Toward a Formal Semantic Framework for Deterministic Parallel Programming *

Michael L. Scott Li Lu
University of Rochester
{scott, llu}@cs.rochester.edu

Abstract

Deterministic parallelism has become an increasingly attractive concept: a deterministic parallel program may be easier to construct, debug, understand, and maintain. However, there exist many different definitions of "determinism" for parallel programming. Many existing definitions have not yet been fully formalized, and the relationships among these definitions are still unclear. We argue that formalism is needed, and that history-based operational semantics—as used, for example, to define the Java and C++ memory models—provides a useful lens through which to view the notion of determinism. As a first step, we suggest several history-based definitions of determinism. We discuss some of their comparative advantages, note containment relationships among them, and identify programming idioms that support them. We also propose directions for future work.

1. Introduction

Determinism is touted as a way to simplify parallel programming—to make it easier to understand what a parallel program does. At the very least, determinism suggests that a given parallel program—like a sequential program under most semantic models—should always produce the same output when run with the same input.

We believe, however, that it needs to mean more than this—that runs of a deterministic program on a given input should not only produce the same output: they should produce it *in the same way*. By analogy to automata theory, a deterministic Turing machine doesn't just compute a single-valued function: it takes a uniquely determined action at every step along the way.

For real-world parallel programs, computing "in the same way" may be defined in many ways. Depending on context, we may expect that repeated runs of a deterministic program will consume (more or less) the same amount of time and space; that they will display the same observable intermediate states to a debugger; that the number of code paths requiring separate testing will be linear in the number of threads (rather than exponential); or that the programmer will be able to straightforwardly predict the impact of source code changes on output or on time and space consumption.

* This work was supported in part by NSF grant CCR-0963759.

History-based operational semantics has proven to be one of the most useful ways to model the behavior of parallel programs. Among other things, it has been used to explain the serializability of transactions [12], the linearizability of concurrent data structures [8], and the memory model that determines the values seen by reads in a language like Java [10] or C++ [5]. Memory models typically distinguish between *ordinary* and *synchronizing* accesses, to build a cross-thread partial order among operations of the program as a whole. Recently we have proposed that the various sorts of synchronizing accesses be unified under the single notion of an *atomic action* [6, 13].

Informally, the parallel semantics of a given program on a given input is defined to be a set of *program executions*. Each execution comprises a set of *thread histories*, each of which in turn comprises a totally ordered sequence of *reads*, *writes*, and other *operations*—notably *external actions* like input and output. The history of a given thread is determined by the program text, the language's (separately specified) sequential semantics, the input provided at run time, and the values returned by reads (which may have been set by writes in other threads). An execution is said to be *sequentially consistent* if there exists a total order on reads and writes, consistent with program order in every thread, such that each read returns the value written by the most recent preceding write to the same location. Under relaxed memory models, a program is said to be *data-race free* if the model's partial order covers all pairs of conflicting operations.

An *implementation* maps source programs to sets of low-level *target executions* on some real or virtual machine. The implementation is correct only if, for every target execution, there exists a corresponding program execution that performs the same external actions, in the same order.

In the strictest sense of the word, a deterministic parallel program would be one whose semantics, on any given input, consists of only a single program execution, to which any legal target execution would have to correspond. In practice, this definition may prove too restrictive. Suppose, for example, that I have chosen, as a programmer, to "roll my own" shared allocator for objects of some heavily used data type, and that I am willing to ignore the possibility of running out of memory. Because they access common metadata, allocation and deallocation operations must synchronize with one

another; they must be ordered in any given execution. Since I presumably don't care what the order is, I may wish to allow arbitrary executions that differ only in the order realized, while still saying that my program is deterministic.

In general, we suggest, it makes sense to say that a program is deterministic if all of its program executions on a given input are *equivalent* in some well-defined sense. A *language* may be said to be deterministic if all its programs are deterministic. Even for a nondeterministic language, an *implementation* may be deterministic if all the target executions of a given program on a given input correspond to program executions that are mutually equivalent. For all these purposes, ***the definition of determinism amounts to an equivalence relation on program executions.***

We contend that history-based semantics provides a valuable lens through which to view the notion of determinism. By specifying semantics in terms of thread histories, we capture the notion of "computing in the same way"—not just computing the same result. We also accommodate programs (e.g., servers) that are not intended to terminate. By separating semantics (source-to-program-execution) from implementation (source-to-target-execution), we fix the level of abstraction at which determinism is expected, and, with an appropriate definition of "equivalence," we codify what determinism *means* at that level.

For examples like the memory allocator mentioned above, history-based semantics highlights the importance of language definition. If my favorite memory management mechanism were a built-in facility, with no implied ordering among allocation and deallocation operations of different objects, then a program containing uses of that facility might still have a single execution. Other potential sources of nondeterminism that might be hidden inside the language definition include parallel iterators, bag-of-task work queues, and container data types (sets, bags, mappings). Whether *all* such sources can reasonably be shifted from semantics to implementation remains an open question (but we doubt it).

From an implementation perspective, history-based semantics differentiates between things that are required to be deterministic and things that an implementation might *choose* to make deterministic. This perspective draws a sharp distinction between projects like DPJ [4], which can be seen as constraining the set of program executions, and projects like DMP [7], CoreDet [2], Kendo [11] and Grace [3], which can provide deterministic execution even for pthread-ed programs in C. (Additional projects, such as Rerun and DeLorean [9], are intended to provide deterministic *replay* of a program whose initial run is more arbitrary.)

If we assume that an implementation is correct, history-based semantics identifies the set of executions that an application-level test harness might aspire to cover. For purposes of debugging, it also bounds the set of global states that might be visible at a breakpoint—namely, those that correspond to a consistent cut through the partial order of a legal program execution.

We believe the pursuit of deterministic parallel programming will benefit from careful formalization in history-based semantics. Toward that end, we present several possible definitions of equivalence for program executions in Section 2. We discuss their comparative advantages in Section 3. We also note containment relationships among them, and identify programming idioms that ensure them. Many other definitions of equivalence are possible, and many additional questions seem worth pursuing in future work; we list a few of these in Section 4.

2. Example Definitions of Equivalence

For the purposes of this position paper, we define an *execution* to be a 3-tuple $E : (OP, <_p, <_s)$, where *OP* is the set of operations, $<_p$ is the *program order*, and $<_s$ is the *synchronization order*. An operation in *OP* can be written as (op, val*, tid), where op is the operation name, val* is a sequence of involved values, and tid is the ID of the executing thread. An operation may read or write a variable, perform input or output, or invoke an atomic operation (e.g., enqueue or dequeue) on some built-in data type. Certain objects are identified as *synchronization variables*; operations on them are *synchronization operations*. We use $OP|_s$ to represent the synchronization operations of *OP*.

Program order is a union of per-thread total orders. order is a partial order, consistent with program order, on synchronization operations. *Happens-before order*, $<_{hb}$, is the irreflexive transitive closure of $<_p$ and $<_s$. A read r is allowed to "see" a write w iff w is the most recent previous write to the same variable on some happens-before path, or, in some languages, if r and w are incomparable under $<_{hb}$. Two operations *conflict* if they access the same variable and at least one of them writes it. An execution is *data-race free* if all conflicting operations are ordered by $<_{hb}$.

Input and output operations are known as *external events*. For simplicity, we assume that every external event is a synchronization operation, and that these are totally ordered by $<_s$. We use $ext(E)$ to represent the sequence of external operations in E, without their thread ids. We also model input and output as vector variables: an input operation is a read of the next element of the input vector (and a write of the variable into which that element is input); an output operation is a write of the next element of the output vector (and a read of the variable from which that element is output).

To accommodate nonterminating programs, we allow input, output, *OP*, $<_p$, and $<_s$ to be unbounded. Rather than model fork and join, we assume the availability, throughout execution, of an arbitrary number of threads.

Singleton. *Executions* $E_1 : (OP_1, <_{p1}, <_{s1})$ *and* $E_2 : (OP_2, <_{p2}, <_{s2})$ *are said to be equivalent if and only if* $OP_1 = OP_2$, $<_{p1}=<_{p2}$, *and* $<_{s1}=<_{s2}$.

Singleton uses the strictest possible definition of determinism: there must be only one possible execution for a given program and input.

Dataflow. *Executions* $E_1 : (OP_1, <_{p1}, <_{s1})$ *and* $E_2 : (OP_2, <_{p2}, <_{s2})$ *are said to be equivalent if and only if there is a one-one mapping (bijection) between* OP_1 *and* OP_2 *that preserves (1) the content other than thread id in every operation and (2) the reads-see-writes relationship induced by* $<_p$ *and* $<_s$.

As its name suggests, *Dataflow* defines E_1 and E_2 to be equivalent if and only if they see the same flow of values among their operations. Because we model input and output as vector reads and writes, this implies that the executions have the same external behavior. Note that we do not require either $<_{p1}=<_{p2}$ or $<_{s1}=<_{s2}$, nor do we require that the executions be data-race free.

SyncOrder. *Executions* $E_1 : (OP_1, <_{p1}, <_{s1})$ *and* $E_2 : (OP_2, <_{p2}, <_{s2})$ *are said to be equivalent if and only if* $OP_1|_s = OP_2|_s$ *and* $<_{s1}=<_{s2}$.

SyncOrder requires only that there be a fixed pattern of synchronization among threads (with all synchronization events occurring in the same threads, with the same values, in both E_1 and E_2). Note, however, that if executions are data-race free (something that *SyncOrder* does not require), then they are also sequentially consistent [1], so $E_1 \equiv_{\text{SyncOrder}} E_2 \ \wedge \ E_1, E_2 \in DRF \longrightarrow E_1 \equiv_{\text{Dataflow}} E_2$.

ExternalEvents. *Executions* $E_1 : (OP_1, <_{p1}, <_{s1})$ *and* $E_2 : (OP_2, <_{p2}, <_{s2})$ *are said to be equivalent if and only if* $ext(E_1) = ext(E_2)$

ExternalEvents is the most widely accepted language-level definition of determinism. It guarantees that program executions look "the same" from the outside world on multiple executions with the same input.

FinalState. *Executions* $E_1 : (OP_1, <_{p1}, <_{s1})$ *and* $E_2 : (OP_2, <_{p2}, <_{s2})$ *are said to be equivalent if and only if (1) they have the same sets of variables, (2) they both terminate, and (3) for every variable (including the output vector), each final write (each write for which there is no later write to the same variable under* $<_{hb}$*) assigns the same value.*

Like *ExternalEvents*, *FinalState* says nothing about how E_1 and E_2 compute. It requires only that final values be the same. Unlike *ExternalEvents*, *FinalState* requires agreement on variables *other* than output, and it focuses on the values themselves, rather than the operations that produce them.

3. Discussion

In this section, we discuss the comparative advantages of the definitions in Section 2. With all five definitions, equivalent program executions will have the same set of external events (this is Theorem 2 in Section 3.2). This ensures that the output of a deterministic program will never depend on the program execution to which a given target execution corresponds—in particular, it will never depend on scheduling choices outside the programmer's control.

In the first subsection below, we consider several other potential goals of determinism—specifically:

- Whether a definition of equivalence can guarantee deterministic program state at any debugger breakpoint (to enable repetitive debugging). We assume a symbolic debugger that does not differentiate among target executions that correspond to the same program execution.
- Whether a definition applies to nonterminating programs.
- How hard it is likely to be to implement and verify a programming model that ensures the definition; how much run-time cost it is likely to impose.

In Section 3.2 we consider containment relationships among our equivalence relations. In Section 3.3 we consider programming languages and idioms that guarantee various forms of equivalence among their program executions.

3.1 Comparative Advantages

Singleton is the strictest definition of equivalence, and thus of determinism. It requires a single execution for any given source program and input. At any program breakpoint in a *Singleton* system, a debugger that works at the level of program executions will be guaranteed to see a state that corresponds to some consistent cut across the happens-before order of the single execution. This guarantee facilitates repetitive debugging, though it may not make it trivial: a breakpoint in one thread may participate in an arbitrary number of cross-thread consistent cuts; global state is not uniquely determined by the state of a single thread. If we allow all other threads to continue running, however, until they wait for a stopped thread or hit a breakpoint of their own, then global state will be deterministic. Moreover, since *Singleton* requires runs of a program to correspond to the same program execution, monitored variables' values will change deterministically. This should simplify both debugging and program understanding.

Singleton allows nonterminating program executions to be equivalent, because equality is well defined even on unbounded operation sets and partial orders. It allows us to talk about determinism even for programs like servers, which are intended to run indefinitely. Interestingly, while we have not insisted that *Singleton* executions be sequentially consistent, they seem likely to be so in practice: a language that admits non-sequentially consistent executions (e.g., via data races) seems likely to admit multiple executions for some ⟨program, input⟩ pairs.

On the down side, *Singleton* requires target executions of a given program on a given input to be identical in every program-execution-level detail. This may be straightforward for certain restrictive programming idioms (e.g., *independent split-merge*, which we discuss in Section 3.3), but for

more general programs, a conforming, scalable implementation seems likely to require either special-purpose hardware or very high run-time overhead. *Singleton* also has the disadvantage of ruling out "benign" differences of any kind among program executions. It is likely to preclude a variety of language features and programming idioms that users might still like to think of as "deterministic" (examples appear below and in Section 3.3).

Dataflow relaxes *Singleton* by loosening the requirements on control flow. Equivalent executions must still have the same operation sets (except for thread ids), but the synchronization and program orders can be different, so long as values flow from the same writes to the same reads. Intuitively, *Dataflow* can be thought of as an attempt to accommodate programming models in which the work of the program is fixed from run to run, but may be partitioned and allocated differently among the program's threads. Monitored variables in a debugger will still change values deterministically, but two executions may not reach the same global state when a breakpoint is triggered, even if threads are allowed to "coast to a stop." A program state encountered in one execution may never arise in an equivalent execution.

Consider the code fragment shown in Figure 1, written in a hypothetical language. Assume that f() is known to be a pure function, and that the code fragment is embedded in a program that creates two worker threads for the purpose of executing parallel iterators. In one plausible semantics, the elements of a parallel iteration space are placed in a synchronous queue, from which workers dequeue them atomically. Even in this trivial example, there are four possible executions, in which dequeue operations in threads 0 and 1, respectively, return $\{0, \perp\}$ and $\{1, \perp\}$, $\{1, \perp\}$ and $\{0, \perp\}$, $\{0, 1, \perp\}$ and $\{\perp\}$, or $\{\perp\}$ and $\{0, 1, \perp\}$. These executions will contain exactly the same operations, except for thread ids. They will have different program and synchronization orders. *Dataflow* will say they are equivalent; *Singleton* will say they are not. If we insist that our programming model be deterministic, *Dataflow* will clearly afford the programmer significantly greater expressive power. On the other hand, a breakpoint inserted at the call to f() in thread 0 may see very different global states in different executions; this could cause significant confusion.

Like *Singleton*, *Dataflow* accommodates nonterminating executions. It can be guaranteed with minimal cost by a somewhat larger class of restricted programming models (e.g., *bag of independent tasks*; see Section 3.3). Scalable performance for more general programs again seems problematic in the absence of special-purpose hardware.

SyncOrder also relaxes *Singleton*, but by admitting benign changes in data flow, rather than control flow. Specifically, *SyncOrder* requires equivalent executions to contain the exact same synchronization operations, executed by the same threads in the same order. It does *not* require that a read see the same write in both executions, but it does require

```
parfor i in [0, 1]
  A[i] = f(i)
seqfor i in [0, 1]
  print A[i]
```

Figure 1. A program fragment amenable to self-scheduling.

that any disagreement have no effect on synchronization order (including output). Like *Dataflow*, *SyncOrder* fails to guarantee deterministic global state at breakpoints, but we hypothesize that the variability will be significantly milder in practice: benign data flow changes, which do not impact synchronization or program output, seem much less potentially disruptive than benign synchronization races, which can change the allocation of work among threads.

Because $<_s$ can be unbounded, *SyncOrder*, like *Singleton* and *Dataflow*, accommodates nonterminating programs. Because it has fewer ordering restrictions than *Singleton*, *SyncOrder* is likely to be cheaper to implement with scalable performance for a reasonably broad class of programs. It may also be cheaper and more straightforward than *Dataflow*, because an implementation may be able to limit instrumentation to synchronization events, rather than arbitrary reads and writes.

ExternalEvents can be seen as a looser version of both *Dataflow* and *SyncOrder*. It requires agreement only on external actions; the set of operations, their distribution among threads, and their order can all vary among equivalent executions. Because input and output can be unbounded, *ExternalEvents* accommodates nonterminating executions. Because its internal computation is entirely unconstrained, however, it seems much less appropriate for repetitive debugging. Implementation cost is hard to predict: it will depend on the programming model used to guarantee deterministic output. While it is easy to look at two executions and decide if their output is the same, it may be much harder to ensure that the executions permitted by a given programming model will always include the same external events for a given program and input. In the fully general case, of course, the output-equivalence of programs is undecidable.

The appeal of *ExternalEvents* lies in its generality. If output is all one cares about, it affords the language designer and implementor maximum flexibility. Knowing that a parallel program will always generate the same output from the same input, regardless of scheduling idiosyncrasies, is a major step forward from the status quo. For users with a strong interest in predictable performance and resource usage, debugability, and maintainability, however, *ExternalEvents* may not be enough.

FinalState is essentially a variant of *ExternalEvents* restricted to programs that terminate. Like *ExternalEvents*, it offers significant flexibility to the language designer and implementor, but seems problematic for repetitive debugging, and difficult to assess from an implementation perspective.

3.2 Containment Properties

Figure 2 posits containment relationships among the definitions of determinism given in Section 2. The space as a whole is populated by sets $\{X_i\}$ of executions of some given program on a given input, with some given semantics. If region S is contained in region L, then every set of executions that are equivalent under definition S are equivalent under definition L as well; that is, S is a stricter and L a looser definition. (The regions can also be thought of as containing languages or, analogously, executions: a language [execution] is in region R if for every program and input, all program executions generated by the language semantics [or corresponding to target executions generated by the implementation] are equivalent under definition R.) We informally justify the illustrated relationships as follows.

Theorem 1. Singleton *is contained in* Dataflow, SyncOrder, *and* ExternalEvents.

Rationale: Clearly every execution has the same dataflow, synchronization order, and external events as itself. If it terminates, it has the same final state as itself: *Singleton* and *FinalState* have a nontrivial intersection.

Theorem 2. Singleton, Dataflow, SyncOrder, *and* FinalState *are all contained in* ExternalEvents.

Rationale: This is trivial for *Singleton*, and true by definition for the others: External events are considered to be both reads and writes (so *Dataflow* preserves them) and synchronization operations (so *SyncOrder* preserves them). And since input is modeled as the initial values of a read-only input vector, and output as the final values of a distinguished output vector, executions with the same final state have the same external events.

Theorem 3. *There are sets of executions that are equivalent under* Dataflow *but not under* SyncOrder.

Rationale: This is the light gray region, labeled "1" in Figure 2. It corresponds to programs containing benign synchronization races. Suppose, for example, that threads t_1 and t_2 compete to update a flag, under the protection of a lock. Whichever thread gets there first assumes responsibility for executing function F. The two resulting executions have isomorphic data flow (all that changes is the thread ids in the corresponding reads and writes), but a different synchronization order. (Further discussion of this region appears under "Bag of Independent Tasks" in Section 3.3.)

Theorem 4. *There are sets of executions that are equivalent under* SyncOrder *but not under* Dataflow.

Rationale: This is the medium gray region, labeled "2" in Figure 2. It corresponds to programs containing benign data races. Suppose, for example, that variable x is initially 0 and that threads t_1 and t_2 execute concurrent functions F_1 and F_2 that access only one variable in common: specifically, F_1 sets x to 1 and F_2 sets x to 2. If some other thread subsequently executes `if (x != 0) ...`, then we will have executions with different data flow but the same synchronization order (and the same external events).

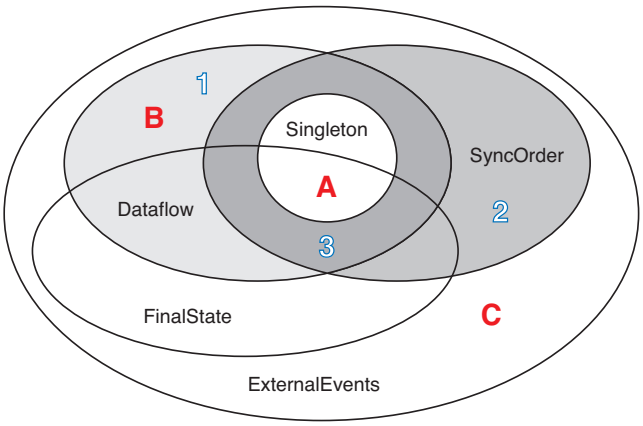

Figure 2. Containment relationships among definitions of determinism, or, equivalently, program execution equivalence. Names of equivalence definitions correspond to ovals. Outlined numbers label the light, medium, and dark shaded regions. Bold letters show the locations of programming idioms discussed in Section 3.3.

Theorem 5. Singleton, Dataflow, *and* SyncOrder *all have nontrivial intersections with* FinalState.

Rationale: *Singleton*, *Dataflow*, and *SyncOrder* all contain sets of nonterminating executions, which cannot be equivalent according to *FinalState*. At the same time, *Singleton*, *Dataflow*, and *SyncOrder* all contain sets of terminating executions that have the same final state.

3.3 Programming Languages and Idioms

While equivalence relations and their relationships, seen from a theoretical perspective, may be interesting in their own right, they probably need to correspond to some intuitively appealing programming language or idiom in order to be of practical interest. As illustrations, we describe programming idioms whose sets of executions would appear in the regions labeled "A," "B," and "C" in Figure 2.

Independent Split-Merge (Region "A"— *Singleton* in Figure 2.) Consider a language providing parallel iterators or cobegin, with the requirement (enforced through the type system or run-time checks) that concurrent tasks access disjoint sets of variables. If every task is modeled as a separate thread, then there will be no synchronization or data races, and the execution of a given program on a given input will be uniquely determined.

Bag of Independent Tasks (Region "B"— *Dataflow* $\setminus$ *SyncOrder* in Figure 2.) Consider a programming idiom in which a fixed set of threads dynamically self-schedule independent tasks from a shared bag. The resulting executions

will have isomorphic data flow (all that will vary is the thread ids in the corresponding reads and writes), but their synchronization orders will vary with the order in which they access the bag of tasks.

One might expect that a program with deterministic sequential semantics, no data races, and no synchronization races would have only a single program execution—that is, that *Dataflow* ∩ *SyncOrder* = *Singleton*. We speculate, however, that there may be cases—e.g., uses of rand()—that are easiest to model with more than one execution (i.e., with classically nondeterministic sequential semantics), but that we might still wish to think of as "deterministic parallel programming." We have left a region in Figure 2 (the dark gray area labeled "3") to suggest this possibility.

Parallel Iterator with Reduction (Region "C"— *External-Events* ⧵ (*Dataflow* ∪ *SyncOrder*) in Figure 2.) Consider a language with explicit support for reduction by commutative functions. The order of updates is not fixed, leading to executions with different synchronization orders and data flow, but only a single result. It seems plausible that we might wish to call programs in such a language "deterministic."

4. Conclusions and Future Work

Deterministic parallel programming needs a formal definition (or set of definitions). Without this, we really have no way to tell whether the implementation of a deterministic language is correct. History-based operational semantics seems like an excellent framework in which to create definitions, for all the reasons mentioned in Section 1. We see a wide range of topics for future research:

- Everything in this position paper needs to be formalized much more rigorously. Theorems (e.g., the containment properties of Section 3.2) need to be carefully stated and then proven.
- Existing projects need to be placed within the framework. What are their definitions of execution equivalence?
- Additional definitions need to be considered, evaluated, and connected to the languages and programming idioms that might ensure them.
- We need to accommodate condition synchronization, and spinning in particular. Even in *Singleton*, executions that differ only in the number of times a thread checks a condition before finding it to be true should almost certainly be considered to be equivalent.
- We need to decide how to handle operations (e.g., rand()) that compromise the determinism of sequential semantics. Should these in fact be violations? Should they be considered inputs? Should they perhaps be permitted only if they do not alter output?
- Languages embodying the more attractive definitions of determinism should be carefully implemented, and their relative costs assessed.
- For the most part, we have concerned ourselves in this position paper with deterministic *semantics*, which map a source program and its input to a set of program executions that are mutually equivalent. We also need to consider deterministic *implementations*, which map a source program and its input to a set of *target* executions whose corresponding program executions are mutually equivalent—even when the semantics includes other, non-equivalent program executions.

References

[1] S. V. Adve and M. D. Hill. Weak Ordering—A New Definition. *17th Intl. Symp. on Comp. Arch.*, May 1990.

[2] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. *15th Intl. Conf. on Arch. Support for Prog. Lang. and Op. Systems*, Mar. 2010.

[3] E. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. *OOPSLA Conf. Proc.*, Oct. 2009.

[4] R. L. Bocchino Jr., V. S. 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. *OOPSLA Conf. Proc.*, Oct. 2009.

[5] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, June 2008.

[6] L. Dalessandro, M. L. Scott, and M. F. Spear. Transactions as the Foundation of a Memory Consistency Model. *24th Intl. Symp. on Dist. Comp.*, Sept. 2010.

[7] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. *14th Intl. Conf. on Arch. Support for Prog. Lang. and Op. Systems*, Mar. 2009.

[8] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Prog. Lang. and Systems*, 12(3):463-492, July 1990.

[9] D. R. Hower, P. Montesinos, L. Ceze, M. D. Hill, and J. Torrellas. Two Hardware-Based Approaches for Deterministic Multiprocessor Replay. *Comm. of the ACM*, 52(6):93-100, June 2009.

[10] J. Manson, W. Pugh, and S. Adve. The Java Memory Model. *32nd ACM Symp. on Principles of Prog. Lang.*, Jan. 2005.

[11] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. *14th Intl. Conf. on Arch. Support for Prog. Lang. and Op. Systems*, Mar. 2009.

[12] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *J. of the ACM*, 26(4):631-653, Oct. 1979.

[13] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. Ordering-Based Semantics for Software Transactional Memory. *12th Intl. Conf. on Principles of Dist. Sys.*, Dec. 2008.