What is the Cost of Determinism?

Cedomir Segulja Tarek S. Abdelrahman
University of Toronto
{seguljac, tsa}@eecg.toronto.edu

Abstract

We analyze the fundamental performance impact of enforcing a fixed thread communication order to achieve deterministic execution. Our analysis consists of three parts, performed on a real system using the SPLASH-2 and PARSEC benchmarks. First, we quantify the effect that various sources of non-determinism have on program execution. We find that thread communication is the prevalent source of non-determinism and that it sometimes affects program output. Second, we divorce the implementation overhead of a system imposing a specific communication order from the impact of enforcing this order. We show that this fundamental cost of determinism is small (slowdown of 4% on average and 33% in the worst case). Finally, we analyze this cost under perturbed execution conditions. We find that demanding determinism when threads face such conditions can cause almost 2x slowdown.

1. Introduction

Recognizing non-determinism as one of the main culprits for struggles with parallel programming, there has recently been a stream of systems that suppress non-determinism by enforcing a fixed thread communication order, or *schedule*. Evaluation of these proposals reflects that there is a significant cost to determinism (Table 1).

System	Max. Overhead	Schedule
Grace [8]	3.6x	serial
Dthreads [26]	4x	round-robin
Conversion [32]	5x	round-robin
Parrot [16]	3.8x	round-robin
Kendo [34]	1.6x	dynamic
RCDC [19]	1.7x	dynamic
RFDet [27]	2.6x	dynamic
DMP [17]	1.7x	hybrid
CoreDet [5]	10x	hybrid
Calvin [23]	1.7x	hybrid

Table 1: State-of-the-art deterministic systems. The overhead numbers, reported in [7] and in papers introducing these systems, show that the authors themselves found that enforcing determinism comes with a significant cost.

These systems come in both hardware [17,19,23] and software [5, 8, 16, 26, 27, 32, 34] flavors, and enforce determinism using a variety of techniques, including speculation [8, 17], buffering [17, 19, 23,26,27] and version control [32]. 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. Implementation overhead set aside, *what is the fundamental cost of determinism*?

In this paper, we seek to answer this question. By running the SPLASH-2 [37] and PARSEC [9] benchmark suites on a dual chip 16-core Xeon E5-2660 machine, we empirically study the cost of making thread communication deterministic under a given schedule. Our study has three parts. In the first part, we quantify the degree to which non-determinism that occurs in a common environment influences program execution. We identify the effect of previously unmentioned sources of non-determinism, like lazy binding, C++ local static variables initialized with a non-compile-time constant and compiler auto-vectorization. In line with popular belief, our quantitative analysis shows that thread communication is the most significant contributor to non-determinism of parallel programs, and it sometimes affects program output.

In the second part of our study, we evaluate the performance impact of enforcing a fixed schedule on program execution. We use a schedule-record-replay framework to divorce the implementation overhead of generating a specific schedule form the fundamental requirements needed to enforce the schedule. This way we find an upper bound on performance that can be achieved with any deterministic execution enforcing that same schedule. The analysis reveals that the cost of determinism is smaller than previously thought: only 4% on average and never greater than 33% increase in the execution time compared to a non-deterministic run.

In the last part of our study, using the same framework we measure the cost of determinism under various execution conditions (e.g., the presence of context switches, background processes and execution on asymmetric architectures). While some deterministic systems have been designed with determinism-across-systems in mind [23], we are unaware of any work that evaluates the performance of deterministic execution across systems. We find that insisting on determinism when threads face uneven conditions like unbalanced system loads or an asymmetric architecture can slow down non-deterministic execution by as much as 2x.

To the best of our knowledge, this is the first study that on a real machine: (i) quantifies the impact of sources of non-determinism on execution variability, (ii) finds the fundamental performance cost of enforcing determinism, and (iii) evaluates the cost of determinism for varying systems.

2. What do you mean by determinism?

The notion of determinism seems to be a simple one: most would agree determinism is a program property that requires observing the same output whenever the program runs with the same input. Still, this (non-)definition relies on the intuitive understanding of the terms *program output* and *program input*. Depending on what one assumes these terms to really mean, many flavours of determinism arise[1]. Here, we highlight common definitions of determinism and state the formulation we adopt in this paper.

Program output. One often considers only the end program result to be the output (*external determinism* [21]). However, while debugging, the intermediate values or even instruction operands might be considered as the program output. In this case determinism is 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* [21]). Besides these two extremes, other formulations are possible; an excellent discussion on plausible definitions of the program output and on the corresponding forms of determinism is given by Lu

[1]Or, as Hans Boehm [1] puts it, "determinism is in the eye of the beholder".

and Scott [28]. 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. This, we believe, strikes the right balance between the internal and external determinism, since it requires that programs calculate the same result in the same way, allowing for benign differences in the instruction stream (which as we show in Section 3 are common on modern systems).

Program input. A natural definition of program input includes supplied program arguments and file/network input. However, many software and hardware parameters may also influence the program output (e.g., process address space layout, the version of libraries, the size of hardware structures, etc.). Consequently, practical approaches to determinism either consider these parameters to be program input as well (by silently assuming they do not vary across executions), they force them to be constant (e.g., by providing a custom OS implementation – *system-level determinism* [3, 6]), or they actually do allow for variations of certain system parameters (e.g., providing determinism across microarchitectural variations – *unbounded determinism* [23]).

We believe that the program input should encompass only the *real* program input and exclude system parameters, since guaranteeing determinism only for a narrow setting is of a little practical use. Hence, we consider system parameters a source of non-determinism rather than program input.

Program class. The formulation of determinism also varies based on the class of programs being considered. Although determinism can be provided for arbitrary parallel programs (*strong determinism* [34]), to make its implementation more efficient, determinism can also be guaranteed only for data-race-free programs (*weak determinism* [34]), or programs that do have data races but do not employ ad-hoc synchronization (e.g., as done with Dthreads [26]).

While we do believe that systems that provide strong determinism are valuable during debugging, we also believe that in production, when the cost of determinism matters the most, it is valid to expect data-race-freedom for two reasons. First, there is a growing consensus that *all* data races are bugs [2, 11, 14, 29, 30], and new programming language standards go as far as providing no semantics to programs with data races [13]. Second, it has been hypothesized that there is little performance to be gained by introducing "fast" racy accesses [12]. Our experience with the SPLASH-2 and PARSEC benchmarks confirms this: although these benchmarks initially contained a number of data races, we removed them noticing a performance degradation only in three cases: `barnes`(11%), `radiosity`(5%) and `parsec.raytrace`(8%). Hence, in this study we focus on data-race-free programs.

3. How much non-determinism is out there?

Here we describe the sources of non-determinism that we experienced with the SPLASH-2 and PARSEC benchmarks and quantify the degree to which they influence program execution. We start by recognizing all sources that can cause executions of the same program, with the same input, not to be identical in the strict sense of internal determinism. Then, we quantify the extent to which each source introduces variability in program execution, in order to understand which sources of non-determinism need to be controlled to achieve *SyncOrder* determinism.

Thread communication. Due to small variations in the execution environment (the state of hardware caches, TLBs, presence of other applications, etc.), accesses to shared resources from different threads can occur in different orders across multiple program runs. This can potentially cause variability in program execution and possibly affect the output.

It is well known that accesses that can occur in different orders are the ones to shared program variables and the ones done inside library/system calls [5, 34]. In addition, we experienced that unordered thread accesses leading to non-determinism also appear through subtle compiler/run-time interactions. For example, the run-time symbol resolution (also known as lazy binding) [20], aside from speeding up program load time, 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 a different thread across program runs, so can the number of instructions executed by each thread, making runs 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 program runs non-deterministic.

Unless noted otherwise, in the rest of this paper we use the term "thread communication" to refer specifically to the communication due to accessing shared program variables.

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 the code that inspects the values of memory 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 [10]. As we witnessed, these transformations result in a 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 subsequently execute the widest data type transfers supported by the hardware. Finally, the application code itself can be dependent on memory addresses.

System and library dependences. Certain library and system calls are non-deterministic by construction (e.g., any timing function like `gettimeofday`). For others however, one would expect to get a deterministic result but this is not necessarily the case. For example, Linux's `read` system call is allowed to return smaller number of bytes than the number of bytes requested [17]. Hence, different runs of the same program that must read a certain number of bytes may make different number of `read` calls. Further, the aforementioned run-time symbol resolution, besides being dependant on the order in which threads invoke library functions, can also cause variability between two runs if the exported symbols of the libraries used in these two runs are not identical (e.g., due to system update).

Quantifying non-determinism. To quantify the degree to which the sources of non-determinism influence programs, we use hardware performance counters. Over 10 program runs, we record the total number of *repeatable* hardware-related events for each thread. These hardware events are repeatable in the sense that their number is always the same for the executions with identical instruction traces. Hence, any change in the number of events signals the variability in the execution. The converse however, does not hold, and this approach can fail to diagnose variability. While we could collect a complete per-thread execution trace, this would significantly perturb program execution and we are interested in the variability that occurs during typical program runs.

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 (Weaver et al. [36] report on the occasional incorrectness of the performance counters). We define the *variability of a thread* as the range of the number of events occurred in that thread, normalized by the average number of events occurred in all the threads. For example, in a program that on average triggers 50 events, 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 then defined as the sum of per-thread variabilities:

$$variability = \sum_{t \in threads} variability_t = \sum_{t \in threads} \frac{\max\limits_{r \in runs} e_{t,r} - \min\limits_{r \in runs} e_{t,r}}{\sum\limits_{\substack{r \in runs \\ t \in threads}} e_{t,r} \;/\; \#runs}$$

where $variability_t$ and $e_{t,r}$ denote variability of the thread t and the number of events in that thread during run r, respectively.

Results. We run the benchmarks on a 16-core dual chip Xeon E5-2660 machine. No effort was made to perturb the executions to induce variability: all 10 runs were executed on a quiet machine and under the same conditions (no changes to dynamic libraries, environmental variables, OS settings, etc., were done between the runs). Table 2 shows the average number of the events occurred 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 non-determinism in multi-threaded programs execution is real, significant, and that it sometimes leads to a different program output.

	Benchmark	Avgerage Number of Events	Execution Variability (%)					
			all	thread communication			vec.	lib.
				app.	lazy.	C++		
splash	barnes	10129529155	3.43	3.11	$3.85 \cdot 10^{-5}$	0	0	$1.01 \cdot 10^{-6}$
	cholesky	164106307	2.95	5.93	$3.56 \cdot 10^{-4}$	0	$1.98 \cdot 10^{-4}$	$5.41 \cdot 10^{-4}$
	fft	3450383778	$2.31 \cdot 10^{-4}$	$2.27 \cdot 10^{-5}$	$2.44 \cdot 10^{-5}$	0	0	$3.03 \cdot 10^{-5}$
	fmm	4828582248	9.49	8.20	$3.54 \cdot 10^{-5}$	0	$2.07 \cdot 10^{-8}$	$1.05 \cdot 10^{-5}$
	lu_cb	4860852648	1.95	4.54	$2.94 \cdot 10^{-5}$	0	$4.94 \cdot 10^{-7}$	$1.47 \cdot 10^{-5}$
	lu_ncb	4831248732	2.36	1.31	$1.21 \cdot 10^{-5}$	0	$4.76 \cdot 10^{-7}$	$2.01 \cdot 10^{-5}$
	ocean_cp	1669826904	2.28	2.61	$1.41 \cdot 10^{-4}$	0	$3.17 \cdot 10^{-4}$	$2.46 \cdot 10^{-6}$
	ocean_ncp	1546323893	0.06	0.09	$2.66 \cdot 10^{-4}$	0	$1.12 \cdot 10^{-4}$	$3.17 \cdot 10^{-6}$
	radiosity	23053230129	11.28	15.75	$2.44 \cdot 10^{-6}$	0	0	$1.60 \cdot 10^{-7}$
	radix	2615152384	$5.97 \cdot 10^{-3}$	0.01	$6.67 \cdot 10^{-5}$	0	0	$4.29 \cdot 10^{-5}$
	raytrace	7783841897	4.37	3.44	$7.50 \cdot 10^{-6}$	0	0	$9.54 \cdot 10^{-5}$
	volrend	2888381273	32.53	40.25	0	0	0	$2.35 \cdot 10^{-6}$
	water_nsquared	22025925182	10.29	11.18	$1.06 \cdot 10^{-5}$	0	0	$1.18 \cdot 10^{-7}$
	water_spatial	5677017700	8.71	12.36	$4.79 \cdot 10^{-5}$	0	0	$1.86 \cdot 10^{-5}$
parsec	blackscholes	922533017	$3.53 \cdot 10^{-4}$	0	$2.19 \cdot 10^{-4}$	0	0	$1.73 \cdot 10^{-6}$
	bodytrack	2903052665	8.46	3.61	$1.94 \cdot 10^{-5}$	0	$3.34 \cdot 10^{-6}$	$1.72 \cdot 10^{-3}$
	dedup	11578237115	22.90	9.56	$1.69 \cdot 10^{-5}$	0	0	$6.60 \cdot 10^{-3}$
	facesim	6067774400	0.04	$7.84 \cdot 10^{-4}$	$1.68 \cdot 10^{-5}$	0	$3.32 \cdot 10^{-5}$	0.05
	ferret	7062608901	31.93	33.81	$7.96 \cdot 10^{-6}$	0	$1.42 \cdot 10^{-8}$	$4.54 \cdot 10^{-3}$
	fluidanimate	3851899211	0.06	0	$7.58 \cdot 10^{-5}$	0	0	0.07
	raytrace	4784353664	3.45	4.00	$4.70 \cdot 10^{-5}$	$5.46 \cdot 10^{-5}$	0	$4.01 \cdot 10^{-6}$
	streamcluster	1063626549	$2.98 \cdot 10^{-4}$	0	$8.24 \cdot 10^{-5}$	0	0	$1.50 \cdot 10^{-6}$
	swaptions	6412816309	$9.23 \cdot 10^{-5}$	0	$8.82 \cdot 10^{-5}$	0	0	$1.02 \cdot 10^{-5}$
	vips	6772016494	3.41	3.80	$4.43 \cdot 10^{-8}$	0	0	$8.04 \cdot 10^{-3}$

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

We further breakdown the variability to the sources discussed earlier. The "app.", "lazy." and "C++" columns show the variability only due to thread communication stemming from respectively: (i) accesses to shared program variables, (ii) the lazy binding, and (iii) the 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) non-deterministic timing functions. communication, address variation and timing functions. The variability for each of the mentioned columns was obtained by examining 10 program runs in which all but the target source of non-determinism were removed (by enforcing a fixed order of synchronization operations, by disabling run-time symbol resolution and auto-vectorization, and by transforming local static variables into global ones) or isolated (by pausing the performance counters during the execution of non-deterministic library calls). Hence, the presented variability breakdown is not exact; the total variability is not equal to the sum of individual variabilities, since these are collected on different sets of runs. Nonetheless, it reflects the extent to which the executions experienced variability due to each particular source of non-determinism.

Based on this data, we make the following two observations. First, thread communication while accessing shared variables is, not surprisingly, the most significant contributor to non-determinism of parallel programs, and it sometimes affects program output. Second, other sources of non-determinism, while being of a lesser extent, do materialize under typical execution conditions. This has important implications for existing deterministic systems that rely on determinism of logical clocks (Section 4). These systems typically do handle non-determinism during library calls, but assume no additional non-determinism we discovered in this study. 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.

Interactions of thread communication and the other sources of non-determinism. Our experience with these benchmarks is that the order in which threads access shared variables (i.e., execute synchronization operations) after any given execution point is only a function of the program input and the order that happened up to that execution point. That is, the other sources of non-determinism do not affect the synchronization order. The exceptions are the `fluidanimate` and `dedup` benchmarks where the addresses of variables, used as inputs to a hash function, influence the synchronization order. This is a bad practice since it limits portability, and for the `fluidanimate` it actually caused a bug.

In other words, fixed thread communication alone, in addition to providing repeatable result, seems to ensure that executions contain the same synchronization operations, executed by the same threads and in the same order – thus it provides the form of determinism we subscribe to.

4. How to achieve determinism?

Formally, thread communication is described 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 relationship between synchronization operations, i.e., the classical Lamport's happened-before relation [25]. In the absence of data races, the schedule fully captures the communication between threads [33]. Hence, repeating the same schedule guarantees deterministic thread communication.

We proceed by describing schedules used by the state-of-the-art systems (shown earlier in Table 1). Our description is conceptual in a way that describes one possible execution that conforms to a given schedule. Note, however, that a single schedule allows for multiple program executions, as long as the synchronization order stays the same. Clearly, we are interested in the best performing execution that satisfies a schedule – we describe how to obtain such execution in Section 5.

Schedules. The central concept used to describe schedules is the concept of a *turn*. At any given time during the execution, it is only one thread's turn, and only while a thread has the turn can it execute synchronization operations. Ensuring that the turn is passed in a deterministic manner leads to a deterministic thread communication. Exactly when is the turn passed, determines which of the following schedule types will be enforced.

In the *serial schedule* [8] a thread holds on to its turn until the end of its execution. Effectively, this schedule forces the parallel execution to emulate the thread communication of an execution in which threads are executed synchronously, in program order (i.e., as function calls). In the *round-robin schedule* [26] a thread passes its turn after each executed synchronization operation. Thus, threads execute synchronization operations in a round-robin fashion.

In the *dynamic schedule* [34] the turn is passed on each tick of a per-thread deterministic *logical clock*. The original proposal by Olszewski et al. [34] uses store instructions to maintain logical clocks, i.e., a thread passes its turn upon executing one memory store. Note that a logical clock does not necessarily follow physical time precisely, but the closer it does so, the closer will the deterministic execution resemble a non-deterministic one.

The *hybrid schedule* [17] can be seen as combination of the serial and dynamic schedule. The execution of each thread is broken down into *quanta* and the serial schedule is enforced during each quantum – a thread passes its turn only when it reaches the end of its single quantum. The duration of a quantum is determined using logical clocks: 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" [5].

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 $N = \infty$), the round-robin schedule (when $N = \infty$ 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.

5. How much does it cost?

To answer this question, we use a schedule-record-replay framework (Figure 1). When evaluating the performance of a program under a specific schedule, we execute it twice. During the first run, we run the program with a scheduler and record the order of synchronization operations. In the second run, we replay the recorded schedule, guaranteeing the same deterministic execution as in the first run, but without the scheduler's overhead. The crucial requirement of the replay run is that it should force threads to wait only when absolutely necessary for the schedule to be respected and that it should do so with little or no overhead. If we can achieve that, then the run time of the second run is a close-to-best time that can be attained with any system enforcing the same schedule.

During replay, the schedule is represented with multiple arrays, one per each thread. Each element of a thread's array corresponds to one critical section[2] that will be executed by this thread and indicates the *next* thread to enter a critical section protected by the same lock. A thread traverses its array during the execution, consuming one element of the array for each executed critical section. Hence, laying out the elements of a thread's array in the order in which the thread will enter critical sections (and this order is known in advance from the recording run) enables a fast fetch of the array elements and guarantees good cache behavior (a single byte array element representation supports programs with up to 256 threads).

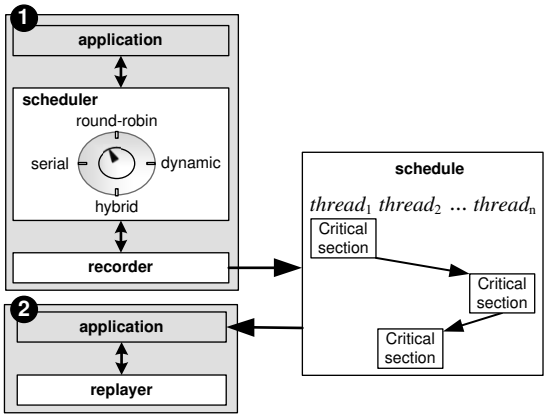

Figure 1: The schedule-record-replay framework.

We use the replay lock and unlock operations, shown in Figure 2, that replace corresponding Pthread routines and are inspired by ticket locks [31]. We hijack the `pthread_mutex_t` structure, and use it to store the identifier of the thread currently holding 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 `pthread_mutex_init` routine and then updated with the identifier of the next thread to acquire the mutex upon exiting a critical section.

```
int pthread_mutex_lock(pthread_mutex_t *mutex){
  volatile char *currentThreadID = (char*) mutex;
  // wait until the previous thread releases the mutex.
  while (*currentThreadID != get_threadID());
  return 0;
}
int pthread_mutex_unlock(pthread_mutex_t *mutex){
  char *currentThreadID = (char*) mutex;
  // allow the next thread to acquire the mutex.
  *currentThreadID = get_nextArrayElement();
  return 0;
}
```

Figure 2: Replay lock and unlock operations.

The overhead associated with the `get_nextArrayElement` and `get_threadID` macros is reduced down to a few assembly instructions (e.g., 3 register-to-register x86 instructions for the `get_threadID` macro) by carefully aligning the stack of each thread, similar to the implementation of Linux's `current` macro [15]. By comparing the run times of the non-deterministic execution of our benchmarks to the run times of the non-deterministic execution in which we modified the synchronization operations to include the replayer's macros, we found the overhead to be under 2% for all the benchmarks. Hence, we consider the execution time obtained by our framework during the replay run of a pre-recorded schedule to provide close to an upper bound on performance that can be achieved with any deterministic execution enforcing the same schedule.

Results. The "Deterministic slowdown" column of Table 3 shows the run times of different deterministic executions normalized w.r.t. the non-deterministic execution on a 16-core dual-chip Xeon E5-2660 machine, i.e., it shows by how much enforcing a particular schedule slows down the execution. The benchmarks were run with at least 8 threads (certain PARSEC benchmarks spawn more threads than specified when starting the program [9]). For the dynamic schedule, we use the number of executed stores instructions to maintain logical clocks, as done by Kendo [34]. For the hybrid schedule, we set the phase duration to 100,000 instructions

[2] We focus our description on critical sections – our framework supports the other commonly used synchronization primitives (barriers and condition variables) as well.

and do not use the reduced serial phase since these setting have been found previously to give the best performance [26].

	Benchmark	Parallel Speedup	Deterministic slowdown					Lock freq.
			serial	rr	dynamic	hybrid	dynamic+	
splash	barnes	6.86	1.10	0.98	0.95	0.99	0.96	$2.49 \cdot 10^{2}$
	cholesky	3.99	3.39	2.39	1.07	1.10	1.05	$1.67 \cdot 10^{1}$
	fft	7.27	4.36	1.02	1.01	1.02	1.01	$2.66 \cdot 10^{-1}$
	fmm	6.66	6.34	1.33	1.16	1.19	1.13	$4.21 \cdot 10^{1}$
	lu_cb	7.32	1.00	1.00	1.00	1.00	1.00	$1.44 \cdot 10^{-3}$
	lu_ncb	4.24	1.00	0.99	1.01	1.00	1.00	$4.52 \cdot 10^{-4}$
	ocean_cp	4.57	1.00	1.00	1.00	1.00	1.00	$4.50 \cdot 10^{-2}$
	ocean_ncp	7.79	1.00	1.00	1.00	1.00	1.00	$2.65 \cdot 10^{-2}$
	radiosity	6.99	7.58	3.04	1.09	2.67	1.08	$1.30 \cdot 10^{2}$
	radix	7.71	1.00	1.00	1.00	1.00	1.00	$2.63 \cdot 10^{-3}$
	raytrace	7.71	7.72	2.93	1.08	1.88	1.02	4.76
	volrend	6.37	6.12	1.91	1.08	1.67	1.02	9.59
	water_nsquared	6.81	1.00	1.00	1.00	1.00	1.00	3.14
	water_spatial	6.31	1.00	1.00	1.00	1.00	1.00	$8.19 \cdot 10^{-3}$
parsec	blackscholes	7.98	1.00	1.00	1.00	1.00	1.00	0.00
	bodytrack	5.83	5.87	1.04	1.05	1.05	1.05	$4.30 \cdot 10^{-1}$
	dedup	3.95	5.04	1.77	1.63	1.34	1.33	3.11
	facesim	6.17	6.19	1.00	1.00	1.00	1.00	1.22
	ferret	2.92	6.19	3.19	1.58	1.25	1.23	$9.70 \cdot 10^{-3}$
	fluidanimate	5.99	1.81	0.99	0.99	0.97	0.97	$6.09 \cdot 10^{2}$
	raytrace	7.18	7.26	1.52	1.06	1.01	1.01	2.19
	streamcluster	6.11	1.00	1.00	1.00	1.00	1.00	$9.19 \cdot 10^{-3}$
	swaptions	7.92	1.00	1.00	1.00	1.00	1.00	0.00
	vips	7.61	7.61	5.27	1.31	1.05	1.06	$6.87 \cdot 10^{-1}$
average slowdown			**3.61**	**1.60**	**1.09**	**1.17**	**1.04**	
maximum slowdown			**7.72**	**5.27**	**1.63**	**2.67**	**1.33**	

Table 3: The speedup of the non-deterministic execution w.r.t. the single-threaded execution, slowdowns occurred during deterministic executions, and the frequency of lock operations (per million cycles) during the non-deterministic execution.

We make the following observations. First, 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 non-deterministic and deterministic case. Looking at the lock frequency, it can be seen that these benchmarks use no or infrequent synchronization. However, the two of the most synchronization-intense benchmarks (`barnes` and `fluidanimate`) experience only a moderate slowdown. Although intuition suggests that lock frequency should be an indicator of deterministic performance (since less synchronization operations gives less opportunities to a schedule to force threads to wait) this is not always the case.

Second, enforcing the serial schedule comes with a significant cost of up to 7.72x slowdown. Moreover, this schedule degrades the performance of certain benchmarks to that extent that their deterministic parallel execution is slower than the single-threaded execution, i.e., the deterministic slowdown is higher than the speedup in these cases. Third, the round-robin schedule on average incurs modest slowdown of 1.60x, although it can as well result with a significant performance degradation of more than 5x. Fourth, the dynamic and hybrid schedules induce only a small average slowdown of 9% and 17%, respectively. However, while the hybrid schedule can result with up to 2.67x slowdown, the executions under the dynamic schedule never experience a slowdown greater than 63%.

We also experimented with a version of the dynamic schedule where the logical clocks are incremented on each instruction, instead of on each store instruction (the "dynamic+" column in Table 3). Although the performance counter for the number of instructions is not stable on our platform (and in general [36]), the replayed executions we use to obtain performance results are deterministic due to our schedule-record-replay approach. With this change we observe even better performance of the deterministic execution – the deterministic slowdown never raises above 33% - and we hypothesize this is caused by the total number of instructions better tracking threads progress than just the number of stores.

This result demonstrates that, at least for our set of benchmarks and our target environment, and with implementation overhead set aside, there is fundamentally a very small cost to determinism.

6. How about determinism in the field?

One of the promised benefits of determinism is that, once the multi-threaded software gets deployed in the field, the software 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*" [18]. This however, makes efficient deterministic execution especially challenging, since the programs are run on a variety of systems. Moreover, even when executed on the same machine, programs are exposed to different conditions due to, for example, OS interactions (interrupts and context switches) or temperature changes (processors with Intel Turbo Boost adjust the clock frequency based on temperature, among other parameters). Nonetheless, we are unaware of any work that analyzes the performance of deterministic execution in more then one setting. To that end, we evaluate deterministic execution of the best performing schedule (dynamic+) using the same framework but in the presence of system perturbations and across systems.

Determinism under perturbations (small perturbations, background processes and DVFS). We simulate small execution perturbations via Linux signals and wait periods. Specifically, to mimic a perturbation in a thread, a signal is first sent to this target thread which upon receiving the signal simulates the magnitude of the perturbation by just waiting a certain time period. On a micro-benchmark in which we test smallest possible perturbation, the observed delay in the target thread execution was around 5μs, which allows us to imitate the first-order effects of perturbations of similar and larger magnitudes (e.g., context switch, thread migrations and page faults) [4, 22]. Hence, we randomly insert two types of delays (10μs and 1ms delays) so that on average there is one 10μs delay for each 1ms of execution time and one 1ms delay for each 100ms of execution time. We explore two scenarios: balanced perturbations (delays inserted in each thread with the previously mentioned frequency) and unbalanced perturbations (the frequency of delays varies over threads, ranging from the aforementioned frequency to the $1/8$ of that frequency).

To account for the presence of other processes, we spawn 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 phase we control the impact that the background thread has on the core utilization and the other running threads. Again we explore two scenarios: a low-intensity balanced one in which there is a one background thread running on each core and each thread spends just 5% of its cycles working, and a intense unbalanced one where there is only one background thread but it spends 50% of it cycles working.

We also analyze the impact of the dynamic voltage and frequency scaling (DVFS). First, by utilizing the Linux's `ondemand` governor, we try the scenario where the power-performance balance is required, and OS dynamically adjusts the CPU frequency. Second, by using the `userspace` governor, we set the frequency of some cores to the lowest value (1.2 GHz), while periodically switching the frequencies of the other cores between 1.2 GHz and 2.2 GHz. This scenario simulates the per-core DVFS approach [24].

Determinism across systems (NUMA and asymmetric architectures). To explore the effects that a non-uniform memory access (NUMA) architecture has on deterministic execution, we spread the threads across the two chips of our NUMA machine in a round robin manner. For experiments with the asymmetric architecture,

we utilize the same machine but this time we employ the earlier described DVFS to create asymmetry (a similar approach to creating asymmetry was done by Shelepov et al. [35]). We explore 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.

Results. Table 4 shows the execution time of the deterministic run with the dynamic+ schedule for each of the described execution settings normalized w.r.t. the non-deterministic run under the same setting, i.e., it shows by how much enforcing the dynamic+ schedule slows down the execution for each setting. Additionally, we have grayed the cells for which there is more than 10% change in the slowdown *compared to the slowdown* from Table 3.

	Benchmark	Small perturb.		Bkgd. processes		DVFS		NUMA	Asym. arch.	
		bal.	ubal.	bal.	ubal.	auto	manual		8/8	1/15
splash	barnes	0.95	0.96	0.96	0.97	0.92	0.96	0.91	0.94	0.96
	cholesky	1.05	1.05	1.06	1.25	1.06	1.02	1.08	1.03	1.09
	fft	1.01	1.02	1.07	1.02	1.01	1.01	1.01	1.00	1.01
	fmm	1.13	1.13	1.19	1.24	1.13	1.13	1.14	1.15	0.97
	lu_cb	1.00	1.00	0.99	1.03	1.00	1.00	1.00	1.00	0.98
	lu_ncb	1.01	1.01	1.01	1.03	0.97	1.01	1.03	0.99	0.98
	ocean_cp	1.00	1.00	1.00	1.01	1.00	1.00	1.00	1.00	1.01
	ocean_ncp	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	radiosity	1.07	1.08	1.19	1.94	1.13	1.07	1.11	1.46	1.71
	radix	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	raytrace	1.03	1.03	1.14	1.92	1.08	1.02	1.03	1.44	1.69
	volrend	1.03	1.03	1.08	1.19	1.06	1.02	1.03	1.38	1.55
	water_nsquared	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.02	0.97
	water_spatial	1.00	1.01	1.00	1.08	1.00	1.00	1.00	1.00	1.03
parsec	blackscholes	1.00	1.00	1.01	1.00	1.00	1.00	1.00	1.00	1.00
	bodytrack	1.04	1.06	1.05	1.51	1.04	1.05	1.03	1.33	1.56
	dedup	1.33	1.33	1.35	1.31	1.29	1.33	1.32	1.64	1.31
	facesim	1.00	1.01	1.00	1.00	0.99	1.00	1.00	1.00	1.00
	ferret	1.24	1.25	1.19	1.29	1.21	1.25	1.15	1.37	1.10
	fluidanimate	0.97	0.97	0.97	0.98	0.97	0.97	0.97	0.98	1.01
	raytrace	1.00	1.01	1.07	1.77	1.05	1.01	1.02	1.39	1.63
	streamcluster	1.00	1.00	1.01	1.00	1.00	1.00	1.00	1.00	1.00
	swaptions	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	vips	1.06	1.07	1.09	1.09	1.15	1.06	1.06	1.43	1.53
average slowdown		**1.04**	**1.04**	**1.06**	**1.19**	**1.04**	**1.04**	**1.04**	**1.15**	**1.17**
maximum slowdown		**1.33**	**1.33**	**1.35**	**1.94**	**1.29**	**1.33**	**1.32**	**1.64**	**1.71**

Table 4: The slowdowns under various execution settings.

The data shows that the cost of determinism increases when threads face significantly askew conditions (unbalanced system load and asymmetric architectures). In the worst case, deterministic execution causes a slowdown of 94%, or almost 2x. This suggests that guaranteeing a fixed communication order across significantly different settings is probably too slow for production use. Looking at the application characteristics, we found that the increase typically happens for programs that implement dynamic load balancing, where the non-deterministic execution is able to adjust the per-thread work to the changing execution conditions, while the deterministic execution is not.

7. Conclusion

Our quantitative analysis shows that thread communication is the most prevalent source of non-determinism; however, previously unmentioned sources, like lazy binding, C++ static initialization and compiler auto-vectorization coupled with ASLR, must be taken into account when designing deterministic systems.

By using a schedule-record-replay framework, we are able to measure the cost of deterministic thread communication in the absence of implementation overhead. We show that, while most schedules inherently cause significant performance degradation, the dynamic schedule has the potential to deliver determinism with only 4% slowdown. This makes us hopeful that a low overhead deterministic system will be a reality one day.

However, our experiments also show that achieving efficient determinism in the field (and not only on a single machine), will be harder and that it will probably require a more relaxed definition of determinism, e.g., by allowing reordering of commutative shared memory accesses, or by allowing an execution to follow any of the multiple permitted schedules for a single program input.

8. References

[1] Proc. of HotPar. *USENIX ;login:*, 36(5):111, 2011.

[2] S. Adve. Data Races are Evil with No Exceptions: Technical Perspective. *Commun. ACM*, 53(11):84–84, 2010.

[3] A. Aviram et al. Efficient System-Enforced Deterministic Parallelism. In *Proc. of OSDI*, pages 1–16, 2010.

[4] A. Bastoni et al. Cache-Related Preemption and Migration Delays: Empirical Approx. and Impact on Schedulability. In *Proc. of OSPERT*, pages 33–44, 2010.

[5] T. Bergan et al. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *Proc. of ASPLOS*, pages 53–64. ACM, 2010.

[6] T. Bergan et al. Deterministic Process Groups in dOS. In *Proc. of OSDI*, pages 1–16, 2010.

[7] T. Bergan et al. The Deterministic Execution Hammer: How Well Does it Actually Pound Nails? In *Proc. of WODET*, 2011.

[8] E. D. Berger et al. Grace: Safe Multithreaded Programming for C/C++. In *Proc. of OOPSLA*, pages 81–96, 2009.

[9] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[10] A. J. C. Bik et al. Automatic Intra-register Vectorization for the Intel Architecture. *Int. J. Parallel Program.*, 30(2):65–98, 2002.

[11] H.-J. Boehm. How to Miscompile Programs with "Benign" Data Races. In *Proc. of HotPar*, pages 3–3, 2011.

[12] H.-J. Boehm. Position Paper: Nondeterminism is Unavoidable, but Data Races are Pure Evil. In *Proc. of RACES*, pages 9–14. ACM, 2012.

[13] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *Proc. of PLDI*, pages 68–78, 2008.

[14] H.-J. Boehm and S. V. Adve. You Don't Know Jack About Shared Variables or Memory Models. *Commun. ACM*, 55(2):48–54, Feb. 2012.

[15] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. O'Reilly & Associates Inc, 2005.

[16] H. Cui et al. Parrot: A Practical Runtime for Deterministic, Stable, and Reliable Threads. In *Proc. of SOSP*, pages 388–405, 2013.

[17] J. Devietti et al. DMP: Deterministic Shared Memory Multiprocessing. In *Proc. of ASPLOS*, pages 85–96, 2009.

[18] J. Devietti et al. DMP: Deterministic Shared-Memory Multiprocessing. *IEEE Micro*, 30(1):40–49, 2010.

[19] J. Devietti et al. RCDC: A Relaxed Consistency Deterministic Computer. In *Proc. of ASPLOS*, pages 67–78, 2011.

[20] U. Drepper. How to Write Shared Libraries. `http://www.akkadia.org/drepper/dsohowto.pdf`, 2011.

[21] P. A. Emrath and D. A. Padua. Automatic Detection of Nondeterminacy in Parallel Programs. In *Proc. of PADD*, pages 89–99, 1988.

[22] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 5th edition, 2011.

[23] D. Hower et al. Calvin: Deterministic or Not? Free Will to Choose. In *Proc. of HPCA*, pages 333 –334, 2011.

[24] W. Kim et al. System Level Analysis of Fast, Per-Core DVFS Using On-Chip Switching Regulators. In *Proc. of HPCA*, pages 123–134, 2008.

[25] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558–565, 1978.

[26] T. Liu et al. DTHREADS: Efficient Deterministic Multithreading. In *Proc. of SOSP*, pages 327–336. ACM, 2011.

[27] K. Lu et al. Efficient Deterministic Multithreading Without Global Barriers. In *Proc. of PPOPP*, 2014.

[28] L. Lu and M. L. Scott. Toward a Formal Semantic Framework for Deterministic Parallel Programming. In *Proc. of DISC*, pages 460–474, 2011.

[29] B. Lucia et al. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *Proc. of ISCA*, pages 210–221, 2010.

[30] D. Marino et al. DRFX: A Simple and Efficient Memory Model for Concurrent Programming Languages. In *Proc. of PLDI*, pages 351–362, 2010.

[31] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, Feb. 1991.

[32] T. Merrifield and J. Eriksson. Conversion: Multi-version Concurrency Control for Main Memory Segments. In *Proc. of EuroSys*, pages 127–139, 2013.

[33] A. Nistor et al. Light64: Lightweight Hardware Support for Data Race Detection During Systematic Testing of Parallel Programs. In *Proc. of MICRO*, pages 541–552, 2009.

[34] M. Olszewski et al. Kendo: Efficient Deterministic Multithreading in Software. In *Proc. of ASPLOS*, pages 97–108, 2009.

[35] D. Shelepov et al. HASS: A Scheduler for Heterogeneous Multicore Systems. *SIGOPS Oper. Syst. Rev.*, 43(2):66–75, 2009.

[36] V. Weaver et al. Non-Determinism and Overcount on Modern Hardware Performance Counter Implementations. *Proc. of ISPASS*, pages 215–224, 2013.

[37] S. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of ISCA*, pages 24–36, 1995.