# The Deterministic Execution Hammer: How Well Does it Actually Pound Nails?

Tom Bergan Joseph Devietti Nicholas Hunt Luis Ceze
University of Washington
{tbergan,devietti,nhunt,luisceze}@cs.washington.edu

## Abstract

This paper takes a critical look at the benefits provided by state-of-the-art deterministic execution techniques. Specifically, we look at four applications of deterministic execution: debugging, fault-tolerant replication, testing, and security. For each application, we discuss what an ideal system would provide, and then look at how deterministic systems compare to the ideal. Further, we discuss alternative approaches, not involving determinism, and we judge whether or not these alternatives are more suitable. Along the way, we identify open questions and suggest future work.

Ultimately, we find that there are competitive alternatives to determinism for debugging and replicating multithreaded programs; that determinism has high, though unproven, potential to improve testing; and that determinism has distinct security benefits in eliminating some covert timing channels. Furthermore, determinism is a unified solution for all four applications: this confers a distinct advantage over point solutions that do not compose well with one another.

## 1. Introduction

There is a consensus in the community that nondeterminism makes the development of parallel and concurrent software substantially more difficult. Recent work has proposed eliminating nondeterminism through *deterministic execution* [3, 5–7, 15, 16, 28]. Proponents of deterministic execution argue that it simplifies many challenging tasks faced by software developers. This paper examines how much deterministic execution actually simplifies those challenging tasks.

We examine four specific tasks: debugging (Section 3), fault-tolerant replication (Section 4), testing (Section 5), and removal of covert channels for security (Section 6). For each task, we ask: How much does determinism simplify the task? What are alternative approaches, not involving determinism, and how does determinism compare to those alternatives? Are the benefits of determinism worth the costs? Our goals are to identify weak spots in the current state of the art, and to direct future research towards what we consider the most important open problems. Key among these problems is understanding the performance limits of determinism.

In the rest of this paper, we first provide context and summarize the current state of deterministic execution (Section 2). We then examine the benefits and costs of deterministic execution, as applied to four applications (Sections 3–6), discuss the performance limits of determinism (Section 7), and finally conclude (Section 8).

| System | Notes | Overheads |
|---|---|---|
| Kendo [28] | runtime library | 1x–1.6x |
| DMP [15] | custom hardware | 1x–1.7x |
| CoreDet [5] | compiler and runtime | 1.1x–10x |
| dOS [6] | operating system | 1.2x–10x |
| Determinator [3] | operating system | 1x–10x |
| Calvin [18] | custom hardware | 1x–1.7x |
| RCDC [16] | custom hardware + runtime | 1x–1.7x |
| Grace [7] | runtime library | 1.2x–3.6x |

**Table 1.** Systems for deterministic execution

## 2. Deterministic Execution: Overview

A program is deterministic if its execution is solely a function of its explicit inputs. Programs written in specialized languages (*e.g.,* NESL [8], StreamIt [37], or DPJ [9]) are deterministic by construction and enjoy all the benefits of determinism, with zero runtime overhead. Unfortunately, deterministic languages like these have had limited uptake. The majority of today's parallel programs are developed using general-purpose, nondeterministic languages. Therefore, although we consider deterministic languages to be good ideas, we do not discuss them further.

Rather, we are interested in systems which take programs written in nondeterministic languages (*e.g.,* C++ or Java) and execute them in a deterministic way. Such systems eliminate the *internal* nondeterminism [6] that can arise due to nondeterministic timing variations in hardware and OS schedulers. These systems ensure that, given a fixed input, a program will appear to execute the same sequence of operations every time it runs, even when run on a multiprocessor.

Prior research has proposed three broad strategies for deterministic execution that we summarize below. Table 1 summarizes the state-of-the-art systems, along with their reported runtime overheads.

In the **wait-for-turn** approach, first implemented by Kendo [28], a thread is allowed to complete a synchronization operation only if all other threads have completed more total instructions (hence "wait-for-turn"). This provides a deterministic order of synchronization, which guarantees determinism for race-free programs. However, it is *weakly deterministic* [28] only: it does not provide any guarantees for programs with data races. *Strongly deterministic* systems, on the other hand, guarantee determinism even in the presence of data races.

**Lockstep quanta** are a common approach for providing strong determinism. The idea is to divide execution into

bulk-synchronous quanta. During an individual quantum, threads are isolated, and memory updates are applied deterministically at the end of the quantum. Four quantum formation strategies have been proposed: DMP-O, which uses data-ownership tracking; DMP-TM, which uses transactional memory with ordered commit; and DMP-TSO[1] and DMP-HB, which both use non-speculative store buffering with ordered commit. These strategies have been implemented by a few systems: DMP-O by [5, 6, 15], DMP-TM by [15], DMP-TSO by [3, 5, 16, 18], and DMP-HB by [16]. All of these systems provide determinism for arbitrary programs, even in the presence of data races.

Determinator [3] provides deterministic execution for arbitrary programs using a DMP-TSO-inspired buffering strategy, and also provides primitives for check-in/check-out memory updates in fork-join programs. By using these primitives directly, programmers can write deterministic fork-join programs in general-purpose languages.

A few systems are tailored to **specific classes of programs**. For example, Grace [7] implements deterministic execution for C/C++ fork-join parallel programs, using transactions with ordered commit to deterministically resolve data races, and the Revisions programming model [10] supports determinism in a task-parallel system. Others have used deterministic locking protocols to execute a stream of database transactions deterministically [38].

### 2.1 Overheads of deterministic execution

Kendo is implemented purely in software and has relatively low overheads, although its scalability has not been analyzed. Further, it provides weak determinism only, making it less applicable to arbitrary programs.

Current software-only approaches to strong determinism either do not support arbitrary programs (Grace) or do not have overheads low enough for use in production environments (CoreDet, dOS, Determinator). Hardware-accelerated determinism has low overheads but requires architectural changes (DMP, Calvin, RCDC). Further, even with hardware support, current techniques for strong determinism may not perform well in highly dynamic environments, such as in datacenters, where multiple applications are packed onto a single machine, and further, they may not scale well to systems with a large number of cores. Section 7 includes a more detailed discussion of these issues.

The most obvious open problem for deterministic execution, then, is to provide strong determinism with robust and scalable performance. Specifically:

> **Open Problem 1.** Can we provide strongly deterministic execution with performance that is robust enough, scalable enough, and fast enough for use in production environments?

[1] DMP-TSO was named DMP-B in [5]. We use the more precise name DMP-TSO as suggested by [16].

While some uses of deterministic execution are justified even with its current performance profile, higher performance clearly makes all of its uses more compelling.

## 3. Debugging

The first application we examine in detail is debugging. The ability to reproduce a bug is crucial for helping a programmer understand the bug's root cause. Reproducibility enables repeated examination of the bug's effects as well as advanced debugging techniques like reverse execution. To be most useful, reproducibility should have low time and space costs at runtime: a higher-overhead approach could be useful during development, but a low-overhead mechanism would allow production executions—where many bugs are discovered—to be reproduced as well. Finally, reproducibility should be available for programs with data races, as these programs are often the most buggy.

**Record-and-replay** systems make an execution reproducible by recording all sources of nondeterminism into a replay log. Deterministic execution schemes support record-and-replay with no space costs at runtime, aside from the logging of explicit program inputs, which all record-and-replay systems must do. Unfortunately, current deterministic execution schemes are not ideally suitable: weakly deterministic schemes do not support programs with data races, and strongly deterministic schemes are not yet fast enough for use in production (see Section 2).

### 3.1 Alternatives to Deterministic Execution

There is a wealth of work on pure record-and-replay techniques for reproducing multithreaded executions. Beyond recording program inputs, these techniques also log information about memory interleavings so that an execution can be precisely replayed. Hardware approaches [19, 25, 26, 40] work for arbitrary programs and have low overheads that are comparable to hardware deterministic execution, and recent proposals produce logs with reasonably small sizes [19, 25].

Software record-and-replay techniques offer a trade-off between runtime performance and replay precision. Some approaches guarantee precise replay by logging all shared memory interactions [17, 21, 22], often with high time and space overheads. Recently, however, LEAP [20] and DoublePlay [39] have demonstrated that clever recording techniques can reduce recording overheads to less than 20% slowdown for many applications. Other approaches log a subset of shared memory interactions (typically synchronization operations) to reduce time and space costs during recording, at the expense of either decreased replayability [32] or increased replay costs as in PRES [30] and ODR [1], which both perform a potentially impractical search of the execution space during replay.

### 3.2 Discussion

There is no ideal software-only solution for record-and-replay. Software-only deterministic execution schemes pro-

duce very small record logs (up to 80,000× smaller than pure record-and-replay systems [6]), but have unacceptable runtime performance. Probabilistic replay systems like PRES and ODR seem to make the right trade-off: they sacrifice some amount of replay precision to achieve high performance in production environments.

An interesting alternative, not yet explored, is to combine Kendo's weak determinism with the replay technique proposed by PRES and ODR. This would enable some other benefits of determinism in addition to debugging, without imposing the runtime costs of strong determinism.

# 4. Fault-Tolerant Replication

Replication is a common technique used by fault-tolerant systems. Standard techniques for fault-tolerant replication using the *state machine approach* [33] assume the program being replicated executes deterministically. This requirement has made it difficult to replicate multithreaded programs on conventional systems.

Deterministic execution is a natural solution for replication [6, 31, 38]. Unfortunately, current deterministic execution systems are unsuitable for production environments, being either too slow, susceptible to execution divergence on data races, or requiring hardware support (see Section 2).

## 4.1 Alternatives to Deterministic Execution

If deterministic execution is not suitable, how can we replicate multithreaded programs in an efficient, fault-tolerant way? Prior work has proposed three possible answers to this question that we survey below. Unfortunately, none of the alternatives is completely satisfying.

A first alternative is to **record-and-replicate** all sources of internal nondeterminism, including the order of synchronization and the order of shared memory accesses. This approach is also known as *online* replay. Unfortunately, systems built using this approach either assume uniprocessor execution [36] or race freedom [4], or have unacceptable performance [35]. It is difficult to envision this approach performing efficiently for programs with frequent synchronization, due to the costs of transmitting large replay logs.

A second alternative is to **execute-and-check** for divergence. The idea is to execute replicas independently and verify that they produce the same output. When divergence is detected, the replicas rollback to a checkpoint and reexecute in a slower but deterministic way, for example using deterministic serialization. This approach follows the same observation made by PRES and ODR that *output determinism* is sufficient for deterministic replay. ReSpec [23] is a Linux implementation of this approach with runtime performance that generally exceeds that of software-only deterministic execution. However, ReSpec's implementation supports replication on a single machine only, not across multiple machines.

A third alternative is **continuous checkpointing**, as implemented in Remus [14]. In this approach, the program executes on a leader machine, while the system takes rapid checkpoints and transfers those checkpoints to remote machines, which serve as hot backups. Remus has high performance for small chekpoints, but decreasing performance as checkpoint size increases. Moreover, the master/slave structure of continuous checkpointing precludes using voting protocols to survive Byzantine failures.

## 4.2 The Case for Diversity in Deterministic Execution

Even assuming deterministic execution had no runtime overhead, it is not the end-all answer to multithreaded replication. In fact, some replication schemes exploit nondeterminism to reduce the chance that all replicas will crash simultaneously due to a heisenbug. ReSpec, for example, exploits multithreaded scheduler nondeterminism. This replication property is known as *execution diversity* [12].

Can we provide similar diversity in deterministic execution? The question seems absurd, since determinism is, by design, completely lacking in diversity. However, these opposites could be profitably combined: diversity across replicas increases fault-tolerance, while determinism for individual replicas provides debugging (Section 3) and testing (Section 5) benefits. For this to happen, we need a notion of *diverse* deterministic execution:

> **Open Problem 2.** Can we provide efficient and meaningfully *diverse* deterministic execution for more effective replication?

One idea is to introduce a user-supplied *seed* that can alter a program's schedule independently of other inputs. Example seeds include quantum size, pseudorandom commit orders, or pseudorandom store buffer merging policies. The schedules produced by different seed values must be (1) equally performant and (2) "meaningfully diverse," which says, informally: given two randomly chosen seeds, it is unlikely those seeds will produce schedules that trigger the same heisenbug. Unfortunately, the performance and diversity requirements may be in conflict. For example, the performance of lockstep quanta is highly sensitive to large changes in quantum size, but it is unclear whether quantum sizes close together will produce meaningfully diverse schedules.

If execution is diverse, how do we ensure that all replicas produce the same output without diverging? One solution is to use an execute-and-check strategy as in ReSpec. Another solution is to assume the program is designed to produce deterministic output, even though it is written in a nondeterministic language. This assumption is common in prior work on diverse replication [12]. Given this assumption, any divergence in execution should be considered a bug in the program, meaning it should be dealt with like any other Byzantine fault.

## 5. Testing

Testing a multithreaded program is a daunting challenge. A tester must wade through not only a large number of possible inputs, but also an exponential number of schedules for each input, hoping to find one of the few behaviors that is buggy. Prior work on testing multithreaded programs has focused on schedule fuzzing: given a fixed input (usually provided by a developer), a custom scheduler attempts to explore as many interesting schedules as possible [11, 27, 34].

When a multithreaded program is executed deterministically, however, there is just *one* possible schedule for any given input. On the surface, this appears to reduce the problem of testing a deterministic multithreaded program to the problem of testing a sequential program, greatly simplifying the testing process for multithreaded software. This section explores the following question: What would be the ramifications on the testing process if deterministic execution was cheap and pervasive?

First, we observe that while determinism's one-schedule-per-input is a valuable simplification from nondeterminism, it implies little about the total number of schedules possible for a given program (Section 5.1). Next we discuss how the notion of an "input" for current deterministic execution schemes is not the straightforward notion of input for a sequential program (Section 5.2). Finally we discuss how determinism adds global dependences to a program that impact important modular testing practices (Section 5.3).

### 5.1 Size of the schedule space

While deterministic execution produces just one schedule per input, Figure 1 illustrates the possible effects that determinism can have on the total number of possible schedules for a program. For any given input, note that determinism allows only one schedule (a), while nondeterminism allows many (b). However, as we consider increasing numbers of inputs, it is far from clear whether the total number of schedules reaches some bound that is lower than the bound for nondeterminism. In the best case, determinism would shrink the schedule space to a small enough size that it can be exhaustively explored, substantially reducing testing effort (c). However, it may be the case that determinism either does not reduce the schedule space, or reduces it by such a small degree as to be immaterial (d). This brings us to a key open question:

> **Open Problem 3.** Can deterministic execution, in practice, reduce the total number of schedules possible for a program?

It is clear that executing a program $P$ deterministically results in *no more* possible schedules than executing $P$ nondeterministically. In the pathological case, if $P$ reads its schedule from its input, then $P$ will have exactly as many schedules as inputs, and determinism doesn't reduce the size of the space to be explored. However, real programs are unlikely to be quite so diabolical, and so determinism will likely reduce the size of the schedule space at least somewhat. Experiments run with the Tern system suggest that point (c) is plausible, at least for some programs [13]. Precisely quantifying this effect remains an open problem.

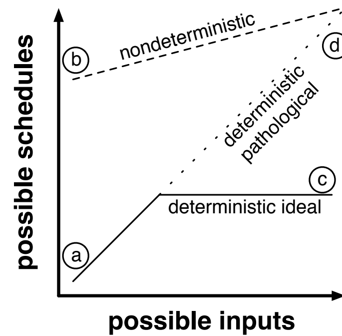


**Figure 1.** Comparing the space of possible schedules for nondeterministic execution and deterministic execution.

It is important to understand why point (c) is a meaningful ideal. If deterministic execution shrinks the schedule space to a small enough size that it can be exhaustively explored, an entire class of bugs could be eliminated through testing. Specifically, data races and atomicity violations could be eliminated through exhaustive testing of all schedules. This would be an obviously huge win for determinism.

However, when the number of schedules grows with the number of inputs (d), it becomes impossible to enumerate all schedules since in practice we cannot enumerate all inputs. In this case, testing tools must be heuristic-driven. Interestingly, the goal shifts from uncovering buggy *schedules* in the nondeterministic world, to uncovering buggy *inputs* in the deterministic world. Prior work in the nondeterministic world has exploited the "small scope hypothesis," showing that most concurrency bugs can be uncovered by a schedule with at most two preemptions [11, 27]. It is not clear how to derive schedules with few preemptions in the deterministic world, given the highly input-dependent nature of deterministic schedules. Because of this, it may actually be *easier* to find bugs by searching for schedules in the nondeterministic world, rather than by searching for inputs in the deterministic world.

### 5.2 Size of the input space

Though current deterministic execution schemes enforce one schedule per input, this belies a need to be precise about the definition of "input", which includes all parameters that affect the schedule that ultimately manifests at runtime. For example, lockstep quanta approaches produce different schedules depending on the target quantum size, all other inputs being held constant. Unfortunately, prior work has shown that tuning parameters like quantum size is typically very important for getting high performance on a given in-

| schedule affected by | DMP [15] | CoreDet [5] | RCDC [16] | Calvin [18] | Kendo [28] |
|---|---|---|---|---|---|
| data races | | | | | ✓ |
| quantum size | ✓ | ✓ | ✓ | ✓ | ✓ |
| cache size | ✓ | | ✓ | | |
| owner. gran. | ✓ | ✓ | | | |

**Table 2.** Implicit inputs of current systems (fewer is better).

put.[2] Thus, these parameters function as implicit inputs, and a tester will want to explore many of these implicit inputs to ensure that programs are reliable as well as performant.

Table 2 gives an overview of the implicit inputs of current deterministic execution strategies. Kendo [28] is weakly deterministic, so data races can lead to different schedules even when all other inputs are held constant. Early proposals for strongly deterministic execution [15] used custom hardware and were sensitive to the granularity at which memory ownership was tracked (*e.g.,* cache line size) and the size of on-chip structures. More recent proposals solved some of these problems with unbounded store buffers [5, 18] and new algorithms [5, 16]. All proposals are sensitive to quantum size parameter: the strongly deterministic schemes perform scheduling in terms of lockstep quanta, while Kendo uses a "chunk size" parameter to adjust the rate at which logical time increments. This leads to another open question:

> **Open Problem 4.** Can we provide strongly deterministic execution that is performant *without* dependences on implicit inputs?

The notion of implicit inputs generalizes beyond the particular factors identified in Table 2. We can imagine deterministic execution schemes that are insensitive to changes in program input. A good starting point is Tern [13], although Tern provides best-effort schedule memoization only, not deterministic execution. Execution could even be robust to changes in the code of the program itself – early determinism work [15] discussed how to keep execution unperturbed when small amounts of debugging code are inserted, but it is unclear how to extend this to more general code changes.

### 5.3 Modularity

Modular unit testing is vital for efficient software development. It is more efficient to test a module in isolation, rather than retesting the module in every program in which it is used. Unfortunately, all state-of-the-art deterministic execution schemes are *global*: they produce deterministic schedules for entire programs, using either the instruction counts of all threads in the wait-for-turn approach, or enforcing periodic global barriers in the lockstep quanta approach.

As a result, when testing a module in isolation, we must consider all possible contexts in which the module can be called. A context includes a group of calling threads, along with a sequence of calls they make into a module. Since the wait-for-turn and lockstep quanta approaches rely on instruction counts, a context also includes the relative instruction counts of each calling thread. Even though the module executes deterministically given a specific execution context, the vast number of contexts possible even for small modules is not promising. The small scope hypothesis, again, suggests that not all contexts need to be explored, though it is unclear whether deterministic execution provides any advantage over nondeterminism.

Ideally, deterministic execution would produce schedules for each module individually and piece those schedules together, enabling modular testing of deterministic schedules:

> **Open Problem 5.** Can we design deterministic execution strategies that enable better modular testing than nondeterministic execution does?

Modular deterministic testing techniques would also help contain the input-space explosion that plagues execution strategies with implicit input dependences. The sum of the state spaces in each small module would be far less than the product of spaces required to test a monolithic program.

## 6. Secure Covert Channels

Determinism can be an effective countermeasure against timing attacks, as was first observed in [2]. Briefly, even if access to explicit timing channels like `gettimeofday` is disallowed or coarsened, a multithreaded program can construct its own high-resolution timer by using a "timer thread" that runs the code: `for(;;) time++;`. Nondeterministic reads of the `time` variable allow a program to construct a notion of time which can be used in cache timing attacks. Determinism eliminates this timing channel by ensuring the clock increments in logical time instead of real time.

In the absence of deterministic execution, [29] outlines alternative mitigation measures such as random scheduler perturbations and avoiding co-residence. While these alternatives have low performance overhead, they do not eliminate the root cause of the timing channel as determinism does.

## 7. On the performance limits of determinism

Currently, the benefits of determinism are tempered by the runtime cost of existing deterministic execution schemes. To estimate the overheads of future deterministic schemes, we analyze the first-order costs of the two main approaches to deterministic execution, and then compare them to nondeterministic execution.

The wait-for-turn approach, exemplified by Kendo [28], reads every thread's instruction count on every synchronization operation. In other words, synchronization in wait-for-turn determinism requires global communication, even though nondeterministic synchronization rarely requires communication amongst all threads in a program, and sometimes requires no communication at all. Wait-for-turn synchronization can be implemented in $\Omega(\log N)$ time at best,

[2] To our knowledge, no one has studied this effect across different inputs.

where $N$ is the number of processors, and where a tree structure is used for communication.

The lockstep quanta approach, exemplified by DMP [15], requires a mechanism to isolate the updates from each thread. Store buffering is emerging as a popular mechanism [3, 5, 10, 16, 18] due to its avoidance of speculation. Store buffering imposes fixed overheads on an execution, since both loads and stores need to check the store buffer. These overheads can be mitigated via copy-on-write techniques [3] or future hardware support [16, 18].

More importantly, the lockstep quanta approach requires a global barrier at every quantum boundary. These global barriers impose two costs on execution. First is the $\Omega(\log N)$ cost of communication to implement a scalable barrier [24], where $N$ is again the number of processors. Second is the cost of waiting for all threads to reach the barrier. Consider running a program with $N$ threads on $N$ processors. If one thread $t$ is switched out during a quantum to make room for a high priority service, all others threads must wait for $t$ to be rescheduled before they can proceed, effectively serializing execution. This is known as quantum *imbalance*, and it makes the performance of lockstep quanta very sensitive to OS-level scheduling decisions. Quantum imbalance becomes an even bigger problem in environments with dynamic frequency scaling, heterogeneous multicore systems, and in datacenters, where applications are often tightly packed onto a single computer.

Moreover, since program synchronization induces a quantum boundary in the lockstep approach, these barriers are never less frequent than synchronization, and are frequently much more so. From this we conclude that wait-for-turn is the more scalable approach. However, the need for global communication in both approaches raises questions about the scalability of deterministic execution overall and leads to our final open problem:

**Open Problem 6.** Can we build a deterministic execution system that does not require global communication?

Kahn process networks may provide some hints to a solution, as they offer deterministic execution free of global coordination. However, encoding arbitrary programs in Kahn networks without introducing deadlocks is difficult. A future determinism strategy might combine aspects of Kahn networks with the wait-for-turn approach to obtain better scalability than either.

## 8. Conclusions

We have identified several open problems in deterministic execution that we feel present promising areas for future work. While there are often alternatives that provide an individual benefit of determinism, determinism's key strength is that it provides many benefits via a single mechanism. Deploying the best point solution in each of the four applications we've identified can add up to less than the sum of its parts. For example, record-and-replay schemes may do well to favor record speed over replay speed, but replication schemes must balance both. There may also be little reuse of machinery between point solutions, *e.g.,* a logging-based record-and-replay scheme has little in common with a continuous checkpointing replication scheme, resulting in a complicated system design and higher overheads. Future work in determinism will hopefully address the open problems we've identified and discover new ones, making determinism ever more performant and practical.

## References


[1] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *SOSP*, 2009.

[2] A. Aviram, S. Hu, B. Ford, and R. Gummadi. Determinating Timing Channels in Computer Clouds. In *CCSW*, 2010.

[3] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient System-Enforced Deterministic Parallelism. In *OSDI*, 2010.

[4] C. Basile, Z. Kalbarczyk, and R. Iyer. Active Replication of Multithreaded Applications. *IEEE Trans. Parallel Distrib. Syst.*, 17(5), 2006.

[5] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ASPLOS*, 2010.

[6] T. Bergan, N. Hunt, L. Ceze, and S. Gribble. Deterministic Process Groups in dOS. In *OSDI*, 2010.

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

[8] G. Blelloch. NESL: A Nested Data-Parallel Language (Version 3.1). Technical report, CMU.

[9] R. Bocchino, V. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.

[10] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent Programming with Revisions and Isolation Types. In *OOPSLA*, 2010.

[11] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A Randomized Scheduler with Probabilistic Guarantees of Finding Bugs. In *ASPLOS*, 2010.

[12] B.-G. Chun, P. Maniatis, and S. Shenker. Diverse Replication for Single-Machine Byzantine-Fault Tolerance. In *USENIX*, 2008.

[13] H. Cui, J. Wu, C. che Tsai, and J. Yang. Stable Deterministic Multithreading Through Schedule Memoization. In *OSDI*, 2010.

[14] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *NSDI*, 2008.

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

[16] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: A Relaxed Consistency Deterministic Computer. In *ASPLOS*, 2011.

[17] G. Dunlap, D. Lucchetti, M. Fetterman, and P. Chen. Execution Replay of Multiprocessor Virtual Machines. In *VEE*, 2008.

[18] D. Hower, P. Dudnik, D. Wood, and M. Hill. Calvin: Deterministic or Not? Free Will to Choose. In *HPCA*, 2011.

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

[20] J. Huang, P. Liu, and C. Zhang. LEAP: Lightweight Deterministic Multi-processor Replay of Concurrent Java Programs. In *FSE*, 2010.

[21] O. Laadan, N. Viennot, and J. Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *SIGMETRICS*, 2010.

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

[23] D. Lee, B. Wester, J. Flinn, S. Narayanasamy, and P. Chen. Respec: Efficient Multiprocessor Replay via Speculation and External Determinism. In *ASPLOS*, 2010.

[24] J. Mellor-Crummey and M. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM TOCS*, 9(1), 1991.

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

[26] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay. In *ASPLOS*, 2009.

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

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

[29] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. *Topics in Cryptology*, 3860/2006, 2006.

[30] S. Parka, W. Xiong, Z. Yin, R. Kaushik, K. Lee, S. Lu, and Y. Zhou. Do You Have to Reproduce the Bug at the First Replay Attempt? – PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *SOSP*, 2009.

[31] J. Pool, I. Wong, and D. Lie. Relaxed Determinism: Making Redundant Execution on Multiprocessors Practical. In *HotOS*, 2007.

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

[33] F. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4), 1990.

[34] K. Sen. Race Directed Random Testing of Concurrent Programs. In *PLDI*, 2008.

[35] J. Slember and P. Narasimhan. Static Analysis Meets Distributed Fault-Tolerance: Enabling State-Machine Replication With Nondeterminism. In *HotDep*, 2006.

[36] J. Slye and E. Elnozahy. Supporting Nondeterministic Execution in Fault-Tolerant Systems. In *FTCS*, 1996.

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

[38] A. Thomson and D. Abadi. The Case for Determinism in Database Systems. In *VLDB*, 2010.

[39] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *ASPLOS*, 2011.

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