Deterministic Scaling

Gabriel Southern Madan Das Jose Renau

Dept. of Computer Engineering, University of California Santa Cruz
{gsouther, madandas, renau}@soe.ucsc.edu
http://masc.soe.ucsc.edu

ABSTRACT

Deterministic execution can simplify development of multithreaded applications by ensuring that the same input produces the same output. However, current proposals only enforce deterministic execution when run on the same system. We propose deterministic scaling in which an application runs with a fixed number of threads regardless of the number of cores in the target system. Our evaluation shows that in many cases this introduces less than 20% additional overhead.

1. INTRODUCTION

Multithreaded applications are notoriously error-prone and difficult to test [17, 18]. Programmers must consider all possible ways that threads can interleave during execution, and use appropriate synchronization techniques to prevent thread interleavings which produce incorrect results. In addition, while multithreaded applications are typically developed to improve performance, the performance benefit of increasing the number of threads is often unpredictable.

Researchers have proposed various *deterministic execution* [1, 3, 5, 7, 12, 14, 16, 19] systems that seek to address problems associated with arbitrary thread interleavings in traditional multithreaded systems. These systems constrain thread interleavings in ways that produce deterministic results, but often the thread interleavings depend on system parameters that may not be obvious or easily controlled by the application programmer. In addition, none of these systems has evaluated how the number of threads spawned by an application can change its behavior.

In this paper we propose running applications with a fixed number of threads in order to achieve stronger deterministic execution semantics. We selected the number of threads to allow for best scalability by finding the best-performing configuration on the system with the most cores. We then used that number of threads on systems with fewer cores. Of the 12 configurations we evaluated, 8 introduced less than 20% additional overhead. We also characterize existing deterministic execution proposals based on the determinism guarantees that they provide to programmers.

The rest of the paper is organized as follows: Section 2 introduces a categorization of deterministic execution systems; Section 3 describes our proposal for selecting the number of threads that an application should spawn; Section 4 describes our experimental setup; Section 5 shows our results; Section 6 surveys related work; and Section 7 concludes the paper.

2. DETERMINISM CLASSIFICATION

Shared memory multiprocessor systems allow multiple processors to access any physical address in the system's memory, and multithreaded programming models typically allow multiple threads to access any address in a process's virtual memory space. As a result the sequence of memory accesses in a parallel application can overlap in arbitrary and non-deterministic ways. When threads in parallel applications communicate with each other they should use some sort of synchronization mechanism to prevent errors such as atomicity violations or order violations. When threads share data without using proper synchronization, there is a data race, while a program that has all memory accesses properly synchronized is data-race free.

Additional sources of non-determinism come from the operating system and program input. These sources of non-determinism can be called *external non-determinism* [6] and are not addressed in this paper. The sources of non-determinism which this paper considers are *internal non-determinism* which result from the multiple different thread interleavings which are possible when running a multithreaded application. A single-threaded application has no internal non-determinism and this papers classifies systems in comparison to the single-threaded case which executes deterministically.

Previous work on deterministic execution [12, 19] has differentiated between *weak determinism* and *strong determinism*, where weak determinism provides deterministic execution only for programs that are data-race-free, while strong determinism enforces deterministic execution even when there are data races.

However, we argue that this categorization is incomplete because application programmers typically do not program at the level of memory accesses. As a result, a program running under a system that enforces strong determinism may find that minor changes to the system environment or compilation options change program behavior.

We propose grouping deterministic execution systems into the following categories: synchronization determinism; machine code determinism; program determinism; and semantic determinism. Examples of systems which implement these types of determinism are shown in Table 1.

In general, enforcing deterministic execution requires restricting the order of communicating memory operations between threads to a single order. A popular technique is to divide the program into serial and parallel modes of execution. During parallel mode, threads operate primarily on private data, while during serial mode the threads synchronize with each other. Since a single deterministic ordering of

Category	Examples
Synchronization Determinism	Kendo [19], Cilk [21]
Machine Code Determinism	DMP [12], CoreDet [3], RCDC [13], Calvin [14]
Program Determinism	Grace [7], DThreads [16]
Semantic Determinism	DPJ [9], NESL [8], Jade [22], TLS [15]

Table 1: Deterministic Execution Categories

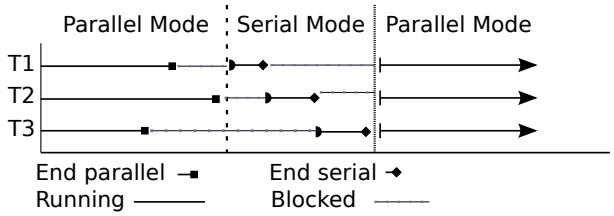

Figure 1: Execution divided into parallel and serial sections

threads is enforced during serial mode, this allows the system as a whole to enforce deterministic execution. However, in order to have practical implementations, existing systems have limits on the determinism guarantees which they provide. Figure 1 illustrates the general technique of dividing execution in serial and parallel modes, and the following subsections describe implementation issues.

2.1 Synchronization Determinism

Synchronization determinism is based on the observation that for a data-race-free program, all communication between threads is constrained by synchronization operations, and enforcing a deterministic ordering of synchronization operations provides determinism for the program as a whole. Kendo [19] is a prototype implementation of this idea; however, it has three important limitations. First it only guarantees determinism for data-race-free programs; although data-race-free is a desirable property, it is not enforced by hardware or software and can be a source of bugs. Second, the Kendo prototype does not provide support for atomic operations. Even a language like C++, which only defines semantics for data-race-free programs, has support for atomic operations [2]. Finally, the division between parallel and serial modes in Kendo is based on the number of retired stores. This metric depends heavily on the machine and compiler settings and can change in ways that are not intuitive to application programmers. It also limits portability across different architectures.

2.2 Machine Code Determinism

Machine code determinism enforces a deterministic ordering of communicating memory operations between threads even in the presence of data races. However, the interleaving of communication points (serial mode in most implementation) depends on the type, number, and communication patters of instructions of machine code instructions in the program binary.

Several prototype systems have implemented this idea using hardware support [12, 13, 14] or with software only [3].

These systems divide execution between serial and parallel modes and use the number of instructions executed as the boundary between the two modes. The number of instructions executed by an application depends on compiler and system options that might not be obvious to the application programmer. As a very simple example, *dead store elimination* is a compiler optimization that changes the number of instructions executed. As a result, application programmers do not know how changes in program source code or compilation settings might change the interleaving of parallel and serial modes used to enforce determinism. The programmer therefore must assume that any source code change could change program behavior.

This category can be further sub-divided between systems where determinism depends on microarchitectural details (such as cache line size) and those which depend only on details exposed through the ISA. We think that this distinction is not significant from the perspective of an application programmer, because even if the application only depends on the ISA, the deterministic execution properties will be closely tied to the target system. We think this form of determinism will be more useful for embedded systems which have hardware/software codesign than for a general purpose computer.

2.3 Program Determinism

Program determinism enforces deterministic execution of a program based on explicit source code synchronization operations rather then implicit characteristics the compiled machine code. For example a `mutex_lock` operation could indicate the division between communicating and isolated modes. This prevents the deterministic execution properties of the application from changing when an application is compiled for different hardware architectures. However, it also means that the programmer must ensure there are enough synchronization points to prevent over serialization or thread imbalance from preventing threads from running when they make communicating writes.

Grace [7] and DThreads [16] are examples of these systems. Grace supports only fork-join-type parallelism, where the programmer divides work among multiple threads and then has the master thread wait for all child threads to complete before continuing. While Grace supports a limited subset of pthread directives, DThreads provides replacement functions for all of the pthread directives. DThreads uses synchronization operations (pthread directives) as the division between serial and parallel modes of execution. This provides a significant advantage for application programmers trying to understand how source code changes affect system determinism. It also allows for portability across architectures and for changes in compilation options that don't change deterministic execution properties.

However, the overall deterministic execution properties are linked to the order of synchronization operations. The order of operations depends of course on application input, but in many cases it also depends on the number of processors in the system because parallel applications spawn a number of threads related to the number of processors. The total number and sequence of synchronization operations will depend on the number of threads that are spawned which themselves execute synchronization operations. This is a significant difference from the simple single-threaded semantics that programmers are familiar with and requires testing, or at least planning, for a range of different numbers

Benchmark	Threads	Speedup
Blackscholes	54	5.5
Canneal	33	6.5
Dedup	27	4.0
Ferret	33	5.1
Streamcluster	14	6.5
Swaptions	64	35.9

Table 2: Number of threads and associated best speedup on 48-core system

of threads when developing multithreaded applications.

2.4 Semantic Determinism

We argue that the goal of deterministic execution should be to provide *semantic determinism*. For sequential applications this is implicit, and given the same input, the program will produce the same output unless it calls an explicitly non-deterministic operation (e.g. `rand()`). Deterministic Parallel Java (DPJ) [9] allows programmers to provide annotations which are used by the compiler to determine what code can safely execute in parallel and to be deterministic by default. However, this is a language-level solution and requires more effort from the programmer than system-level solutions. Automatic parallelization, Thread Level Speculation (TLS) [15], or other systems which convert sequential code to parallel code can be considered as a form of semantic determinism since they maintain the same semantics as a single-threaded application. However, these system are limited in their ability to provide program speedup, in part because they do not provide a mechanism for applications programmers to explicitly express parallelism in their code.

3. SEMANTIC DETERMINISM

We propose combining a system that spawns a fixed number of threads with one that uses source-code level synchronization points between tasks (such as DThreads). Together these techniques allow semantic determinism to be enforced by a runtime system for multithreaded programs written using familiar programming directives (such as pthreads).

We used a simple heuristic to select the number of threads that an application should spawn. We ran each application with DThreads and varied the number of threads spawned on a 48-core system. Afterwards we selected the number of threads which provided the greatest speedup as the number of threads that the application should spawn whenever it ran. The selected number of threads along with the associated speedup on a 48-core system is shown in Table 2.

We also ran each benchmark on a 4-core system and a 16-core system and compared the performance of the optimal number of threads on each benchmark with the performance using the number of threads from the 48-core system (shown in Table 2). We describe our results in more detail in Section 5; however, Figure 2 shows that for many of the configurations this proposal has limited additional overhead.

4. SETUP

To test our proposal of running applications with a fixed number of threads we evaluated the scalability of 6 PARSEC benchmarks on three different systems with 4, 16, and 48 cores. We ran all of the PARSEC workloads that DThreads

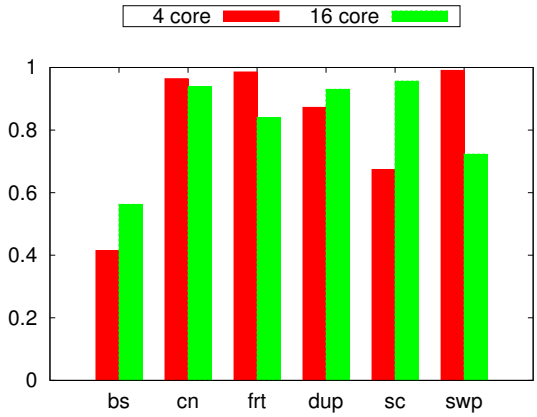

Figure 2: Fixed thread speedup fraction of maximum speedup

Cores	CPU	Memory
4	Intel Core i5-2500K	8 GB
16	4 x Intel Xeon X7350	32 GB
48	4 x AMD Opteron 6172	64 GB

Table 3: Evaluation systems

currently works with (Liu et al. [16] explain the limitations in the prototype systems which prevent the remaining PARSEC workloads from running with DThreads).

The configuration of our experimental systems is shown in Table 3. We used the *native* input sets when running canneal, dedup, streamcluster, and swaptions. For blackscholes and ferret we used the *simlarge* input sets because DThreads could not allocate enough memory for the native input sets. For these benchmarks the execution time for a single thread was 5 seconds or less, so OS scheduling effects caused noticeable variation in the results. We ran these two benchmarks 20 times for each of the data points we collected and averaged the runtimes when generating our results. For the other benchmarks the runtimes were longer, so we ran them 5 times each and averaged the results. PARSEC allows users to specify the *minimum* number of threads that a benchmark will spawn, but some benchmarks spawn more. In particular for the workloads that we evaluated when specifying n threads, ferret spawns $1+2+4n$ threads, and dedup spawns $1+2+3n$ threads. The other workloads spawn $1+n$ threads. We ran all benchmarks to completion and report speedups relative to the case of spawning a single thread ($n = 1$) on each machine.

5. EVALUATION

Figure 2 shows the speedup when running with the number of threads that give the maximum speedup for the 48-core system. Overall, the results show that running applications with a fixed number of threads can provide benefits for deterministic execution. In many cases there is minimal overhead for running a benchmark with many threads on a 4-core system. Even though this causes the system to be heavily overcommitted, it still has good performance. Likewise, few of the benchmarks really benefit from running with 48 cores or more on the 48-core system. In most cases the additional core provides only a marginal speedup, at

the expense of significantly more system resources. Several of the benchmarks have non-linear scaling, which suggests that a good strategy could be to optimize the performance for the number of threads which give a good performance / resources tradeoff and then always use that number of threads. This strategy can provide both deterministic execution and predictable performance. Finally, for the benchmarks that don't scale well when overcommitted, the best option probably is to make changes in the application source code when targeting a deterministic execution environment. Deterministic execution environments tend to be more restrictive than typical shared-memory programming and in some cases optimizations that work well for general unconstrained shared-memory perform poorly in a deterministic execution environment.

5.1 Characterization

Our results are shown in Figure 3. For each benchmark we have calculated the speedup for each machine relative to the execution time of the benchmark with a single thread.

Blackscholes: This benchmark is divided into a parallel section which has no sharing between threads, and a single threaded section which performs initialization. As a result the overall scalability is limited by the serial portion of the program. When we ran blackscholes on the 48-core system the best speedup was 5.5 times when spawning 54 threads. However, spawning this many threads meant that the benchmark was heavily overcommitted when running on the 4-core and 16-core systems. The benchmark runtime was also very short (less than 0.5 seconds) which caused the overhead associated with spawning threads on an overcommitted system to be more significant as a percentage of overall execution time. Blackscholes performed worst, out of the benchmarks we evaluated, in the overall speedup shown in Figure 2. However, the maximum speedup for the 48-core configuration was relatively small; consequently, spawning a large number of threads did not lead to the most efficient use of systems resources. In this case a more complex heuristic for selecting the number of threads to run may be useful. And running the benchmark with a smaller but still fixed number of threads could provide a better performance tradeoff while retaining the benefits of deterministic execution.

Canneal: This benchmark is an excellent candidate for running with a fixed number of threads. The overall speedup for the 4-core and 16-core systems is very stable even when the systems are heavily overcommitted. The maximum speedup for the 48-core system occurs when 33 threads are spawned, and spawning 33 threads for the 4-core and 8-core systems results in less than 5% slowdown compared to the optimal configuration for each of those systems.

Dedup: This benchmark uses a pipelined model of execution and it has a wide variation in speedup. Here spawning more threads can lead to much worse performance even when the system is not overcommitted. The reason is that if the workload distribution between stages is unbalanced this slows down the execution of the entire program because of implicit serialization of work. Selecting the optimal number of threads for the 48-core system is not completely optimal for the 4- and 8-core systems. But it still provides good performance which is much better than one of the poor performance configurations would be.

Ferret: This is another benchmark like canneal that has no performance degradation even when heavily overcommitted. Running with a fixed number of threads provides the benefits of deterministic execution without any accompanying overhead for overcommitted systems.

Streamcluster: This benchmark has the worst performance for overcommitted workloads. However, because of the difficultly that streamcluster has with scaling when running with DThreads the best performing configuration for the 48-core system is when 14 threads are spawned. This relatively small number of threads still provides acceptable performance for the 4-core system and is near optimal for the 16-core system. The reason that streamcluster does not scale is that it uses custom barriers which employ busy-waiting to synchronize threads. While busy-waiting can be useful for fast synchronization when running on a non-deterministic system with extra resources, it is not a good choice for an overcommitted system or for a deterministic execution runtime. Thus streamcluster is an example of a type of programming that is not well suited for deterministic execution systems.

Swaptions: This benchmark shows good scalability with minimal overhead for overcommitted systems. One interesting note about the benchmark is that when spawning a large number of threads the performance change appears to follow a step function. Previous work on analyzing PARSEC workloads [20, 23] identified an imbalance in the distribution of work between threads as the reason for this uneven scaling. This is an example of how application performance can vary in unpredictable ways when adding more threads. Thus running with a fixed number of threads provides benefits both for program correctness and for providing predictability about performance characteristics.

6. RELATED WORK

We have categorized existing deterministic execution systems and evaluated the scalability of DThreads with respect to the number of threads used per application.

Bergan, et al. [4] evaluated the strengths and weaknesses of a variety of deterministic execution systems. This work has similarities to ours in identifying ways that system parameters affect determinism, but it does not evaluate the classifications we propose.

Devietti [11] examines the interaction between deterministic execution runtime systems and programming languages which enforce deterministic execution. In particular the thesis proposes MELD, which studies how to combine DPJ with a deterministic runtime system such as DMP.

Cui et al. have proposed Peregrine [10] as a way to enforce efficient deterministic execution. It has some similarities to our proposal because it seeks to eliminate implicit inputs by recording and reusing schedules that have been verified. However, Peregrine does not provide a clear link between source code semantics and the execution schedule that will be enforced.

7. CONCLUSION AND FUTURE WORK

We have demonstrated that having applications spawn a fixed number of threads while running with DThreads introduces limited overhead. The advantage of this technique is that it provides an application programmer with stronger deterministic execution guarantees, which we call semantic determinism. Our proposal had less than 20% additional overhead on 8 of the 12 benchmarks that we evaluated.

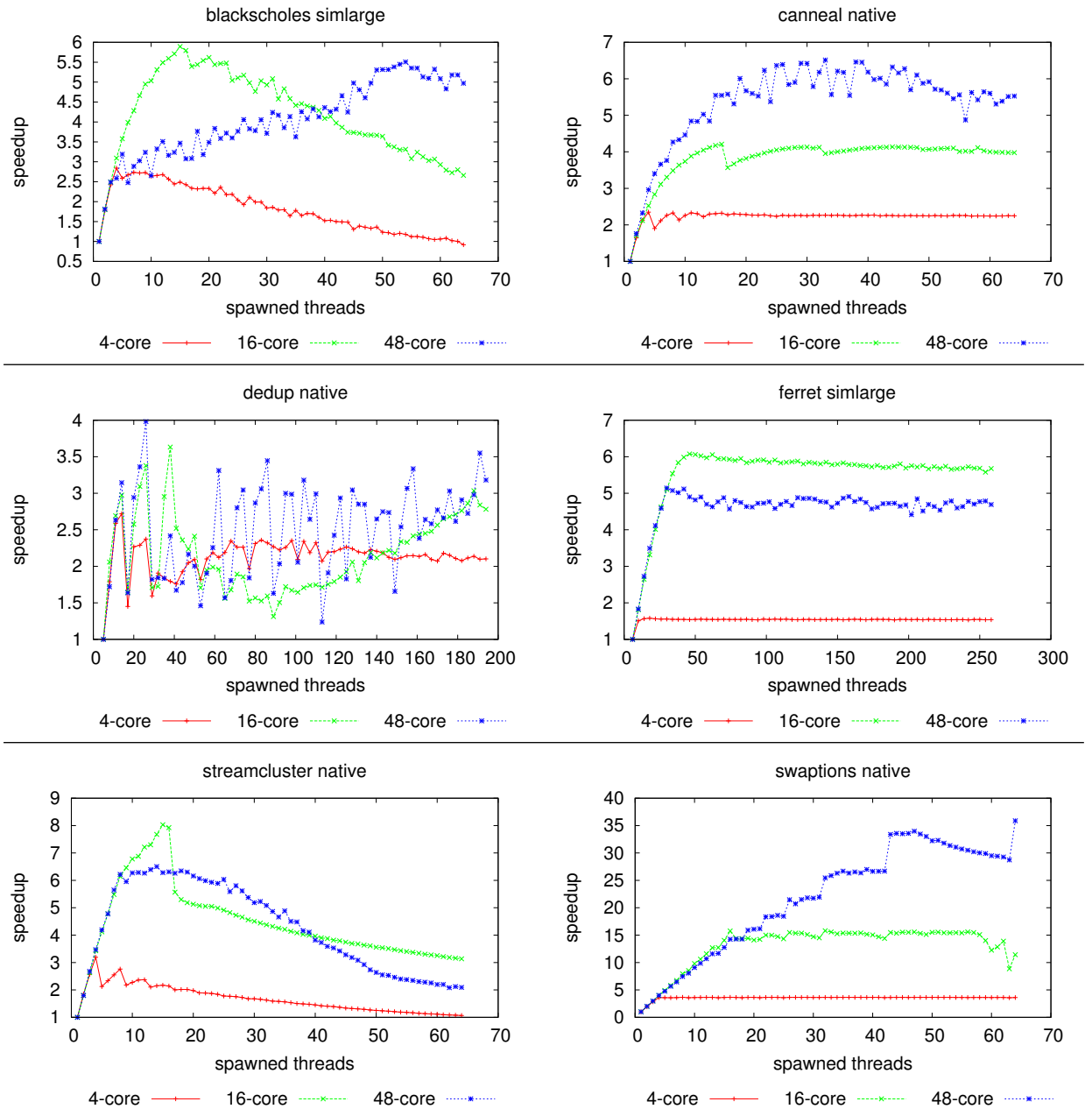

Figure 3: Scalability results

Although this proposal has shown good results on the set of benchmarks we evaluated, there are several PARSEC benchmarks that DThreads is not able to run. In the future we plan to evaluate the scalability of DThreads on a larger set of workloads and see if limitations in the existing system can be solved.

In addition, we believe that the technique of spawning a fixed number of threads can be valuable for other deterministic execution runtime environments.

Research in deterministic execution remains a new but promising way to improve the usability of parallel applications. However, we argue that researchers should target system-independent solutions that enforce what we call semantic determinism, as work continues on solving limitations in existing systems.

8. REFERENCES

[1] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. OSDI'10, 2010.

[2] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing c++ concurrency. POPL '11, 2011.

[3] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. ASPLOS '10, 2010.

[4] T. Bergan, J. Devietti, N. Hunt, and L. Ceze. The deterministic execution hammer: How well does it actually pound nails? WODET'11, 2011.

[5] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dos. OSDI'10, 2010.

[6] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dos. OSDI '10. USENIX Association, 2010.

[7] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for c/c++. OOPSLA '09, 2009.

[8] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 1996.

[9] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. OOPSLA '09, 2009.

[10] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient deterministic multithreading through schedule relaxation. SOSP '11, pages 337–351, 2011.

[11] J. Devietti. *Deterministic Execution for Arbitrary Multithreaded Programs*. PhD thesis, University of Washington, 2012.

[12] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. Dmp: deterministic shared memory multiprocessing. ASPLOS '09, 2009.

[13] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. Rcdc: a relaxed consistency deterministic computer. ASPLOS '11, 2011.

[14] D. Hower, P. Dudnik, M. Hill, and D. Wood. Calvin: Deterministic or not? free will to choose. HPCA '11, 2011.

[15] T. Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, 1986.

[16] T. Liu, C. Curtsinger, and E. D. Berger. Dthreads: efficient deterministic multithreading. SOSP '11, 2011.

[17] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. ASPLOS XIII, 2008.

[18] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. OSDI'08, 2008.

[19] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. ASPLOS '09, 2009.

[20] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Thread reinforcer: Dynamically determining number of threads via os level monitoring. IISWC '11, pages 116–125, 2011.

[21] K. H. Randall. *Cilk: E cient Multithreaded Computing*. PhD thesis, Massachusetts Institute of Technology, 1998.

[22] M. C. Rinard. *The design, implementation and evaluation of Jade: a portable, implicitly parallel programming language*. PhD thesis, Stanford University, 1994.

[23] M. Roth, M. Best, C. Mustard, and A. Fedorova. Deconstructing the overhead in parallel applications. IISWC '12, 2012.