.11x slowdown in the worst case. A low lock span translates to good performance under any schedule. Looking at the remaining benchmarks, it is evident that a high lock span translates to significant slowdown under the serial schedule but not necessarily under the other schedules.

The inability of the lock span to fully characterize the performance under the round-robin, dynamic and hybrid schedules stems from the fact that the earliest and latest acquire of a lock do not always get reordered, as assumed in the definition of the lock span. We define the round-aware lock span in the same way as the lock span with the difference that it only considers two acquire of a lock as a potential source of slowdown if these acquire indeed can be reordered under a schedule. When calculating the round-aware lock span, each lock acquire is given an index that denotes its order among other acquire done by the same thread; the time between two acquire is considered only if the “earlier” lock has a larger index than the “later” lock.

As Table 3 shows, a low round-aware lock span translates to good performance under the round-robin schedule, while a high round-aware lock span suggests a significant performance degradation under the same schedule. One exception is facesim: it uses condition variables to effectively create a barrier, leading to incorrect detection of program phases used during the lock span calculation. After manually replac-
notations to improve Parrot’s performance (as done in [13]), for the round-robin schedule using our framework. We run

running the benchmarks with Parrot to the results obtained

weakly deterministic system that employs the round-robin

results with the performance under Parrot [13]. Parrot is

overheads increase the cost of determinism, we compare our

5.4 Comparison with a Deterministic System

To investigate the extent to which the implementation

overheads increase the cost of determinism, we compare our

results under CoreDet [4]. Hypothesizing that other deter-
nistic slowdowns of a few PARSEC and PBBS [10] bench-

marks under CoreDet [4]. Hypothesizing that other deter-

ministic systems would perform similarly, they argued that

cost of enforcing only the necessary scheduling constraints.

includes the overhead of passing and waiting for the turn).

metric machine was previously done by Shelepov et al. [37].

of DVFS in order to simulate asymmetry on otherwise sym-

implementations, which is what we study. The use

tations. While they do evaluate the overhead of the additional

impact of requiring the same deterministic execution acros s

gle implementation, they do not evaluate the performance

in the execution time of multi-threaded workloads, while our study focuses on the variability in the execution trace and program output.

The Cost of Determinism. Liu et al. [23] and Olszewski et al. [35] measure execution times under the round-robin and dynamic schedules, respectively, when the additional features (e.g., memory protection) or overheads (e.g., interrupts used to maintain logical clocks) of their deterministic systems are disabled or isolated. Their analysis blends the cost of both enforcing and generating the schedules (e.g., includes the overhead of passing and waiting for the turn). In contrast, we use a record-replay approach to measure the cost of enforcing only the necessary scheduling constraints.

Nguyen et al. [32] recently reported significant deterministic slowdowns of a few PARSEC and PBBS [10] benchmarks under CoreDet [4]. Hypothesizing that other deterministic systems would perform similarly, they argued that there is a substantial cost to determinism for applications that perform orders of magnitude more synchronization than the PARSEC benchmarks. We evaluate the fundamental cost of determinism on a large set of benchmarks, including ones having several orders of magnitude more synchronization than the PARSEC benchmarks studied in [32] (e.g., fluidanimate and radiosity). Our results show that, under the right schedule and when implementation overhead is set aside, there is little fundamental cost to determinism.

Determinism in the Field. Alameldeen and Wood [1] demonstrate that by analyzing a certain technique in a narrow environment, one can reach wrong conclusions, especially for multi-threaded workloads. How et al. [22] design an architecture for deterministic execution and show how to guarantee determinism across its different implementations. While they evaluate the overhead of the additional support for determinism-across-implementations on a single implementation, they do not evaluate the performance impact of requiring the same deterministic execution across different implementations, which is what we study. The use of DVFS in order to simulate asymmetry on otherwise symmetric machine was previously done by Shelepov et al. [37].

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>bal.</td>
<td>ubal.</td>
<td>bal.</td>
</tr>
<tr>
<td>barnes</td>
<td>0.95</td>
<td>0.96</td>
<td>0.97</td>
</tr>
<tr>
<td>cholesky</td>
<td>1.05</td>
<td>1.05</td>
<td>1.06</td>
</tr>
<tr>
<td>fft</td>
<td>1.01</td>
<td>1.02</td>
<td>1.07</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>1.13</td>
<td>1.13</td>
<td>1.19</td>
</tr>
<tr>
<td>lu_cb</td>
<td>1.00</td>
<td>1.00</td>
<td>0.99</td>
</tr>
<tr>
<td>lu_rcb</td>
<td>1.01</td>
<td>1.01</td>
<td>1.01</td>
</tr>
<tr>
<td>ocean_cp</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>ocean_rncp</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>radiosity</td>
<td>1.07</td>
<td>1.07</td>
<td>0.99</td>
</tr>
<tr>
<td>radix</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>raytrace</td>
<td>1.03</td>
<td>1.03</td>
<td>0.98</td>
</tr>
<tr>
<td>voilrend</td>
<td>1.03</td>
<td>1.03</td>
<td>0.89</td>
</tr>
<tr>
<td>water_resquared</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>water_spatial</td>
<td>1.00</td>
<td>1.00</td>
<td>1.01</td>
</tr>
</tbody>
</table>

Table 4: The slowdowns under various execution set-

gings.

6. RELATED WORK

Quantifying Sources of Nondeterminism. Various

sources of nondeterminism have been previously identified

(unordered thread communication [19, 30], address space layout [30, 38], system dependences [35, 34]). However, our study is the first to measure the extent to which they make executions internally or externally nondeterministic and to identify nondeterminism due to lazy binding, initialization of C++ static variables and compiler auto-vectorization in

the presence of ASLR. Devietti et al. [13] measure how fre-
defly, across multiple runs, a particular dynamic instance of a load reads data created from a different dynamic instance of a store. This quantifies variability in thread communication (which may or may not induce program nondeterminism) and not the extent to which unordered thread communication introduces nondeterminism. Alameldeen and Wood [1] and Mytkowicz et al. [30] quantify the variability in the execution time of multi-threaded workloads, while our study focuses on the variability in the execution trace and program output.

Figure 6: Comparison with Parrot.
Record-Replay. Record-replay systems (see [35] for a good summary) are generally designed for debugging and are typically concerned with low overhead recording. We employ a record-replay technique in order to decouple the overhead of generating a schedule from the fundamental performance impact of enforcing it, and we focus on a low overhead replay.

7. CONCLUSION

In this paper, we use a schedule-record-replay system to divorce implementation overhead from the fundamental cost of enforcing a fixed synchronization order in data-race-free parallel programs. We find that this fundamental cost is significantly lower than previously thought; for the dynamic schedule, it is no more than 32% slowdown in parallel execution and 4% on average. However, in the presence of skewed execution conditions, this cost can slow down execution by a factor of almost 2x. We propose and assess two new program characteristics, lock span and clock imbalance, that indicate how suited a given schedule is for a given program. Finally, we show that deterministic guarantees of systems that rely on the determinism of logical clocks can be compromised because of other sources of nondeterminism that make these clocks nondeterministic.

Our study focuses on data race free programs, i.e., on weak determinism. However, we believe that our results are of interest to both weakly and strongly deterministic systems. While handling data races increases the complexity and performance penalty of strongly deterministic systems, researchers have been successful in reducing the overhead of deterministically resolving data races to the point where the only program serialization is caused by synchronization operations. Evaluating the tightness of the lower bound empirically determined here for data-race-free programs in the context of strong determinism is an interesting direction for future work.

The findings of this study open several opportunities for work on determinism. First, they suggest that there remains room for lowering implementation overheads in existing proposals for deterministic systems. Second, they suggest that insisting on a fixed schedule for programs in production is probably too costly. Thus, it may be necessary to relax the definition of determinism, e.g., by allowing an execution to follow any of the multiple permitted schedules for a single program input. Finally, our findings give insights into program characteristics that impact the cost of enforcing determinism, which may guide the design of future deterministic systems.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their many helpful comments. We also thank the members of our research group for their feedback on the manuscript.

9. REFERENCES