The Case For Merging Execution- and Language-level Determinism with MELD

Joseph Devietti Dan Grossman Luis Ceze
University of Washington
{devietti, djg, luisceze}@cs.washington.edu

Abstract

Nondeterminism is a key contributor to the difficulty of parallel programming. Many research projects have shown how to provide deterministic parallelism, but with unfortunate trade-offs. *Deterministic execution* enforces determinism for arbitrary programs but with significant runtime cost, while *deterministic languages* enforce determinism statically (without runtime overhead) but only for fork-join programs expressible in their static type systems.

MELD unifies these approaches. We explain the requirements for soundly integrating a deterministic language into a deterministic execution system, and describe a simple qualifier-based type checker that ensures isolation for code written in a deterministic language. We also extend MELD to incorporate nondeterministic operations without compromising the determinism of the rest of the program. Our experiments with benchmarks from the SPLASH2 and PARSEC suites show that a small number of annotations can accelerate the performance of deterministic versions of these programs by 2-6x.

1. Introduction

Nondeterminism is one of the main reasons parallel programming is so complicated. Non-reproducible "heisenbugs" plague both developers and users of parallel programs, undermining the quality of parallel software. There is a clear need to simplify the practice of parallel programming so that programmers can take advantage of increasingly ubiquitous parallel computing resources.

Deterministic parallelism is a promising approach to simplifying the use of parallelism. **Execution-level techniques** [1–8] enforce a deterministic, but still parallel, interleaving of memory operations at runtime. Some techniques work for only restricted classes of programs such as data-race-free [2] or fork-join [4] programs, while others work for arbitrary multithreaded code. All of these approaches impose some runtime overhead in exchange for determinism. **Language-level techniques** [9–13] eschew runtime overheads by adopting a more restrictive programming model, such as pipelines [9] or fork-join [12]. For code that fits into such a paradigm, determinism can be enforced by construction or via a static type system, which results in no runtime overhead. MELD proposes a new middle ground: execution-level determinism by default – to support arbitrary existing code – with a targeted application of deterministic language mechanisms to make the common case fast. We believe MELD is the first system to incorporate a deterministic language within a deterministic execution system.

1.1 Overview

MELD incorporates a lightweight data-centric qualifier system for C that allows a program's data to be partitioned at fine-grain between static and dynamic determinism enforcement schemes. Initially, one might think that it is straightforward to make a function call from code managed by a deterministic execution system into code managed by a deterministic language. However, naively performing such a call can actually *break determinism* because deterministic languages make assumptions about aliasing and concurrency that do not hold if threads can make arbitrary simultaneous calls with arbitrary data into code written in a deterministic language (see Section 2 for an example).

To evaluate the MELD system, we built a prototype compiler that augments programs with a deterministic execution runtime system, and allows the integration of code written in a deterministic language. We build upon the CoreDet deterministic compiler [3] because it is open-source and provides execution-level determinism for arbitrary parallel C programs. We also leverage Deterministic Parallel Java [12] as the deterministic language because it is also open-source and provides statically-enforced determinism for an imperative language. The MELD prototype is not a wholly integrated system – implementing a deterministic parallel language on top of C/C++ would be a major undertaking in its own right. We have, however, validated our system design by implementing our qualifier type system for C and using the isolation it guarantees to extract portions of our benchmarks that are amenable to static determinism. We then translated these code portions into DPJ to verify they could be expressed in a deterministic language. MELD is general, however, and can be used to integrate other deterministic languages into alternative general-purpose deterministic execution schemes.

The value of the MELD system is shown in CoreDet and DPJ's **complementary** strengths. CoreDet performs an alias analysis that is able to remove runtime instrumentation from simple uses of thread-private data, but DPJ's programmer-driven effect system is much more powerful. DPJ's parallelism constructs are limited to fork and join, while CoreDet supports all pthread synchronization. MELD combines these systems to form a deterministic system that is faster and more general than either alone.

Not all weaknesses can be complemented away, however. To boost program throughput, CoreDet's quantum formation (but not store buffering) is enabled for all code, even code written in a deterministic language. Moreover, due to determinism's non-composable nature, data managed by CoreDet and then passed to a DPJ function cannot recover DPJ's sequential semantics, though the data can soundly be computed upon without store buffer instrumentation while preserving determinism. The practical ramification for MELD is that while the entire program is guaranteed to be deterministic, it carries the guarantees provided by an execution-level determinism system, which is weaker than that provided by a language-level approach. A parallel program written entirely in a deterministic language has sequential semantics, ensuring that differing numbers of threads at runtime cannot affect the outcome of the program – the parallelism is implicit and invisible. For execution-level techniques, however, thread count is part of program input that must be explicitly tested – running a program with a different number of threads may expose new interleavings and new program behaviors, albeit in a deterministic manner.

We show that MELD is straightforward to use by adding annotations to several parallel benchmarks from the SPLASH2 [14] and PARSEC [15] suites. Most of these benchmarks are not candidates for use in deterministic languages due to their sophisticated use of pointers and synchronization. However, we found that each of these benchmarks contains one or more computationally-intensive kernels that are amenable to deterministic language techniques. We found that a small number of annotations can eliminate a large amount of runtime instrumentation, speeding up programs substantially.

2. Combining Execution- and Language-Level Determinism

Our initial model for integrating deterministic languages is based on functions, *e.g.*, to verify that an array sort function (and any helpers it calls) is deterministically parallel. A deterministic language component (DLC) has a single entry point via a function call, and can only call other code written in a deterministic language. A DLC may be internally parallel, *e.g.*, spawning threads to help with the sort, but these threads cannot outlive the scope of the entry point call. For simplicity, we discuss integrating a single DLC within a program, but the generalization to multiple DLCs is straightforward. Each DLC is verified separately, and can be compiled without much of the runtime instrumentation (see Section 3) that is required to enforce execution-level determinism. Placing performance-critical code into DLCs is MELD's primary performance optimization.

This MELD model brings to light the implicit preconditions of deterministic languages: that the language mechanism has full visibility into the threading and aliasing of the data it operates upon. This is naturally fulfilled when an entire program is written in a deterministic language, but in our modular use, we violate this "closed world" precondition. A deterministic language statically guarantees non-interference among the threads it creates and memory it allocates but it cannot provide any guarantees about other threads or memory in the system beyond its control.

For determinism, we require non-interference between:

1. the threads internal to the DLC
2. concurrent calls to a DLC by different threads
3. a DLC and a thread running outside the DLC
4. two threads running outside the DLC

Condition 1) is handled by the deterministic language mechanism, and 4) is handled by the execution-level determinism mechanism. Conditions 2) and 3) are the interesting ones. The crucial issues for proving determinism are understanding memory aliasing and what other code is running concurrently. MELD relies on the constraints of a type qualifier system to constrain aliasing (Sections 2.1) and targets simple barrier-based parallelism that is expressible via a deterministic language (Section 2.2). While only parts of programs are amenable to this approach, we find that this is often enough to substantially improve performance.

2.1 Type Qualifiers for Isolation

To ensure isolation of a DLC, we use type qualifiers to track whether a declaration is used exclusively by a DLC, in which case it is tagged as langdet, and otherwise it is tagged with the exdet qualifier. To ease the annotation burden, declarations are exdet by default.

The MELD type system ensures program data is partitioned into exdet and langdet parts. This partition is data-centric and fine-grained: each declaration in the program can have a different qualifier. Our typing rules prohibit pointers from langdet data to exdet data, and vice versa.

This static partition of a program's data allows a DLC to reference exdet data in a non-interfering manner (langdet data is only accessible within a DLC and thus non-interference is provided by the deterministic language). The MELD compiler inserts instrumentation to enforce non-interference whenever a DLC references exdet data. This access requires the same instrumentation that is used to enforce execution-level determinism in the non-DLC part of the program. If used naively, this mechanism risks adding a large number of instrumented accesses to a DLC, which is, of course, precisely what DLCs are designed to alleviate. To avoid this pitfall, a DLC can make a local, langdet copy of exdet data and operate upon the copy, whose aliasing properties can then be fully understood. By having this copy occur inside a deterministic language, while continuing to track qualifiers on external exdet data, we can guarantee determinism without resorting to trusting programmers.

2.2 Supported Parallelism Patterns

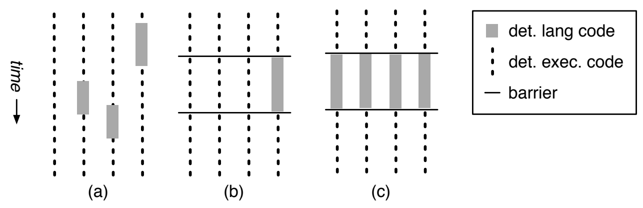

Figure 1. The deterministic language parallelism patterns supported by MELD: (a) isolated function, (b) concurrent summary and (c) data parallel.

Making copies can be prohibitively expensive, however, so a better option is often to give the deterministic language a (conservative) model of the code that runs concurrently with a DLC. Armed with such a model, a deterministic language can reason soundly about the non-interference of the DLC.

While we currently construct these models by hand, our requirements are modest enough that this could likely be done automatically with a combination of static analysis and lightweight runtime checks. Pointer aliasing in C is notoriously complicated, *e.g.*, due to pointers into the middle of arrays. The MELD qualifier system helps constrain the scope of aliasing, ensuring that all potential aliases for a given langdet location will be marked langdet themselves. This makes it easier for humans and computers to reason about aliasing when constructing models.

We utilized three different models in MELD, based on three different patterns of parallelism (Figure 1). The simplest pattern is the isolated function pattern: a function that operates on private data (perhaps by privatizing it first as explained above). An isolated function makes no assumptions about what other code is running concurrently in the system, but is also quite limited in expressivity. The second pattern is the concurrent summary pattern: we form a model of all the code that can run concurrently with a DLC and prove that it and the DLC are non-interfering. Because our target deterministic language of DPJ does not support sophisticated synchronization like locks, we cannot express complicated happens-before constraints. Data that relies on such synchronization for non-interference must remain exdet. Barrier-delimited regions of code (as in Figure 1b) are helpful, but not necessary, for determining what code can run concurrently. Our final pattern is the data parallel pattern, wherein all threads in the program perform the same computation, typically delimited by global barriers (while our prototype trusts the programmer to verify the barrier actually involves all threads, this could also be done via a simple run-time check). With this pattern, all of the parallelism can be expressed inside the DLC. We choose patterns for each workload based on the complexity of the code and what is expressible in DPJ.

2.3 Case Study: streamcluster

To motivate the MELD mechanism, we describe its application to the streamcluster benchmark from PARSEC, an online data-mining algorithm that finds a fixed number of median points over an input stream of n-dimensional points. The algorithm adjusts medians to try to minimize the sum-of-squared distances between all points and their nearest median. The most computationally intensive part of the algorithm is a repeated distance calculation between points and a current or prospective median point. Verifying the entirety of streamcluster is impossible with current-generation deterministic languages, as streamcluster generates an initial solution via an approximation pass that chooses random medians and uses condition variables to signal other threads when they need to recompute distances. However, there are several smaller portions of the workload that are amenable to deterministic language constructs.

Each point in the input stream is represented by a struct:

```
typedef struct {
  float weight, cost;
  float *coord;
  long assign;
} Point;
```

The distance calculation involves a pairwise subtraction of two Point s' coord arrays:

```
float dist( Point p1, Point p2, int dim) {
  float result=0.0;
  for (int i=0;i<dim;i++)
    result += (p1.coord[i] − p2.coord[i])*(p1.coord[i] − p2.coord[i]);
  return result;
}
```

streamcluster performs many distance calculations in parallel when evaluating potential medians.

The simplest way to use a deterministic language in streamcluster would be to employ the isolated function pattern and rewrite the dist function in such a language. The main concern is the aliasing of the coord array. Looking at dist alone does not reveal whether other threads may be performing concurrent conflicting updates to coord. Thus, to guarantee determinism, each read of coord (which is the real work of the function) would be instrumented. The MELD type system automatically inserts this instrumentation since the Point struct (and all its fields) are labeled, by default, as exdet. We could try to amortize the cost of this instrumentation by making a copy of coord before computing the distance, but there are not enough uses of each coord element to justify this.

Thus, to avoid excessive instrumentation inside dist, our deterministic language needs more information. Some of dist's call sites occur from inside small pieces of code delimited by global barriers, allowing us to use the data parallel pattern. The code that runs between these barriers is a straightforward partitioned-array parallel computation and can be expressed in DPJ using partitioned (blocked) arrays.

Across our workloads, we found many performance-critical kernels expressible using one of our DLC patterns. The extra reasoning enabled by global barriers for the data parallel pattern was also important in many benchmarks.

3. Implementation

We implemented our qualifier-based type system via gcc attributes that can be attached to any declaration. We modified the Clang front-end to the LLVM compiler toolchain to perform type checking over qualifiers, and to annotate all AST nodes with these qualifiers. Next, a modified version of the CoreDet compiler 1) instruments all accesses to potentially-shared memory to use the store buffer and 2) instruments control flow edges to count instructions for quantum formation. Store buffer instrumentation is elided for accesses to data that are tagged as langdet or nondet. Since every store buffer access involves a hash table lookup plus extracting/updating the desired bytes (and an additional access/-copy to global memory for locations not yet buffered), eliding this instrumentation is a substantial performance win.

4. Incorporating Nondeterminism

We may wish to allow a certain amount of nondeterminism within our program, *e.g.*, logging, network output or profiling code, for two reasons: 1) its nondeterminism will not have a large bearing on the determinism of the rest of the program and 2) such nondeterministic code can run without the overheads of the execution-level determinism system. To employ nondeterminism in a sound way, we need to formally guarantee that its effects are not allowed to "contaminate" the determinism of the rest of our program.

To accomplish this we employ a standard static information-flow tracking type system. The main extensions are 1) additional restrictions placed on scalar assignments, and 2) protection against implicit flows via control flow. These additions suffice to prevent the nondeterministic part of the program from affecting the other, deterministic parts.

The nondeterministic part of the program has no special requirements. To return to our list of correctness conditions from Section 2, we must additionally ensure non-interference between:

5. nondeterministic code and threads running inside a DLC
6. nondeterministic code and threads running outside a DLC

There is, of course, no requirement to isolate nondeterministic code from itself. It is sufficient to extend the type lattice to langdet $\sqsubseteq$ exdet $\sqsubseteq$ nondet and run standard information-flow typing rules.

Casts that modify the qualifiers of nondet data (*endorsements* in the information-flow tracking literature) have special semantics: they represent a kind of "internal input" to the deterministic part of a program, analogous to external input read from, *e.g.*, files or sockets. A record-and-replay system building upon MELD would perform logging at nondet endorsements to precisely capture this internal input and allow for repeatability of the deterministic portion of the program.

MELD's data-centric annotation approach is especially significant when nondeterminism is allowed into a program, because determinism guarantees are only meaningful when referring to data. An integer i will have deterministic contents at the end of a program iff it is only ever updated with deterministic values.[1] A code-centric approach to determinism like [13] allows deterministic and nondeterministic values to be assigned to i, albeit only during distinct deterministic and nondeterministic phases, respectively, of a program's execution. Such an approach cannot, however, make any guarantees about i's value being deterministic. Different levels of determinism are not composable: passing a variable with a nondeterministic value to a function written in a deterministic language cannot "recover" determinism.

5. Evaluation

5.1 Experimental Setup

We ran our experiments on an 8-core 2.4GHz Intel Xeon E5462 ("Nehalem") with 10GB of RAM, using 64-bit Ubuntu Linux 8.10. We present results that are the average of 5 runs. We used C benchmarks from the SPLASH2 and PARSEC suites. Details about each benchmark are given in Table 1, including the input problem size and optimal CoreDet quantum size. barnes, lu and radix are from SPLASH2, and blackscholes and streamcluster are from PARSEC 2.0. We scaled the problem sizes for SPLASH2 workloads so that the program runs for about a minute on our test machine with 8 nondeterministic threads. For PARSEC workloads we use the standard native input set. All benchmarks were run with 64-byte chunks for the store buffer, all of CoreDet's compiler optimizations enabled, and the memory consistency optimizations from [5]. The streamcluster benchmark was originally written in C++, but as it used very few C++ idioms it was straightforward to port it to C for compatibility with MELD.

5.2 Performance

Figure 2 shows the runtime overhead of MELD and CoreDet normalized to nondeterministic execution, with an equal number of threads (lower is better, and 0% means running

[1] Modulo the special case of resetting the variable to a known deterministic value. This is useful in a security context for regaining trust from untrusted values [16], but is not useful in our context since the nondeterministic value cannot be read.

benchmark	description	input	quantum size (insns)	LOC	MELD annots/casts	cloned LOC	parallelism pattern (2.2)	locks	conds	barriers
barnes	n-body simulation	4M bodies	200k	2964	6/7	0	conc. sum.	6	0	6
blackscholes	options pricing	parsec-native	500k	420	8/0	0	data par.	0	0	0
lu	LU factorization	7000×7000 matrix	750k	993	10/3	8	data par.	1	0	6
radix	radix sort	2^{29} integers	400k	878	2/2	0	data par.	1	2	7
streamcluster	online clustering	parsec-native	200k	2347	3/1	8	data par.	3	1	19

Table 1. Benchmarks and configurations used.

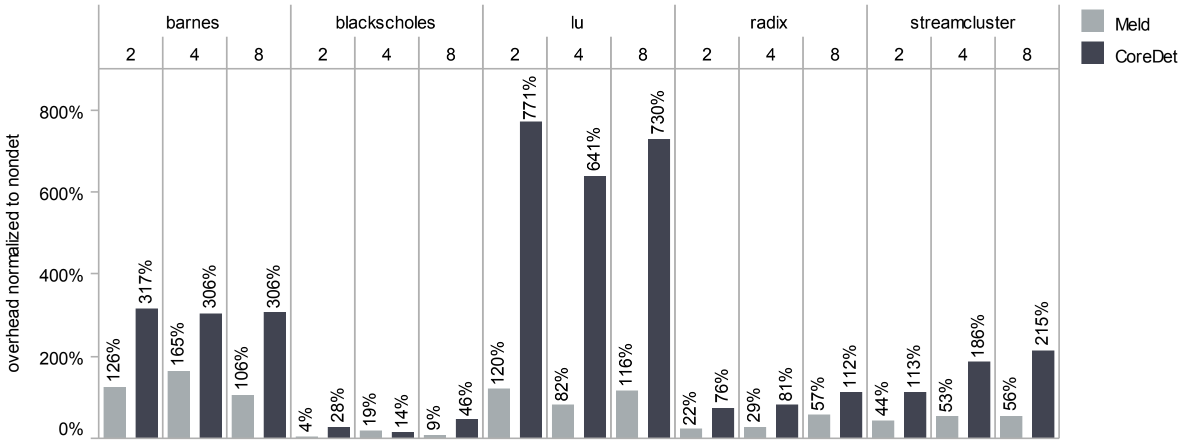

Figure 2. Performance of MELD (light bars) and CoreDet (dark bars) normalized to nondeterministic execution with the same number of threads.

just as fast as nondeterminism). While CoreDet experiences large performance overheads for some workloads, MELD consistently reduces CoreDet's overheads by 2-6x. More importantly, MELD's overhead is quite reasonable in an absolute sense. Programs running with MELD never experience more than a 2.6x slowdown over nondeterministic execution, even when running with 8 threads. Many of our workloads have overheads of 50% or lower, bringing determinism for arbitrary programs into the realm of practical applicability.

5.3 Usability

To evaluate MELD's usability, we quantitatively measure its annotation burden. Table 1 shows the size of each benchmark in lines of code, and gives the number of declaration annotations and casts used in MELD-ing each program. For lu and streamcluster, we also had to clone a small function to work around our prototype's lack of qualifier polymorphism. The annotation burden for MELD is quite low and we found the process of adding qualifiers straightforward, particularly since our compiler provides precise compile-time errors about missing or incorrect qualifiers. Table 1 also shows that the data parallel pattern is a common idiom for expressing the safe parallelism in our workloads.

The right-hand columns of Table 1 show the number of lock acquire, condition variable wait and barrier wait statements in our workloads. The presence of locks and condition variables is a sign that a benchmark will not be eminently suitable for porting to a deterministic language. Other, subtler cases like complicated array indexing can also be an issue. Ultimately, only one of our workloads – blackscholes – can be readily ported to a deterministic language. MELD's more flexible model, however, can easily accomodate more sophisticated workloads.

6. Related Work

MELD is most related to previous work on deterministic execution systems and deterministic languages. We elide discussion of record-and-replay schemes as they provide repeatability (with additional space overhead) but no guarantee of a consistent outcome like determinism does.

Deterministic execution systems allow programs written in standard parallel languages like C and C++ to execute deterministically. DMP [1] proposed hardware support for deterministic execution. Subsequently, Calvin [17] and RCDC [5] proposed simpler hardware that eschewed the need for speculation. These hardware proposals offer high performance (typically less than 30% runtime overhead) but require new processor architectures. MELD builds upon the CoreDet deterministic compiler [3, 5] that enforces determinism purely in software. Dthreads [8] provides determinism for arbitrary programs using copy-on-write paging for isolation in lieu of CoreDet's store buffers. Dthreads' approach, while occasionally vulnerable to forward progress issues, is generally faster than CoreDet's; replacing the

CoreDet runtime with Dthreads would be interesting future work. The dOS [6] operating system provides deterministic execution as an OS service, enforcing determinism for arbitrary binaries (no recompilation necessary) but at high runtime cost. Kendo [2] provides determinism for race-free programs via deterministic synchronization, and is able to elide CoreDet's memory and quantum formation instrumentation as a result. Grace [4] and Determinator [7] provide determinism for fork-join programs. Tern [18] and Peregrine [19] use schedule memoization to constrain the execution of a parallel program. While not offering true determinism, Tern and Peregrine show that many programs will run well even when limited to just a handful of schedules.

Many **deterministic parallel languages** have been proposed. Many are functional languages, like NESL [10] and Data Parallel Haskell [20]; their lack of mutability greatly assists automated reasoning about parallelism. SHIM [21] is a deterministic message-passing language based on MPI. The StreamIt [9] language provides determinism by using streams in a pipeline processing model instead of using shared memory. Jade [11] is an implicitly parallel language, where programmers write sequential code that is annotated to allow a compiler to extract parallelism automatically. Deterministic Parallel Java [12] is a set of extensions to Java's type system that uses effects to reason about interference between concurrent operations. These effects can in many cases by inferred automatically [12]. DPJ supports fork-join parallelism over array and tree data structures.

A recent extension to DPJ [13] provides support for code-centric nondeterminism within DPJ's existing fork-join parallel framework. However, as we explain in Section 4, determinism's non-composability means that, in principle, a single nondeterministic construct can invalidate the determinism of a DPJ program because, with a code-centric approach, the data manipulated by a nondeterministic operation is sandboxed only for the duration of such an operation.

7. Conclusion

We have presented MELD, a new system for integrating deterministic languages within a deterministic execution system. We leverage deterministic execution's support for arbitrary parallel programs, and deterministic languages' support for fast, statically-checked determinism to accelerate the performance of deterministic execution on programs that could not be readily expressed in a deterministic language. Our results show that integrating Deterministic Parallel Java into the CoreDet deterministic execution system can improve determinism's performance by 2-6x, while requiring little annotation overhead.

References

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

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

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

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

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

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

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

[8] T. Liu, C. Curtsinger, and E. Berger. Dthreads: Efficient and Deterministic Multithreading. In *SOSP*, 2011.

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

[10] G. Blelloch. NESL: A Nested Data-Parallel Language (Version 3.1). Technical report, Carnegie Mellon University, Pittsburgh, PA, 2007.

[11] M. Rinard and M. Lam. The Design, Implementation, and Evaluation of Jade. *ACM TOPLAS*, 20(3), 1988.

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

[13] R. Bocchino, S. Heumann, N. Honarmand, S. Adve, V. Adve, A. Welc, and T. Shpeisman. Safe Nondeterminism in a Deterministic-by-Default Parallel Language. In *POPL*, 2011.

[14] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[16] M. Tiwari, H. Wassel, B. Mazloom, S. Mysore, F. Chong, and T. Sherwood. Complete Information Flow Tracking from the Gates Up. In *ASPLOS*, 2009.

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

[18] H. Cui, J. Wu, C. c. Tsai, and J. Yang. Stable Deterministic Multithreading through Schedule Memoization. In *OSDI*, 2010.

[19] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang. Efficient Deterministic Multithreading through Schedule Relaxation. In *SOSP*, 2011.

[20] M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A Status Report. In *Workshop on Declarative Aspects of Multicore Programming (DAMP)*, 2007.

[21] S. A. Edwards and O. Tardieu. SHIM: A Deterministic Model for Heterogeneous Embedded Systems. In *EMSOFT*, 2005.