TARDIS: Task-level Access Race Detection by Intersecting Sets *

Weixing Ji
Beijing Institute of Technology
jwx@bit.edu.cn

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

Abstract

A data race detector aids in the debugging of parallel programs, and may help to ensure determinism in a language that also excludes synchronization races. Most existing race detectors use *shadow memory* to track references to shared data on the fly. An attractive alternative, particularly in languages with constrained, *split-merge* parallelism, is to record access sets in each concurrent task, and intersect these sets at merge points. In comparison to shadow-memory based race detectors, access-set based detection has the potential to significantly improve locality and reduce per-access overhead.

We have implemented an access-set based data race detector, TARDIS, in the context of a deterministic parallel Ruby extension. TARDIS runs in parallel. For our deterministic language, it is also *precise*: it finds all data races for a given program on a given input. On a set of six parallel benchmarks, TARDIS is slightly slower than the state-of-the-art SPD3 shadow-memory based race detector in two cases, but faster in the other four—by as much as 3.75×. Its memory consumption is also comparable to—and often better than—that of SPD3. Given these encouraging results, we suggest directions for future development of access-set based data race detectors.

1. Introduction

Data races are increasingly seen as bugs in parallel programs, especially among authors who favor deterministic parallel programming. Data race detectors are thus increasingly popular tools. Many existing detectors track the happens-before relationship, looking for unordered conflicting accesses [6, 7, 9, 14, 15]. Some track lock sets, looking for conflicting accesses not covered by a common lock [21, 25]. Still others take a hybrid approach [16, 23, 24].

Most existing detectors assume a set of threads that remains largely static over the history of the program. Such detectors can be used for fine-grain task-based programs, but only at the risk of possible *false negatives*: data races may be missed if they occur between tasks that happen, in a given execution, to be performed by the same "worker" thread. Conversely, lock-set–based detectors may suffer from *false positives*: they may announce a potential data race when conflicting accesses share no common lock, even if program logic ensures through other means that the accesses can never occur in parallel.

Both false negatives and false positives may be acceptable in a detector intended for debugging of complex programs. Our interest in data race detection, however, is rooted in dynamic (scripting) languages, where ease of programming is a central goal. We believe determinism to be particularly appealing in such languages. Toward that end, we are exploring "split-merge" extensions to Ruby (cobegin, parallel iterators) that facilitate the creation of numerous concurrent tasks. We insist that these tasks be independent of one another. In principle, independence can be guaranteed by the type system [4] or by explicit assignment of "ownership" [10], but we believe the resulting complexity to be inappropriate for scripting—hence the desire for dynamic data race detection.

To ensure program correctness with minimal burden on the programmer, a data race detector must be precise: it should identify all (and only) those conflicting accesses that are logically concurrent (unordered by happens-before) in a given program execution—even if those accesses occur in the same worker thread. The Nondeterminator detector for Cilk (without locks) meets this requirement [8], but runs sequentially (an extension to accommodate locks is also sequential [5]). Mellor-Crummey's offset-span labeling provides a space-efficient alternative to vector clocks for fine-grain split-merge programs, but was also implemented sequentially [13]. The state of the art would appear to be the Habanero Java SPD3 detector [18], which runs in parallel and provides precise data race detection for arbitrary split-merge (async-finish) programs.

Existing race detectors typically use *shadow memory* to store metadata. On each data access, the detector consults the associated shadow memory to reason about concurrency between conflicting operations. Unfortunately, metadata accesses must generally be synchronized to protect against concurrent access and dynamic resizing of objects. Even when tasks access disjoint sets of fields, this synchronization tends to induce cache misses, and may limit scalability.

In this paper, we propose a new race detector, TARDIS, that can outperform shadow-memory–based detectors on programs with split-merge task graphs. As each task executes, TARDIS records its reads and writes in a local ac-

* This work was conducted while Weixing Ji was a visiting scholar at the University of Rochester, supported by the Chinese Scholarship Council (No. 2011603518). Li Lu and Michael Scott were supported in part by the National Science Foundation under grants CCR-0963759, CCF-1116055, and CNS-1116109.

cess set. At a task merge point, access sets from concurrent tasks are intersected to find conflicting operations, and then merged (union-ed) and retained for subsequent comparison to other tasks from the same or surrounding concurrent constructs. Given that tasks are properly nested, the total number of intersection and merge operations is guaranteed to be linear (rather than quadratic) in the number of tasks in the program.

Ronsse et al. [19, 20] propose a trace/replay-based race detector for general, fork-join programs. It reasons about concurrency among thread traces by tracking happens-before order. In TARDIS, we observe that split-merge style parallelism makes trace (access-set)–based race detection significantly more attractive than it is in the general case, raising the possibility of sound, complete, and (relatively) inexpensive race detection in support of guaranteed determinism.

If we consider the average time to access and update shadow memory to be C_{sm}, a program with N accesses will incur detector overhead of NC_{sm}. An access-set–based detector performs a smaller amount of (entirely local) work on each access, and postpones the detection of conflicts to the end of the current task. It merges accesses to the same location within each access set, so the end-of-task work is proportional to the memory footprint of the task, rather than the number of its dynamic accesses. If a task accesses K locations (object fields), there will be K entries in its access set. Suppose the average time to insert an item in an access set is C_{as}, and the time to check a location for conflicts is C_{ck}. Then the total access-set–based detector overhead will be $NC_{as} + KC_{ck}$. In general, it seems reasonable to expect $N >> K$ and $C_{as} < C_{sm}$, which suggests that access-set–based detection may be significantly faster than shadow-memory–based detection. Worst case, if there are T tasks, M memory locations in the program and $K \approx M$, total space for access-set–based data race detection will be $O(TM)$, versus $O(M)$ for shadow-memory–based detection. In practice, however, it again seems reasonable to expect that a set of concurrent tasks will, among them, touch much less than all of memory, so $TK < M$—maybe even $TK << M$—in which case access-set–based detection may be more space efficient than shadow memory.

2. Parallel Extensions for Ruby

We have prototyped TARDIS as part of a parallel extension of Ruby, implemented in the JRuby [1] virtual machine. JRuby in turn is written in Java, and has scalable multithread performance. Our extension provides a built-in thread pool and a Cilk [3]-style work-stealing scheduler. Tasks are created using constructs such as *co-begin* (`co`) and *parallel iterators* (`.all`), examples of which can be seen in Figure 1. The `co` construct accepts an arbitrary list of lambda expressions. The `.all` method is reminiscent of the built-in `.each`, but does not imply any order of execution. We provide a built-in implementation of `.all` for ranges and arrays; additional implementations can be defined for other collection types.

```
1 a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
2 b = Array.new(10)
3 c = Array.new(10)
4
5 co → { a.all { |i| b[i] = f(a[i]) } },
6    → { a.all { |i| c[i] = g(a[i]) } }
```

Figure 1. Example code segment for the two parallel constructs

Both `co` and `.all` can be nested. Tasks from the same construct are semantically concurrent, and may run in any order or in parallel, depending on the workload and the underlying system. When all tasks of a given construct have completed, execution will continue in the surrounding context. The result is a properly nested, split-merge tree of tasks.

Because our constructs are free by design of synchronization races, we know that if all concurrent tasks are independent (i.e., free of data races), programs will satisfy the strong "Singleton" definition of deterministic execution [12].

Figure 1 shows an example code segment for both parallel constructs. If `f()` and `g()` are pure functions, we can safely assert that all tasks in this example will be mutually independent. In the `co` construct at Line 5, two concurrent tasks are started. Each task then uses a call to `.all` to initialize an array in parallel. Execution of the original task (the main program) waits for the two branches of the `co` to complete before continuing. The thread in each branch in turn waits for all instances of `.all`.

3. TARDIS Design

3.1 Data Structure for Access Sets

Because objects in Ruby may move due to dynamic resizing or compacting garbage collection, conflicting language-level references cannot reasonably be detected by address. Each object, however, has a unique, unchanging `id`, assigned when the object is created. We use ⟨*id*, *field_no*⟩ pairs to identify accessed locations.

Because the number of memory operations issued by tasks varies dramatically, we use an adaptive, hybrid implementation of access sets to balance performance and memory consumption. As illustrated in Figure 2, when a task starts, two fix-sized lists are allocated for it, one to store reads, the other writes. Accesses are recorded sequentially into the two lists. If a task executes "too many" accesses, however, the lists may overflow, at which point we convert them to hash tables. Each hash table implements a mapping from object `id`s to bitmaps containing a read bit and a write bit for each field. Generally speaking, sequential lists require less work on each access—a simple list append. Once we switch to the hash table, each access requires us to compute the hash function, search for the matching bucket, and index

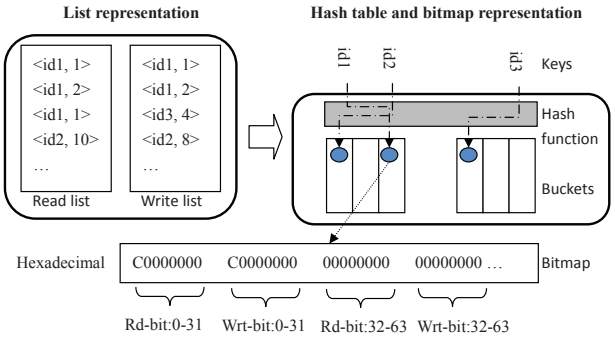

Figure 2. TARDIS's hybrid representation of access sets. An access set is started with list representation (left) and may convert to hash table representation (right). The lower part of the figure shows an example of the bitmap that represents object fields.

into the bitmap. Hash tables are more space efficient, however, because they eliminate duplicate entries.

Per-object bitmaps provide a compact representation of field access information, and facilitate fast intersection and union operations. For simple objects, a single 64-bit word can cover reads and writes for 32 fields. Arrays use expandable bitmaps with an optional non-zero base offset. Tasks that access a dense subset of an array will capture their accesses succinctly.

Some systems (e.g. SigRace [14]) have used Bloom filter "signatures" to capture access sets, but these sacrifice precision. We considered using them as an initial heuristic, allowing us to avoid intersecting actual access sets when their signatures intersected without conflict. Experiments indicated, however, that the heuristic was almost never useful: signature creation increases per-access instrumentation costs, since full access sets are needed as a backup (to rule out false positives). Moreover, signature intersections are significantly cheaper than full set intersections only when the access sets are large. But large access sets are precisely the case in which signatures saturate, and require the backup intersection anyway.

3.2 Race Detection Algorithm

In TARDIS, each task `T` maintains the following fields for race detection:

- **local_set**: an access set containing the union of the access sets of completed child tasks of `T`
- **current_set**: an access set containing all the accesses performed so far by `T` itself
- **parent**: a reference to `T`'s parent task

As shown in algorithm 1, when n child tasks are spawned by T_p at a splitting point, two new access sets (T_{ci}.*current_set* and T_{ci}.*local_set*) are created and assigned to each child task T_{ci}. At the same time, T_{ci}.*parent* is set to T_p so that T_{ci} can find its merge-back point at the end of its execution.

Algorithm 1 On Task Split

Require: parent task T_p, number of child tasks n
1: **for** $i = 1 \rightarrow n$ **do**
2: $T_{ci} \leftarrow$ new task
3: T_{ci}.*current_set* $\leftarrow \varnothing$
4: T_{ci}.*local_set* $\leftarrow \varnothing$
5: T_{ci}.*parent* $\leftarrow T_p$
6: **end for**

After the split operation, the parent task T_p waits for all child tasks to terminate. Algorithm 2 shows the algorithm performed by TARDIS when each child task T_c runs to its merge point. Since children may run in parallel, the algorithm must synchronize on T_p (Line 2). As an optimization, tasks executed by a single worker thread may first be merged locally, without synchronization, and then synchronously merged into T_p.

In Algorithm 2, we use T_p.*local_set* to store all the memory locations accessed by all the concurrent siblings that joined back before the current task T_c. Consequently, intersection is only performed between T_c.*current_set* and T_p.*local_set*. After that, T_c.*current_set* is merged into T_p.*local_set* so that following tasks can be processed in a similar fashion. At the end of Algorithm 2, T_p.*local_set* is merged into T_p.*current_set* if T_c is the last child task merging back. T_p.*local_set* is also cleared so that it can be reused at the next merge point.

Algorithm 2 On Task Merge

Require: child task T_c
1: $T_p \leftarrow T_c$.*parent*
2: **sync** on T_p
3: **if** T_p.*local_set* $\cap$ T_c.*current_set* $\neq \varnothing$ **then**
4: Report a data race
5: **end if**
6: T_p.*local_set* $= T_p$.*local_set* $\cup$ T_c.*current_set*
7: **if** T_c is the last task to join **then**
8: T_p.*current_set* $= T_p$.*current_set* $\cup$ T_p.*local_set*
9: T_p.*local_set* $\leftarrow \varnothing$
10: **end if**
11: **end sync**

The *local_set* for each task is always allocated as a hash set. As a result, each intersection / merge in the algorithm is performed either over a hash set and a list set, or over two hash sets. In either case, each entry in the set has a read bit and a write bit. The intersection operation attempts to confirm that whenever an $\langle id, field\_no \rangle$ pair is found in both sets, only the read bits are set.

Whenever task `T` reads or writes a memory location, we insert a new $\langle id, field\_no \rangle$ pair into (the current implementation of) its access set, as shown in Algorithm 3.

Algorithm 3 On Read/Write

Require: Object *id*, field *field_no*, operation type t (read-/write), task T

1: $T.current\_set$.add($\langle id, field\_no \rangle, t$)

Table 1. Benchmarks used for evaluation

Benchmark	Source	Input
Blackscholes	PARSEC	simlarge
Swaptions	PARSEC	simsmall
Series	Java Grande	A (small)
Crypt	Java Grande	A (small)
SparseMatMult	Java Grande	A (small)
SOR	Java Grande	A (small)

4. Performance Evaluation

As there are no parallel and computing intensive benchmarks available for Ruby at present, we manually translated 6 applications from standard parallel benchmark suites (see Table 1). The Blackscholes and Swaptions programs are originally from the PRASEC suite [2], where they were written in C. The Series, Crypt, SparseMatMult, and SOR programs are originally from the Java Grande Forum (JGF) suite [22]. All six Ruby versions were parallelized using the constructs introduced in section 2.

We have implemented our data race detector and parallel library in the JRuby 1.6.7.2 virtual machine, which in turn runs on a 64-bit OpenJDK with version 1.8. Our experiments were conducted on an Intel Xeon E5649 system with 2 processors, 6 cores per processor, 2 threads per core (i.e., 12 cores and 24 threads total), and 12 GB of memory, running Linux 2.6.34. We also constructed best-effort reimplementations of the Cilk Nondeterminator [8] and SPD3 [18] race detectors, based on the descriptions in the respective papers. Each object in SPD3 shadow space is protected by a sequence lock [11] to accommodate concurrent access and potential resizing.

Figure 3 shows relative speeds for Nondeterminator, SPD3, and TARDIS. All data were collected with a maximum JVM heap size of 4 GB. 1024 slots are allocated for both reads and writes in the list-based representation of access sets. The baseline for speed measurement is the (parallelized) Ruby version of each benchmark, running on a single thread with race detection turned off. Each data point represents the average of 5 runs. Since Nondeterminator is a sequential detector, it is reported only at 1 thread.

Overall, TARDIS outperforms SPD3 by substantial margins in three applications, and by modest amounts in another. It is slightly slower in the other two. The large wins in Blackscholes, Crypt and SparseMatMult stem from tasks with relatively small memory footprints, but large numbers of repeat accesses to the same locations. SPD3 allocates 3 nodes for each task in its Dynamic Program Structure Tree (DPST), and performs relatively more expensive tree operations for most of the memory accesses. TARDIS reduces the instrumentation overhead on each access, and detects conflicts using a subsequent per-location pass.

In Series and Swaptions, TARDIS is slightly slower than SPD3. Profiling reveals a very small number of memory accesses per task in Series and a very large number of memory locations per task in Swaptions, neither of which offers an opportunity to optimize repeated access. In both applications the time spent initializing and managing access sets dominates the run time of our algorithm. Nondeterminator outperforms TARDIS at one thread in SparseMatMult and Series, but is unable to scale up.

Peak memory consumption of all benchmarks for TARDIS and SPD3 appears in Figure 4. TARDIS consumes more memory than SPD3 in Swaptions, but substantially less in Blackscholes, Crypt, and SparseMatMult. The large size of shadow memory and the DPST in SPD3 may contribute to the performance advantage of TARDIS, due to cache pressure and the overhead of garbage collection.

5. Conclusions and Future Work

In this paper, we introduced TARDIS, a task-level access-set–based data race detector for parallel dynamic languages. In our experiments, TARDIS generates promising results relative to comparable state-of-the-art detectors. Relative to uninstrumented execution, TARDIS introduces a typical slowdown of approximately 2× and a maximum of less than 5×. While this is probably still too slow to enable in production runs, it is eminently reasonable during testing and debugging.

Topics for future work include:

More generalized structured parallelism. TARDIS currently works only in programs whose tasks are properly nested. For more general `fork`/`join` programs we may lose the guarantee of a linear number of set intersections and unions, but overhead may still be reasonable in practice.

Support commutative atomic operations. Our current parallel JRuby extension lacks constructs that allow explicit synchronization among tasks. One appealing route that would still preserve determinism is to allow atomic execution, within tasks, of operations that conflict internally, but commute at the level of program semantics. Extensions to TARDIS would be required to detect races among such operations.

Out-of-band (parallel) data race detection. Because it relies on access sets, data race detection in TARDIS can be performed asynchronously with respect to a program's normal execution. In our current implementation, detection (merging of access sets) is synchronous, and may be on a program's critical path. In principle, we could move set intersection and union operations to a separate thread,

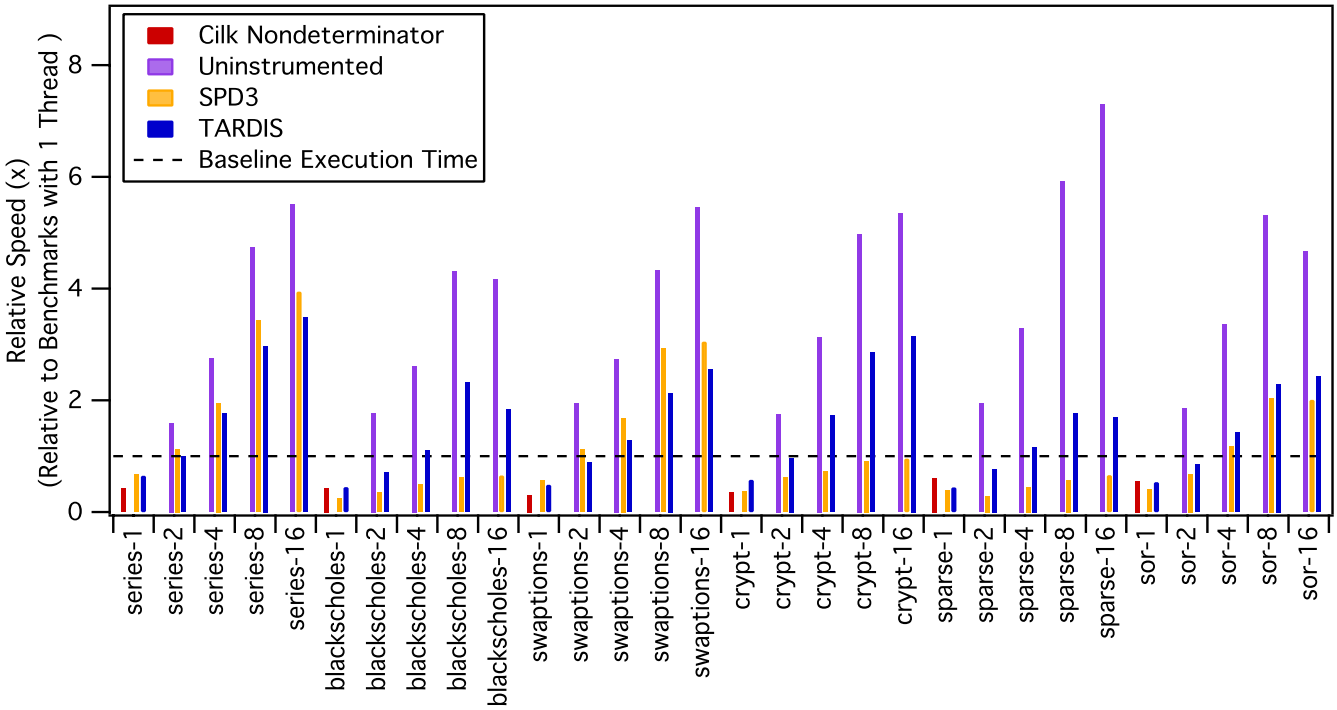

Figure 3. Relative speed of uninstrumented parallel extension, Nondeterminator, SPD3, and TARDIS. "Benchmark Name–x" indicates a benchmark run with x threads. All speed results are relative to the 1-thread uninstrumented case, which is represented by the dashed line in the figure (we omit its bars). Performance information for Nondeterminator is reported only at 1 thread, since the data race detector is sequential.

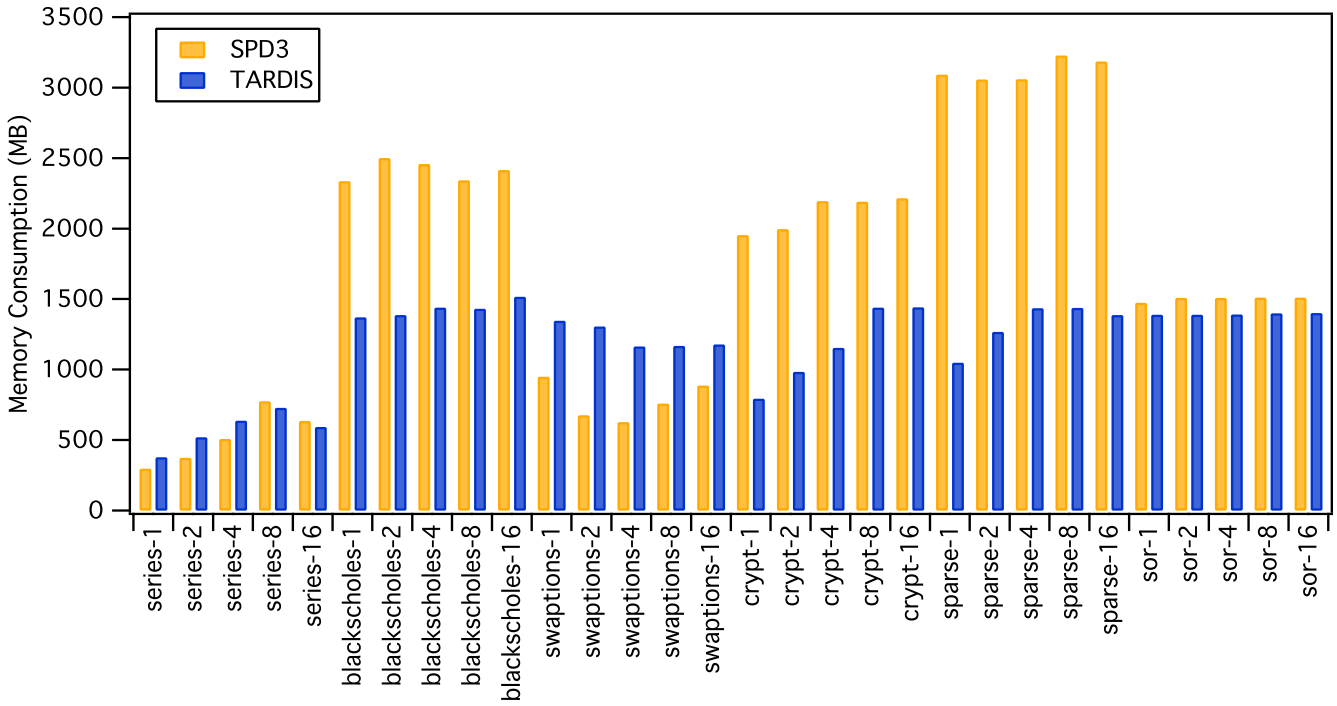

Figure 4. Maximum memory usage of TARDIS and SPD3 for all benchmarks. "Benchmark-x" indicates a benchmark run with x threads.

potentially shortening the critical path. Such an option is not available to shadow-memory–based detectors.

More space-efficient access-set representation. Experiments have shown that TARDIS's performance is sensitive to the memory consumption of access sets. We plan to explore more space-efficient set representations, as well as adaptive techniques more sophisticated than simply switching to a hash table on list overflow.

Possible hardware support. Several groups have investigated hardware support for lock-set [25] and happens-before [6, 14, 17]–based data race detection. We hope to explore similar optimizations for access-set–based detection.

References

[1] Jruby. Dec. 2012. `http://jruby.org/`.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Toronto, ON, Canada, Oct. 2008.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[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. In *Object-Oriented Programming Systems, Langauges, and Applications (OOPSLA)*, Orlando, FL, Oct. 2009.

[5] C.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *Symp. on Parallel Algorithms and Architectures (SPAA)*, Puerto Vallarta, Mexico, June–July 1998.

[6] J. Devietti, B. P. Wood, K. Strauss, L. Ceze, D. Grossman, and S. Qadeer. RADISH: Always-on sound and complete race detection in software and hardware. In *Intl. Symp. on Computer Architecture (ISCA)*, Portland, OR, June 2012.

[7] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. IFRit: Interference-free regions for dynamic data-race detection. In *Object-Oriented Programming Systems, Langauges, and Applications (OOPSLA)*, Tucson, AZ, Oct. 2012.

[8] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Symp. on Parallel Algorithms and Architectures (SPAA)*, Newport, RI, June 1997.

[9] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

[10] S. T. Heumann, V. S. Adve, and S. Wang. The tasks with effects model for safe concurrency. In *18th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Shenzhen, China, Feb. 2013.

[11] C. Lameter. Effective synchronization on Linux/ NUMA systems. In *Gelato Federation Meeting*, San Jose, CA, May 2005.

[12] L. Lu and M. L. Scott. Toward a formal semantic framework for deterministic parallel programming. In *Intl. Symp. on Distributed Computing (DISC)*, Rome, Italy, Sept. 2011.

[13] J. Mellor-Crummey. On-the-fly detection of data races for programs with nested fork-join parallelism. In *Supercomputing Conf.*, Albuquerque, NM, Nov. 1991.

[14] A. Muzahid, D. S. Gracia, S. Qi, and J. Torrellas. SigRace: Signature-based data race detection. In *Intl. Symp. on Computer Architecture (ISCA)*, Austin, TX, June 2009.

[15] A. Nistor, D. Marinov, and J. Torrellas. Light64: Lightweight hardware support for data race detection during systematic testing of parallel programs. In *42nd IEEE/ACM Intl. Symp. on Microarchitecture (MICRO)*, New York, NY, Dec. 2009.

[16] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *9th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, San Diego, CA, June 2003.

[17] M. Prvulovic. CORD: Cost-effective (and nearly overhead-free) order-recording and data race detection. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, Austin, TX, Feb. 2006.

[18] R. Raman, J. Zhao, V. Sarkar, M. T. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Beijing, China, June 2012.

[19] M. Ronsse and K. De Bosschere. Jiti: Tracing memory references for data race detection. In *Intl. Parallel Computing Conf. (PARCO)*, Bonn, Germany, Sept. 1997.

[20] M. Ronsse, B. Stougie, J. Maebe, F. Cornelis, and K. D. Bosschere. An efficient data race detector backend for DIOTA. In *Intl. Parallel Computing Conf. (PARCO)*, Dresden, Germany, 2003.

[21] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. on Computer Systems*, 15(4):391–411, Nov. 1997.

[22] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel Java Grande benchmark suite. In *Supercomputing Conf.*, page 8, Denver, CO, Nov. 2001.

[23] X. Xie and J. Xue. Acculock: Accurate and efficient detection of data races. In *9th IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO)*, Seattle, WA, Mar. 2011.

[24] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *ACM Symp. on Operating Systems Principles (SOSP)*, Brighton, United Kingdom, Oct. 2005.

[25] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-assisted lockset-based race detection. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, Phoenix, AZ, Feb. 2007.