Workspace Consistency: A Programming Model for Shared Memory Parallelism

Amittai Aviram, Bryan Ford, and Yu Zhang
Yale University

Abstract

Recent interest in deterministic parallelism has yielded new deterministic programming languages, which offer promising features but require rewriting existing code, and deterministic schedulers, which emulate existing thread APIs but do not eliminate races from the basic programming model. *Workspace consistency* (WC) is a new synchronization and memory consistency model that offers a "naturally deterministic," race-free programming model that can be adopted in both new or existing languages. WC's basic semantics are inspired by—and intended to be as easily understood as—the "parallel assignment" construct in sequential languages such as Perl and JavaScript, where concurrent threads always read their inputs before writing shared outputs. Prototype implementations of a restricted form of WC already exist, supporting only strictly hierarchical fork/join-style synchronization, but this paper develops and explores the model in more detail and extends it to support non-hierarchical synchronization patterns such as producer/consumer pipelines and futures.

1 Introduction

For decades, the "gold standard" in multiprocessor programming models has been sequentially consistent shared memory [29] with mutual exclusion [24]. Alternative models, such as explicit message passing [34] or weaker consistency [21], 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 [31, 33] and security vulnerabilities [41]. Data race detection [20, 35] or transactional memory [23, 39] can help ensure mutual exclusion, but even "race-free" programs may have heisenbugs [3]. Deterministic schedulers [6–8, 16] make heisenbugs reproducible once they manifest, but do not eliminate races or heisenbugs from the programming model: a slight change of input data affecting instruction path lengths can still reveal a race [14].

Heisenbugs fundamentally result from nondeterminism in the parallel programming model. This realization has inspired new languages that ensure determinism through communication constraints [40] or type systems [9]. But to parallelize existing sequential code for new multicore systems, we would like a programming model that is simple, convenient, deterministic, and compatible with existing languages.

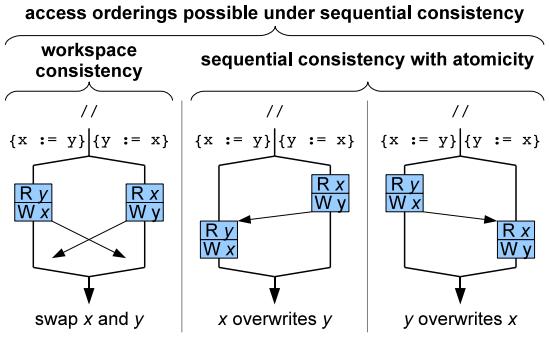

Figure 1: Workspace versus sequential consistency

To this end, we propose a new memory model called *workspace consistency* or WC. In WC, concurrent threads logically share an address space but never see each others' writes, except when they synchronize explicitly and deterministically. To illustrate WC, 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 [24]:

```
{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 nondeterministic choice between `x` overwriting `y` and `y`

overwriting x. How popular would the former "parallel assignment" construct be if it behaved in this way? Workspace consistency, in contrast, defines the behavior corresponding to parallel assignment to be the one and only "correct" behavior: each thread always reads all inputs before writing any shared results.

Like release consistency [21], WC 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. WC 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 WC: the swapping example above contains no data race. A natural way to understand WC—and one way to implement it—is as a distributed shared memory [2, 28] in which a release explicitly "transmits" a message containing memory updates, and the matching acquire operation "receives" and integrates these updates locally.

Determinator [4] already implements and demonstrates the viability of a restricted form of WC, supporting only hierarchical synchronization patterns such as fork/join and barrier. This paper explores approaches to generalizing WC to support non-hierarchical synchronization patterns such as dynamic producer/consumer graphs and inter-thread queues. This approach combines memory update techniques derived from lazy release-consistent distributed shared memory [28] with an underlying synchronization discipline corresponding to message-based Kahn process networks [27].

Section 2 defines WC 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 Workspace Consistency

Since others have eloquently made the case for deterministic parallelism [9, 31], we will take its desirability for granted and focus on workspace consistency (WC). This section defines the basic WC model and its low-level synchronization primitives, leaving the model's mapping to high-level abstractions to the next section.

2.1 Defining Workspace Consistency

As in release consistency (RC) [21, 28], WC 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 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 WC 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.

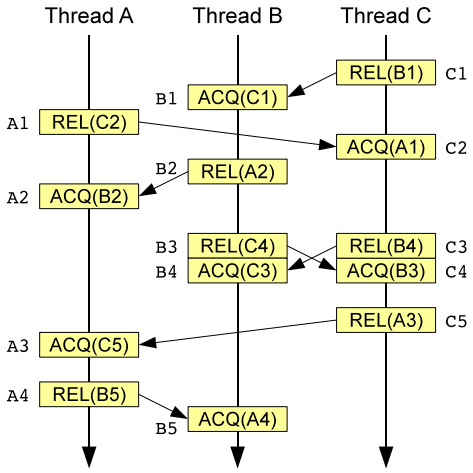

Figure 2: Example synchronization trace for three threads with labeled and matched release/acquire pairs

While RC relaxes the constraints of sequential consistency [29], allowing an even wider range of nondeterministic orderings, WC in turn tightens RC's constraints to permit only one unique execution behavior for a given parallel program. WC 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 integer N, then a (T, N) pair uniquely names any synchro-

nization 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 WC: 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. RC 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 WC requires the write to propagate to T_2 at *exactly* this point. By delaying writes "as long as possible," WC 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.

WC's third constraint affects only programs with data races. If both threads in Figure 1 wrote to the *same* variable before rejoining, for example, WC 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 WC is Deterministic

To clarify why the above rules adequately ensure deterministic execution in spite of arbitrary parallelism, we briefly sketch a proof of WC'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 WC by accumulating memory "diffs" and passing them at synchronization points atop a message-passing substrate, as in distributed shared memory [2, 28]. 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 [27], and WC's determinism follows from that of Kahn networks.

3 High-level Synchronization

While a parallel application could in theory be written using raw, explicit WC acquire/release constructs in the form discussed above, we would never expect ordinary software developers to do so. Instead, practical use requires higher-level, more developer-friendly synchronization abstractions that support common parallel programming idioms while preserving the determinism of the underlying WC model.

As one approach to providing such high-level abstractions, we are developing *DOMP*, a variant of OpenMP [36] based on the WC model. 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. While we describe DOMP in more detail elsewhere [1], here we summarize key synchronization abstractions and describe how they map to the acquire/release "primitives" underlying the WC model.

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 WC, 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 WC 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

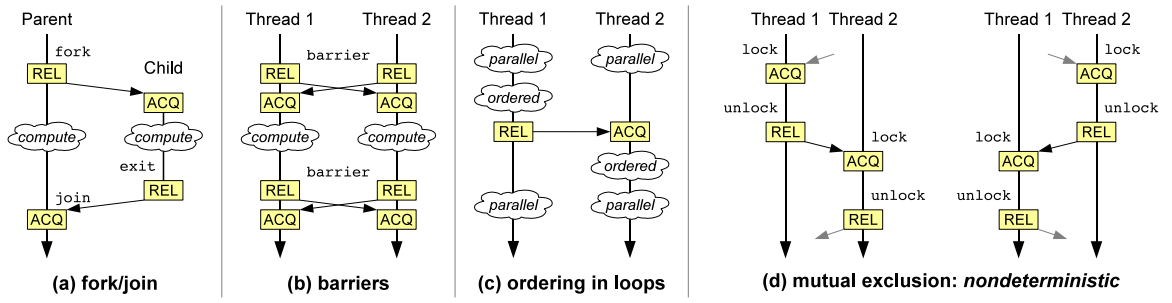

Figure 3: Mapping of High-level Synchronization Operations to Acquire/Release Pairs

`critical` sections or locks. All uses of these serialization constructs in the NAS Parallel Benchmarks [25] 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 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 WC 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 [22] or nonhierarchical dependency graphs [19] 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 WC

We have implemented two early prototype implementations of WC: one that runs in user-space atop Linux and uses `mmap`-based memory management techniques similar to Grace [8], and one in the context of the experimental Determinator OS [4]. Both of these prototypes, however, support only strictly hierarchical synchronization patterns, such as fork/join and barriers.

As one approach to implementing the generalized WC model, we are extending Determinator's virtual memory system to support a *single producer multiple consumer* (SPMC) shared memory primitive. While Determinator normally constrains interprocess communication strictly to direct parent/child relationships, the SPMC primitive allows a parent process to set up virtual memory "communication channels" for communication directly between different children, eliminating the parent as a scalability bottleneck in subsequent computation. The kernel constrains these communication channels so as to preserve a strong guarantee of determinism despite the "peer-to-peer" communication they support. These SPMC channels thus form communication primitives analogous to the Kahn process networks that WC is conceptually based on, and atop which user-level code can build shared memory parallel abstractions supporting nonhierarchical high-level synchronization.

To support SPMC, we extend the Determinator API with two optional arguments (see Table 1) taken by the existing Put/Get system calls [4].

A user process P calls Get with the Zero and Own options to get a Zero-filled SPMC virtual memory range, and becomes the owner (producer) of that memory range. P can transfer ownership of this SPMC memory to any child by invoking Put with the Own, Copy options; the parent then becomes a consumer of the mem-

Put	Get	Option	Description
✓	✓	Own	PUT/GET the ownership of an SPMC memory to/from child.
	✓	Fix	Fix the SPMC memory.

Table 1: Extended options to the Put and Get calls.

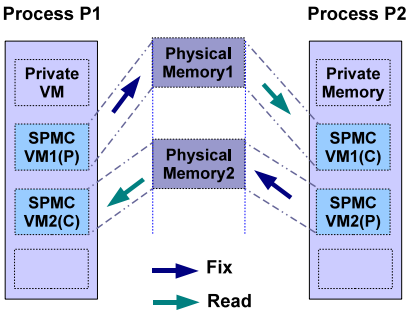

Figure 4: Processes can share several SPMC memories, each of which has only one producer process, but multiple consumer processes. The consumers can access the memory after the producer fixes it.

ory range. P can also hand out consumer mappings of the memory range to any number of child processes, by calling Put with the Copy option.

Determinator maps the virtual memory ranges of the producer and all consumers to the same physical memory, as shown in Figure 4. To ensure determinism, however, Determinator does not allow consumers to read a memory range while the producer is writing to it. Instead, the kernel gives *only* the producer access to a given SPMC memory page, until the producer *fixes* the page by calling Get with the new Fix option. Once Fixed, the producer loses the ability to write to the page, but all consumers gain the ability to read the page, and the page remains read-only for the rest of its lifetime.

If a consumer attempts to read a page before the producer fixes it, the kernel blocks the consumer, to be awoken later when the producer fixes the page. Determinator in this way allows interprocess synchronization to "short-circuit" the process hierarchy while preserving the constraints of the deterministic model.

We believe that Determinator's SPMC primitive will be usable to implement parallel applications more efficiently and scalably than the current Determinator prototype, particularly for applications demanding pipeline parallelism or "all-to-all" communication like the MapReduce model. During a MapReduce, for example, the map stage can create several SPMC memories for each divided range of the input pairs, transfer ownership of each SPMC memory to each corresponding worker, and set each worker to be a consumer of other SPMC memories. Each worker performs its task concurrently and fixes its owned SPMC memory after producing its intermediate pairs. Determinator then makes the map phase results directly available to the workers starting their reduce phases as they become ready, without requiring any coordination with a parent or master thread that might become a scalability bottleneck.

While an early version of the SPMC extension is working, we have not yet implemented the full WC model atop it or ported suitable applications, so a more thorough exploration remains for future work.

5 Related Work

WC conceptually builds on release consistency [21] and lazy release consistency [28], which relax sequential consistency's ordering constraints to increase the independence of parallel activities. WC retains these independence benefits, adding determinism by delaying the propagation of any thread's writes to other threads until *required* by explicit synchronization. This delaying of writes is reminiscent of early parallel Fortran systems [5, 38], and of recent language features such as Sieve [32] and isolation types [12].

Race detectors [20, 35] can detect certain heisenbugs, but only determinism eliminates their possibility. Language extensions can dynamically check determinism assertions in parallel code [13, 37], but heisenbugs may persist if the programmer omits an assertion. SHIM [18, 19, 40] provides a deterministic message-passing programming model, and DPJ [9, 10] 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 [6, 16] 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 [8] runs fork/join-style programs deterministically using virtual memory techniques.

Replay systems can log and reproduce particular executions of conventional nondeterministic programs, for debugging [15, 30] or intrusion analysis [17, 26]. 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, transac-

tional memory (TM) systems [23, 39] 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. Workspace consistency provides an alternative parallel programming model as simple as "parallel assignment," and supports a variety of deterministic synchronization abstractions.

Acknowledgments: This research was supported by the NSF under grant CNS-1017206.

References

[1] Amittai and B. Ford. Deterministic OpenMP, Jan. 2010. *Under submission.*

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

[3] C. Artho, K. Havelund, and A. Biere. High-level data races. In VVEIS, *pages 82–93, Apr. 2003.*

[4] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Determinator: OS support for efficient deterministic parallelism. In 9th OSDI, *Oct. 2010.*

[5] M. Beltrametti, K. Bobey, and J. R. Zorbas. The control mechanism for the Myrias parallel computer system. Computer Architecture News, *16(4):21–30, Sept. 1988.*

[6] T. Bergan et al. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In 15th ASPLOS, *Mar. 2010.*

[7] T. Bergan et al. Deterministic process groups in dOS. In 9th OSDI, *Oct. 2010.*

[8] E. D. Berger et al. Grace: Safe multithreaded programming for C/C++. In OOPSLA, *Oct. 2009.*

[9] R. L. Bocchino et al. Parallel programming must be deterministic by default. In HotPar. *Mar. 2009.*

[10] R. L. Bocchino et al. A type and effect system for deterministic parallel Java. In OOPSLA, *Oct. 2009.*

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

[12] S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In OOPSLA 2010, *pages 691–707, 2010.*

[13] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In FSE, *Aug. 2009.*

[14] H. Cui, J. Wu, and J. Yang. Stable deterministic multithreading through schedule memoization. In 9th OSDI, *Oct. 2010.*

[15] R. S. Curtis and L. D. Wittie. BugNet: A debugging system for parallel programming environments. In 3rd ICDCS, *pages 394–400, Oct. 1982.*

[16] J. Devietti et al. DMP: Deterministic shared memory multiprocessing. In 14th ASPLOS, *Mar. 2009.*

[17] G. W. Dunlap et al. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In 5th OSDI, *Dec. 2002.*

[18] S. A. Edwards and O. Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. Transactions on VLSI Systems, *14(8):854–867, Aug. 2006.*

[19] S. A. Edwards, N. Vasudevan, and O. Tardieu. Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads. In DATE, *Mar. 2008.*

[20] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In 19th SOSP, *Oct. 2003.*

[21] K. Gharachorloo et al. Memory consistency and event ordering in scalable shared-memory multiprocessors. In 17th ISCA, *pages 15–26, May 1990.*

[22] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. TOPLAS, *7(4):501–538, Oct. 1985.*

[23] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In 20th ISCA, *pages 289–300, May 1993.*

[24] 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. 1972. Reprinted in [11], 231–244.*

[25] 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, Oct. 1999.

[26] A. Joshi et al. Detecting past and present intrusions through vulnerability-specific predicates. In 20th SOSP, *pages 91–104. 2005.*

[27] G. Kahn. The semantics of a simple language for parallel programming. In Information Processing, *pages 471–475. 1974.*

[28] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In ISCA, *pages 13–21, May 1992.*

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

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

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

[32] A. Lokhmotov, A. Mycroft, and A. Richards. Delayed side-effects ease multi-core programming. In EuroPar, *Aug. 2007.*

[33] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics. In 13th ASPLOS, *pages 329–339, Mar. 2008.*

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

[35] M. Musuvathi et al. Finding and reproducing Heisenbugs in concurrent programs. In 8th OSDI. *2008.*

[36] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[37] C. Sadowski, S. N. Freund, and C. Flanagan. SingleTrack: A dynamic determinism checker for multithreaded programs. In 18th ESOP, *Mar. 2009.*

[38] J. T. Schwartz. The burroughs FMP machine, Jan. 1980. Ultracomputer Note #5.

[39] N. Shavit and D. Touitou. Software transactional memory. Distributed Computing, *10(2):99–116, Feb. 1997.*

[40] O. Tardieu and S. A. Edwards. Scheduling-independent threads and exceptions in SHIM. In EMSOFT, *pages 142–151, Oct. 2006.*

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