Deterministic Consistency: A Programming Model for Shared Memory Parallelism

Amittai Aviram and Bryan Ford
Yale University

Draft of 2010/02/01 16:29

Abstract

The difficulty of developing reliable parallel software is generating interest in *deterministic* environments, where a given program and input can yield only one possible result. Languages or type systems can enforce determinism in new code, and runtime systems can impose synthetic schedules on legacy parallel code. To parallelize existing serial code, however, we would like a programming model that is naturally deterministic without language restrictions or artificial scheduling. We propose *deterministic consistency*, a parallel programming model as easy to understand as the "parallel assignment" construct in sequential languages such as Perl and JavaScript, where concurrent threads always read their inputs before writing shared outputs. DC supports common data- and task-parallel synchronization abstractions such as fork/join and barriers, as well as non-hierarchical structures such as producer/consumer pipelines and futures. A preliminary prototype suggests that software-only implementations of DC can run applications written for popular parallel environments such as OpenMP with low ($< 10\%$) overhead for some applications.

1 Introduction

For decades, the "gold standard" in multiprocessor programming models has been sequentially consistent shared memory [25] with mutual exclusion [20]. Alternative models, such as explicit message passing [29] or weaker consistency [17], usually represent compromises to improve performance without giving up "too much" of the simplicity and convenience of sequentially consistent shared memory. But are sequential consistency and mutual exclusion really either *simple* or *convenient*?

In this model, we find that slight concurrency errors yield subtle heisenbugs [27, 28] and security vulnerabilities [34]. Data race detection [16, 30] or transactional memory [19, 32] can help ensure mutual exclusion, but even "race-free" programs may have heisenbugs [2]. Heisenbugs result from *nondeterminism in general*, a realization that has inspired new languages that ensure determinism through communication constraints [33] or type systems [7]. But to parallelize the vast body of sequential code for new multicore systems, we would like a programming model that is simple, convenient, deterministic, and compatible with existing languages.

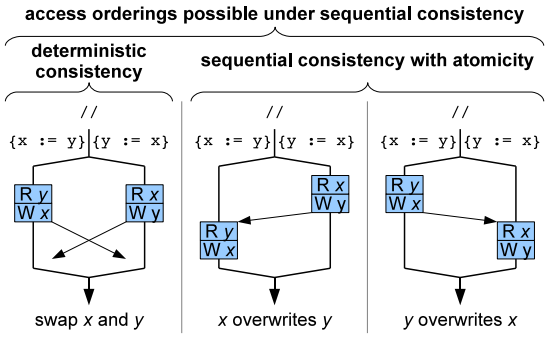

Figure 1: Deterministic versus sequential consistency

To this end, we propose a new memory model called *deterministic consistency* or DC. In DC, concurrent threads logically share an address space but never see each others' writes, except when they synchronize explicitly and deterministically. To illustrate DC, consider the "parallel assignment" operator in many sequential languages such as Python, Perl, Ruby, and JavaScript, with which one may swap two variables as follows:

```
x,y := y,x
```

This construct implies no actual parallel execution: the statement merely evaluates all right-side expressions (in some order) before writing their results to the left-side variables. Now consider a "truly parallel" analog, using Hoare's notation for fork/join parallelism [20]:

```
{x := y} // {y := x}
```

This statement forks two threads, each of which reads one variable and then writes the other; the threads then synchronize and rejoin. As Figure 1 illustrates, under sequential consistency, this parallel statement may swap the variables or overwrite one with the other, depending on timing. Making each thread's actions atomic, by enclosing the assignments in critical sections or transactions, eliminates the swapping case but leaves a nonde-

terministic choice between x overwriting y and y overwriting x. How popular would the former "parallel assignment" construct be if it behaved in this way? Deterministic consistency, in contrast, reliably behaves like a parallel assignment: each thread reads all inputs before writing any shared results.

Like release consistency [17], DC distinguishes *ordinary* reads and writes from *synchronization* operations and classifies the latter into *acquires* and *releases*, which determine at what point one thread sees (acquires) results produced (released) by another thread. DC ensures determinism by requiring that (1) program logic uniquely pairs each acquire with a matching release, (2) only an intervening acquire/release pair makes one thread's writes visible to another thread, and (3) acquires handle conflicting writes deterministically. Unlike most memory models, reads never conflict with writes in DC: the swapping example above contains no data race. A natural way to understand DC—and one way to implement it—is as a distributed shared memory [1, 24] in which a release explicitly "transmits" a message containing memory updates, and the matching acquire operation "receives" and integrates these updates locally.

DC supports not only block-structured synchronization abstractions such as the fork/join, barrier, and task constructs of OpenMP [6], but also non-hierarchical synchronization patterns such as dynamic producer/consumer graphs and inter-thread queues. DC can emulate nondeterministic synchronization constructs in existing parallel code via techniques such as deterministic scheduling [3, 4, 12], but for new or newly parallelized code, we develop deterministic alternatives for common idioms such as pipelines and futures. A prototype in progress promises to be flexible and efficient enough for a variety of parallel applications.

Section 2 defines DC at a low level, and Section 3 explores its use in high-level environments like OpenMP. Section 4 outlines implementation issues, Section 5 discusses related work, and Section 6 concludes.

2 Deterministic Consistency

Since others have eloquently made the case for deterministic parallelism [7, 27], we will take its desirability for granted and focus on deterministic consistency (DC). This section defines the basic DC model and its low-level synchronization primitives, leaving the model's mapping to high-level abstractions to the next section.

2.1 Defining Deterministic Consistency

As in release consistency (RC) [17, 24], DC separates normal data accesses from synchronization operations and classifies the latter into *release*, where a thread makes recent state changes available for use by other threads, and *acquire*, where a thread obtains state

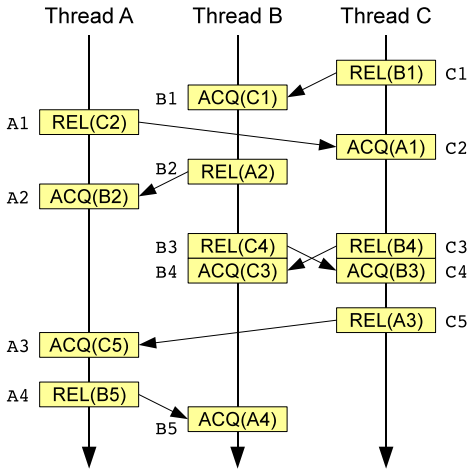

Figure 2: Example synchronization trace for three threads with labeled and matched release/acquire pairs

changes made by other threads. A thread performs a release when forking a child thread or leaving a barrier, for example, and an acquire when joining with a child or entering a barrier. As in RC, synchronization operations in DC are sequentially consistent relative to each other, and these synchronization operations determine when a normal write in one thread must become visible to a normal read in another thread: namely, when an intervening chain of acquire/release pairs connects the two accesses in a "happens-before" synchronization relation.

While RC relaxes the constraints of sequential consistency [25], allowing an even wider range of nondeterministic orderings, DC in turn tightens RC's constraints to permit only one unique execution behavior for a given parallel program. DC ensures determinism by adding three new constraints to those of RC:

1. Program logic must uniquely pair release and acquire operations, so that each release "transmits" updates to a specific acquire in another thread.
2. One thread's writes never become visible to another thread's reads until mandated by synchronization: i.e., writes propagate "as slowly as possible."
3. If two threads perform conflicting writes to the same location, the implementation handles the conflict deterministically at the relevant acquire.

Constraint 1 makes synchronization deterministic by ensuring that a release in one thread always interacts with the same acquire in some other thread, at the same point in each thread's execution, regardless of execution speeds. A program might in theory satisfy this constraint by specifying each synchronization operation's "partner" explicitly through a labeling scheme. If each thread has a unique identifier T, and we assign each of T's synchronization actions a consecutive inte-

ger N, then a (T, N) pair uniquely names any synchronization event in a program's execution. The program then invokes synchronization primitives of the form `acquire(`T_r, N_r`)` and `release(`T_a, N_a`)`, where (T_r, N_r) names the `acquire`'s partner `release` and vice versa. Figure 2 illustrates a 3-thread execution trace with matched and labeled acquire/release pairs. We suggest this scheme only to clarify DC: explicit labeling would be an unwelcome practical burden, and Section 3 discusses more convenient high-level abstractions.

Constraint 2 makes normal accesses deterministic by ensuring that writes in a given thread become visible to reads in another thread at only one possible moment. Release consistency already requires a write by thread T_1 to become visible to thread T_2 *no later* than the moment T_2 performs an acquire directly or indirectly following T_1's next release after the write. RC permits the write to become visible to T_2 *before* this point, but DC requires the write to propagate to T_2 at *exactly* this point. By delaying writes "as long as possible," DC ensures that non-conflicting normal accesses behave deterministically while preserving the key property that makes RC efficient: it keeps parallel execution as independent as possible subject to synchronization constraints.

DC's third constraint affects only programs with data races. If both threads in Figure 1 wrote to the *same* variable before rejoining, for example, DC requires the join to handle this race deterministically. Since data races usually indicate software bugs, one response is to throw a runtime exception. Other behaviors, e.g., prioritizing one write over the other, would not affect correct programs but may be less helpful with buggy code.

2.2 Why DC is Deterministic

To clarify why the above rules adequately ensure deterministic execution in spite of arbitrary parallelism, we briefly sketch a proof of DC's determinism.

Theorem: A parallel program whose sequential fragments execute deterministically, and whose memory access and synchronization behavior conforms to the rules in Section 2.1, yields at most one possible result.

Proof Sketch: Assume each synchronization operation explicitly names its "partner" as described above. Suppose we implement DC by accumulating memory "diffs" and passing them at synchronization points atop a message-passing substrate, as in distributed shared memory [1, 24]. Assume the substrate provides an unlimited number of buffered message channels, each with a unique name of the form (T_r, N_r, T_a, N_a). When a thread T_r invokes a `release(`T_a, N_a`)` operation labeled (T_r, N_r), T_r sends all diffs it has accumulated so far on channel (T_r, N_r, T_a, N_a). Similarly, when thread T_a invokes an `acquire(`T_r, N_r`)` operation labeled (T_a, N_a), it receives a set of diffs on channel (T_r, N_r, T_a, N_a) and applies those it does not already have. Since each channel (T_r, N_r, T_a, N_a) is used by only one sender T_r and one receiver T_a, the resulting system forms a Kahn process network [23], and DC's determinism follows from that of Kahn networks.

3 High-level Synchronization

We are developing *DOMP*, a variant of OpenMP [6] with deterministic consistency. DOMP retains OpenMP's language neutrality and convenience, supporting most OpenMP constructs except for fundamentally nondeterministic ones, and extending OpenMP to support general reductions and non-hierarchical dependency structures.

Fork/Join: OpenMP's foundation is its `parallel` construct, which forks multiple threads to execute a parallel code block and then rejoins them. Fork/join parallelism maps readily to DC, as shown in Figure 3(a): on fork, the parent releases to an acquire at the birth of each child; on join, the parent acquires the final results each child releases at its death. OpenMP's work-sharing constructs, such as parallel `for` loops, merely affect each child thread's actions within this fork/join model.

Barrier: At a barrier, each thread releases to each other thread, then acquires from each other thread, as in Figure 3(b). Although we view an n-thread barrier as $n-1$ releases and acquires per thread, DOMP avoids this n^2 cost using "broadcast" release/acquire primitives, which are consistent with DC as long as each release matches a well-defined *set* of acquires and vice versa.

Ordering: OpenMP's `ordered` construct orders a particular code block within a loop by iteration while permitting parallelism in other parts. DOMP implements this construct using a chain of acquire/release pairs among worker threads, as shown in Figure 3(c).

Reductions: OpenMP's `reduction` attributes and `atomic` constructs enable programs to accumulate sums, maxima, or bit masks efficiently across threads. OpenMP unfortunately supports reductions only on simple scalar types, leading programmers to serialize complex reductions unnecessarily via `ordered` or `critical` sections or locks. All uses of these serialization constructs in the NAS Parallel Benchmarks [21] implement reductions, for example. DOMP therefore provides a generalized `reduction` construct, by which a program can specify a custom reduction on pairs of variables of any matching types, as in this example:

```
#pragma omp reduction(a:a1,b:b1,c:c1)
  { a += a1; b = max(b,b1);
    if (c1.score > c.score) c = c1; }
```

DOMP accumulates each thread's partial results in thread-private variables and reduces them at the next join

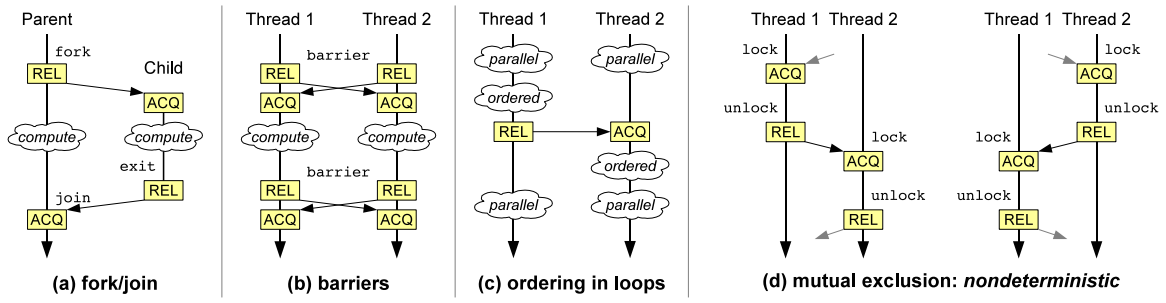

Figure 3: Mapping of High-level Synchronization Operations to Acquire/Release Pairs

or barriar via combining trees, improving both convenience and scalability over serialized reduction.

Tasks: OpenMP 3.0's `task` constructs express a form of fork/join parallelism suited to dynamic work structures. Since DC rules prevent a task from seeing any writes of other tasks until it completes and synchronizes at a barrier or `taskwait`, DOMP eliminates OpenMP's risk of subtle bugs if one task uses shared inputs that are freed or go out of scope in a concurrent task.

DOMP extends OpenMP with explicit *task objects*, with which a `taskwait` construct can name and synchronize with a particular `task` instance independently of other tasks, in order to express futures [18] or non-hierarchical dependency graphs [15] deterministically:

```
omp_task mytask;
#pragma omp task(mytask)
  { ...task code... }
...other tasks...
#pragma omp taskwait(mytask)
```

Mutual exclusion: Unlike `ordered`, which specifies a *particular* sequential ordering, mutual exclusion facilities such as `critical` sections and locks imply an arbitrary, *nondeterministic* ordering. Mutual exclusion violates Constraint 1 in Section 2.1 because it permits multiple acquire/release pairings, as illustrated in Figure 3(d). While DOMP could emulate mutual exclusion via deterministic scheduling, we prefer to focus on developing deterministic abstractions to replace common uses of mutual exclusion, such as general reductions.

Flush: Some OpenMP programs implement custom synchronization structures such as pipelines using the `flush` (memory barrier) construct in spin loops. Like mutual exclusion, DOMP omits support for such constructions, in favor of expressing dependency graphs such as pipelines deterministically using task objects.

4 Implementing DC

We have built an early user space prototype implementing DC with a pthreads-like fork/join API. The prototype encouragingly shows less than 10% overhead on the coarse-grained PARSEC benchmarks [5] Blackscholes and Swaptions. Finer-grained benchmarks such as Streamcluster currently show high overheads, but many optimization opportunities remain. The rest of this section outlines key challenges and opportunities in implementing deterministic consistency, for both shared memory multithreaded programs and multiprocess systems.

4.1 Shared Memory Challenges

Memory Access Isolation: Since DC requires one thread's writes to remain invisible to a second thread until the two threads synchronize, the threads must effectively execute in separate "workspaces" between synchronization events. Virtual memory and write-sharing techniques like those used to implement lazy release consistent distributed shared memory [1] should apply to DC. Memory accesses may also be isolated via instruction-level rewriting [3], possibly reducing the cost of synchronization operations at the expense of adding overhead to all ordinary memory accesses. Hardware support [12, 17] could mitigate the performance cost of isolation, but is unlikely to appear in commodity hardware unless software-based approaches first demonstrate deterministic parallelism to be viable and compelling.

Shared Resources: Shared resources in current environments implicitly introduce nondeterminism through mutual exclusion: calling `malloc()` concurrently in multiple threads may yield different pointers depending on execution timing, for example, and the file descriptor number returned by a call to Unix's `open()` may have similar timing dependencies on other threads' file descriptor operations. The `malloc()` problem may be addressed by assigning each thread a separate virtual memory address range and allocation pool from which to satisfy `malloc()` requests; such an allocator may also benefit scalability. The file descriptor table problem might be addressed by using higher-level equivalents such as `fopen()` that do not imply mutual exclusion. These approaches do not address shared resources outside the application process, however, such as reads and writes to shared files in an external file system.

4.2 Beyond Shared Memory

While we have focused on the intra-process shared memory abstraction, DC may also be applicable at the system level for state shared among processes. Standard operating systems, for example, commonly give all processes sequentially consistent access to a globally shared file system (though network file systems often relax consistency somewhat). This design yields the same problems of nondeterminism and heisenbugs at inter-process level that we see within multithreaded programs: we find often that a large software source tree builds reliably under a sequential '`make`' but fails nondeterministically under a parallel '`make -j`' command, for example.

In place of sequential consistency, an OS might provide a deterministically consistent file system to processes, enabling a multi-process computation to run deterministically even as processes share state by reading and writing files. If a parallel `make` forks off two compiler instances running in parallel, for example, each compiler would execute in its own private virtual copy of the file system until completion; the system would then reconcile the `.o` files produced by each compiler into a single directory once both compilers complete.

There will always be shared resources "outside the reach" of any deterministic environment, whose use will introduce nondeterminism into the program: for example, I/O requests arriving at a network server from its clients. In such cases the only solution may be to accept some nondeterminism, log nondeterministic inputs to enable later replay, or avoid their use entirely.

5 Related Work

DC conceptually builds on release consistency [17] and lazy release consistency [24], which relax sequential consistency's ordering constraints to increase the independence of parallel activities. DC retains these independence benefits, additionally providing determinism by delaying the propagation of any thread's writes to other threads until *required* by explicit synchronization.

Race detectors [16, 30] can detect certain heisenbugs, but only determinism eliminates their possibility. Language extensions can dynamically check determinism assertions in parallel code [10, 31], but heisenbugs may persist if the programmer omits an assertion. SHIM [14, 15, 33] provides a deterministic message-passing programming model, and DPJ [7,8] enforces determinism in a parallel shared memory environment via type system constraints. While we find language-based solutions promising, parallelizing the huge body of existing sequential code will require parallel programming models compatible with existing languages.

DMP [3, 12] uses binary rewriting to execute existing parallel code deterministically, dividing threads' execution into fixed "quanta" and synthesizing an artificial round-robin execution schedule. Since DMP is effectively a deterministic *implementation* of a nondeterministic programming model, slight input changes may still reveal schedule-dependent bugs. Grace [4] runs fork/join-style programs deterministically using virtual memory techniques. These systems still pursue sequential consistency as an "ideal" and rely on speculation for parallelism: if a thread reads a variable concurrently written by another, as in the "swap" example in Section 1, one thread aborts and re-executes sequentially. A partial exception is DMP-B [3], which weakens consistency within a parallel execution quantum. DC, in contrast, keeps threads fully independent between program-defined synchronization points, never requires speculation or rollback, and imposes no artificial execution schedules prone to accidental perturbation.

Replay systems can log and reproduce particular executions of conventional nondeterministic programs, for debugging [11, 26] or intrusion analysis [13, 22]. The performance and space costs of logging nondeterministic events usually make replay usable only "in the lab," however: if a bug or intrusion manifests under deployment with logging disabled, the event may not be subsequently reproducible. In a deterministic environment, any event is reproducible provided only that the original external inputs to the computation are logged.

As with deterministic release consistency, transactional memory (TM) systems [19, 32] isolate a thread's memory accesses from visibility to other threads except at well-defined synchronization points, namely between transaction start and commit/abort events. TM offers no deterministic ordering between transactions, however: like mutex-based synchronization, transactions guarantee only atomicity, not determinism.

6 Conclusion

Building reliable software on massively multicore processors demands a predictable, understandable programming model, a goal that may require giving up sequential consistency and mutual exclusion. Deterministic consistency provides an alternative parallel programming model as simple as "parallel assignment," and supports existing languages and synchronization abstractions.

References

[1] Cristiana Amza et al. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[2] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In *Workshop on Verification and Validation of Enterprise Information Systems (VVEIS)*, pages 82–93, April 2003.

[3] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *15th international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2010.

[4] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for C/C++. In *OOPSLA*, October 2009.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[6] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008. http://www.openmp.org/mp-documents/spec30.pdf.

[7] Robert L. Bocchino Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *1st Workshop on Hot Topics in Parallelism (HotPar '09)*. USENIX, March 2009. http://www.usenix.org/event/hotpar09/tech/full_papers/bocchino/bocchino.pdf.

[8] Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for Deterministic Parallel Java. October 2009. http://dpj.cs.uiuc.edu/DPJ/Publications_files/paper_1.pdf.

[9] Per Brinch Hansen, editor. *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*. Springer-Verlag, Berlin, Germany, 2002.

[10] Jacob Burnim and Koushik Sen. Asserting and checking determinism for multithreaded programs. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, August 2009.

[11] Ronald S. Curtis and Larry D. Wittie. BugNet: A debugging system for parallel programming environments. In *3rd International Conference on Distributed Computing Systems*, pages 394–400, October 1982.

[12] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic shared memory multiprocessing. In *14th international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2009.

[13] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *5th USENIX Symposium on Operating Systems Design and Implementation*, December 2002.

[14] Stephen A. Edwards and Olivier Tardieu. Shim: A deterministic model for heterogeneous embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):854–867, August 2006.

[15] Stephen A. Edwards, Nalini Vasudevan, and Olivier Tardieu. Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads. In *Design, Automation, and Test in Europe*, March 2008.

[16] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[17] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[18] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[19] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th International Symposium on Computer Architecture*, pages 289–300, May 1993.

[20] C. A. R Hoare. Towards a theory of parallel programming. In C. A. R. Hoare and R. H. Perrott, editors, *Operating Systems Techniques: Proceedings of a Seminar at Queen's University*, pages 61–71, New York, New York, USA, 1972. Academic Press. Reprinted in [9], 231–244.

[21] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Ames Research Center, October 1999.

[22] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems prin-*

ciples, pages 91–104, New York, NY, USA, 2005. ACM.

[23] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing*, pages 471–475, Amsterdam, Netherlands, 1974. North-Holland.

[24] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *13th International Symposium on Computer Architecture*, pages 13–21, May 1992.

[25] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[26] Thomas J. Leblanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[27] E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[28] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics. In *13th international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 329–339, March 2008.

[29] Message Passing Interface Forum. MPI: A message-passing interface standard version 2.2, September 2009.

[30] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, and Gerard Basler. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation (OSDI '08)*, pages 267–280, Berkeley, California, USA, 2008. USENIX Association.

[31] Caitlin Sadowski, Stephen N. Freund, and Cormac Flanagan. SingleTrack: A dynamic determinism checker for multithreaded programs. In *18th European Symposium on Programming*, March 2009.

[32] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997.

[33] Olivier Tardieu and Stephen A. Edwards. Scheduling-independent threads and exceptions in SHIM. In *6th Conference on Embedded Software*, pages 142–151, October 2006.

[34] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *1st USENIX Workshop on Offensive Technologies*, August 2007.