Dynamic Enforcement of Determinism in a Parallel Scripting Language

Li Lu
University of Rochester
llu@cs.rochester.edu

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

Michael L. Scott
University of Rochester
scott@cs.rochester.edu

Abstract

Determinism is an appealing property for parallel programs, as it simplifies understanding, reasoning and debugging. It is particularly appealing in dynamic (scripting) languages, where ease of programming is a dominant design goal. Some existing parallel languages use the type system to enforce determinism statically, but this is not generally practical for dynamic languages. In this paper, we describe how determinism can be obtained—and dynamically enforced/verified—for appropriate extensions to a parallel scripting language. Specifically, we introduce the constructs of Deterministic Parallel Ruby (DPR), together with a run-time system (TARDIS) that verifies properties required for determinism, including correct usage of reductions and commutative operators, and the mutual independence (data-race freedom) of concurrent tasks. Experimental results confirm that DPR can provide scalable performance on multicore machines and that the overhead of TARDIS is low enough for practical testing. In particular, TARDIS significantly outperforms alternative data-race detectors with comparable functionality. We conclude with a discussion of future directions in the dynamic enforcement of determinism.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programing; D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages, Ruby; D.2.5 [*Software Engineering*]: Testing and Debugging—Diagnostics

General Terms Design, Languages, Reliability

Keywords Determinism, deterministic parallelism, data races, scripting language

1. Introduction

Deterministic parallel programming [3] is an increasingly popular approach to multithreaded systems. While the exact definition of determinism varies from system to system [22], the key idea is that the run-time behavior of a parallel program (or at least the "important" aspects of its run-time behavior) should not depend on the thread interleavings chosen by the underlying scheduler.

PLDI '14, June 09–11 2014, Edinburgh, United Kingdom.

http://dx.doi.org/10.1145/2594291.2594300

We believe determinism to be particularly appealing for dynamic "scripting" languages—Python, Ruby, Javascript, etc. Raw performance has never been a primary goal in these languages, but it seems inevitable that implementations for future many-core machines will need to be successfully multithreaded. At the same time, the emphasis on ease of programming suggests that the tradeoff between simplicity and generality will be biased toward simplicity, where determinism is an ideal fit.

In general, determinism can be obtained either by eliminating races (both data races and synchronization races) in the source program ("language level" determinism), or by ensuring that races are always resolved "the same way" at run time ("system level" determinism). While the latter approach is undeniably useful for debugging, it does not assist with program understanding. We therefore focus on the former.

Language-level determinism rules out many historically important programming idioms, but it leaves many others intact, and it offers significant conceptual benefits, eliminating many of the most pernicious kinds of bugs, and allowing human readers to more easily predict a program's behavior from its source code. Using well-known constructs from previous parallel languages, we have defined a language dialect we call Deterministic Parallel Ruby (DPR). Our goal is not to invent the ideal language, but rather to demonstrate that determinism can easily be embedded *and dynamically verified* in a parallel scripting language. The flexible syntax of Ruby—in particular, its easy manipulation of lambda expressions—makes it an ideal vehicle for this work.

The most basic parallel constructs in DPR are co-begin and unordered iterators, which create properly nested "split-merge" tasks. We interpret the use of these constructs as an assertion on the programmer's part that the tasks are independent—an assertion we need to check at run time. By default, we regard two tasks as independent if neither writes a location (object field) that the other reads or writes. A violation of this rule (a data race) constitutes a conflict because writes do not *commute* with either reads or writes: conflicting accesses that occur in a different order are likely to lead to an observably different execution history [22].

But not always. One of the principal problems with a read-write notion of conflict is that data races are defined at a very low level of abstraction. Scripting programs often manipulate higher-level objects, some of whose operations commute semantically even though they comprise ostensibly conflicting reads and writes. To raise the level of abstraction in DPR, we support a variety of built-in *reduction* objects. We also provide a general mechanism to define the commutativity relationships among (atomically invoked) methods of arbitrary classes. A set, for example, might indicate that insert operations commute with other inserts (and likewise

removes with removes and lookups with lookups), but none of the operations (in the general case) commutes with the other two.[1]

In addition to split-merge tasks, reductions, and atomic commutative operations, DPR also provides futures and pipelines. In general, invocation of a future function might be taken as an assertion that the call is independent of the caller's continuation, up to the point of use; for simplicity, we infer the stronger assertion that the function is *pure*. Pipelines, for their part, allow us to accommodate the equivalent of loop-carried dependences: they constitute an assertion that for elements of a stream, each pipeline stage is independent of the subsequent stage for the preceding element, the previous stage for the following element, and so forth. In ongoing work (not discussed here), we are exploring parallel implementations of operations on built-in classes, and mechanisms to accommodate limited (structured) nondeterminism, including arbitrary choice from a set and atomic handling of asynchronous events.

Our focus on shared-memory multithreading is a deliberate alternative to the process-fork–based parallelism more commonly used in scripting languages today. Multithreading incurs certain synchronization overheads (e.g., on metadata access) that are avoided by processes with separate virtual machines. At the same time, it avoids copy-in/copy-out and process start-up overheads, allowing us to consider finer-grain parallelization. It also enables certain forms of fine-grain interaction—e.g., in reductions. Most importantly (though this is a subjective judgment), many programmers appear to find it conceptually cleaner. We expect the synchronization overheads to decrease over time, with improvements in virtual machines. Our experimental results were collected on JRuby, whose threads currently display significantly better scalability than those of the standard C Ruby VM.

Other groups have added deterministic parallelism to scripting languages via speculative parallelization of semantically sequential code [8] or via high-level parallel primitives (built-in parallel arrays) [13]. Our approach is distinguished by its use of general-purpose constructs to express nested, independent tasks. Among our constructs, only reductions and other high-level operations can induce a synchronization race, and for these the commutativity rules ensure that the race has no impact on program-level semantics. Enforcement of determinism in DPR thus amounts to verifying that (a) there are no (low-level) data races and (b) higher-level operations are called concurrently only when they commute.

1.1 Log-Based Data Race Detection

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 [7, 10, 12, 24, 25]. Some track lock sets, looking for conflicting accesses not covered by a common lock [31, 38]. Still others take a hybrid approach [6, 26, 36, 37]. In general, data race detection introduces significant run-time overhead. Recent work has suggested that this overhead might be reduced by crowdsourcing [18].

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 concurrently.

In principle, task independence can be guaranteed by the type system [4] or by explicit assignment of "ownership" [15], but we believe the resulting complexity to be inappropriate for scripting—hence the desire for dynamic data race detection. Similarly, both false negatives and false positives may be acceptable when searching for suspected bugs in a complex system, but for DPR we aim to transparently *confirm the absence* of races; for this task we take as given that the detector must be *precise*: it should identify all (and only) those conflicting operations that are logically concurrent (unordered by happens-before) in a given program execution—even if those operations occur in the same worker thread.

The Nondeterminator race detector for Cilk (without locks) [11] is both sound (no false positives) and complete (no false negatives), but runs sequentially; an extension to accommodate locks is also sequential, and no longer sound [5]. Mellor-Crummey's offset-span labeling [23] provides a space-efficient alternative to vector clocks for fine-grain split-merge programs, but was also implemented sequentially. The state of the art would appear to be the Habañero Java SPD3 detector [27], which runs in parallel and provides precise (sound and complete) data race detection for arbitrary split-merge (async-finish) programs.[2]

Like most recent race detectors, SPD3 relies on *shadow memory* to store metadata for each shared memory location. On each read and write, the detector accesses the metadata to reason about conflicting operations. Unfortunately, this access generally requires synchronization across threads, leading to cache misses that may limit scalability even when threads access disjoint sets of object fields. By contrast, our run-time system, TARDIS [16], logs references locally in each concurrent task, and intersects these logs at merge points.

Log-based detection has two key advantages over shadow memory. First, as we show in Section 4, it can provide a significant performance advantage. Second, it is easily extended to accommodate higher-level operations. Several past researchers, including Schonberg [32] and Ronsse et al. [29, 30] have described trace-based race detectors for general, fork-join programs. These systems reason about concurrency among thread traces by tracking happens-before. In TARDIS, we observe that split-merge parallelism makes trace (access-set)–based race detection significantly more attractive than it is in the general case: 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.

1.2 Higher-Level Operations

With appropriate restrictions on usage, the determinism of reductions, atomic commutative operations, futures, and pipelines can all be checked via extensions to log-based data-race detection.

Reduction operations, though they have their own special syntax, are treated as a subcase of atomic commutative operations (*AC ops*). As in Galois [21], commutativity relationships among methods in DPR are specified by programmers. Rather than use these relationships to infer a parallel schedule for semantically sequential tasks, however, we use them to verify the independence of semantically unordered tasks. Calls to AC ops are logged in each thread, along with reads and writes. Two tasks are then said to be independent if (1) they contain no conflicting pairs of reads and writes *other than* those in which both accesses occur within operations

[1] We could also distinguish among operations based on argument values—allowing, for example, an insert of 5 to commute with a lookup of 3. In our current work we reason based on method names alone.

[2] In recent work, concurrent with our own, the Habañero Java group has extended SPD3 to accommodate higher-level commutative operations [35]. Their implementation is based on a syntactic *permission region* construct that allows commutativity in a shadow-memory–based system to be checked once per region rather than once per method invocation.

that have been declared to commute with one another, and (2) they contain no pairs of higher-level operations on the same object *other than* those that have been declared to commute with one another.

This assumes, of course, that commutative operations have been labeled correctly by the programmer. Checking of such labels is difficult. It requires both that we formalize the notion of conflict (e.g., via formal specification of the abstract object under construction) and that we verify that the implementation matches the specification (an undecidable property in the general case). A variety of strategies have been suggested for checking [19, 20, 28]. They are, for the most part, orthogonal to the work reported here. Integrating them with TARDIS is a subject of ongoing work.

For *future*s, we start by performing a deep copy of all provided arguments. We then ensure, via run-time tracing, that an executed function performs no references outside its dynamic extent. For pipelines, we treat each cycle of the pipeline as a set of concurrent tasks, one for each pipeline stage, and use our existing infrastructure (for reads, writes, and higher-level operations) to verify their independence.

We describe our Ruby extensions and their connection to determinism in Section 2. Implementation details are described in Section 3. Performance evaluation appears in Section 4. Across a variety of applications, we observe a typical slowdown of $2\times$ and a maximum of $4\times$ when TARDIS checking is enabled. This is too slow to leave on in production use, but reasonable for testing, and often significantly faster than the original version of SPD3 (for programs that the latter can accommodate). Conclusions and future work appear in Section 5.

2. Deterministic Parallel Ruby

Our Ruby dialect adopts well-known parallel constructs from the literature, with sufficient limitations to ensure determinism. These constructs can be nested with each other. All of the constructs employ existing language syntax; this allowed us, initially, to build a library-based implementation that would run on top of any Ruby virtual machine, either sequentially or using built-in Ruby threads. Performance improved significantly, however, when we implemented a lightweight task manager *inside* the virtual machine. All results reported in Section 4 employ such an integrated implementation—specifically, within the JRuby [17] virtual machine.

2.1 Parallel Constructs

Independent tasks The `co` construct in DPR is intended for task parallelism. It accepts an arbitrary list of lambda expressions (introduced in Ruby with the `->` sign); these may be executed in arbitrary order, or in parallel:

`co ->{` *task*$_1$ `}, ->{` *task*$_2$ `}, ...`

The `.all` method is intended for data parallelism. It is reminiscent of the built-in `.each`, but like `co` 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. Given an array / range `r` and a user-defined code segment *op*, a parallel iterator will apply *op* to each of `r`'s elements in turn:

`r.all {|x|` *op* `}`

Uses of both `co` and `.all` constitute an assertion on the programmer's part that the constituent tasks are *independent*. With the possible exception of reductions and atomic commutative ("AC") operations, as described below, independence means that no task writes a location that another, concurrent task reads or writes. The lack of access conflicts means that the values read—and operations performed—by concurrent tasks will be the same across all possible interleavings: a program based on only `co` and `.all` satisfies the strong *Singleton* definition of determinism [22].

Reductions Reductions are provided as a built-in class with `push` and `get` methods. Use of a reduction is an assertion by the programmer that the `push` calls are mutually commutative and associative, and that they are internally synchronized for safe concurrent access. We provide variants for addition, multiplication, minimum, and maximum on integers; others are easily defined. Our variants operate locally within each worker thread until the end of a parallel construct, at which point they perform a cross-thread merge.

A simple accumulator would be created via

```
s = Reduction.new(Reduction::IntegerAdd)
```

Concurrent tasks could then safely insert new values via

```
s.push(val)
```

After all accumulating tasks have finished, the result could be retrieved with `s.get`.

Assuming that `push` operations are indeed commutative and associative, the value retrieved by `get` will not depend on the order of the calls to `push`. A program in which otherwise independent tasks employ a reduction object will therefore satisfy the weaker *ExternalEvents* definition of determinism [22]: synchronization order and dataflow may vary from one run of the program to another (a fact that might be visible in a debugger), but output values will always be the same.

Atomic commutative operations (AC ops) In the base case of independent concurrent tasks, concurrent reads of a given location x do not conflict with one another, because they are mutually commutative: when thread histories interleave at run time, program behavior will not depend on which read happens first. A write to x, on the other hand, will conflict with either a read or a write of x in another thread (and thus violate independence) because writes do not commute with either reads or writes: program behavior is likely to depend on which operation happens first.

In a fashion reminiscent of *boosting* [14] in transactional memory systems, we can also consider the commutativity of higher-level operations, which may conflict at the level of reads and writes without compromising the determinism of the program at a higher level of abstraction (so long as each operation executes atomically).

DPR allows the programmer to identify methods that should be treated as higher-level operations for the purpose of conflict detection, and exempted from checking (with one another) at the level of reads and writes. Specifically, AC ops can be identified using the meta-programming method `setACOps`:

```
self.setACOps :m0, :m1, ..., :mk
```

All AC ops are assumed to commute with one another unless explicitly indicated otherwise with method `setNoCommute`:

```
self.setNoCommute :m0, :m1
```

Consider a concurrent memoization table that stores the values computed by some expensive function `f`. The class definition might look something like the code in Listing 1. The `lookUpOrCompute` method will modify the table on the first call with a given key, but the modifications will be semantically neutral, and the method can safely be called by concurrent tasks.

Each AC op must (appear to) execute atomically. The default way to ensure this is to bracket the body of each method with the `atomic` primitive, which DPR provides (and implements with a global lock). Alternatively, the designer of a class with AC methods

```
class ConcurrentMemoTable
  def initialize(f)
    @f = f
    @table = Hash.new
  end
  def lookUpOrCompute(key)
    atomic
      val = @table[key]
      if val == nil
        @table[key] = val = @f.call(key)
      end
      return val
    end
  end
  self.setACOps :lookUpOrCompute
end
```

Listing 1. AC op for a memoization table. The @ sign indicates an instance variable (object field).

```
class MyInStream
  def initialize
    @data = [obj0, obj1, obj2, obj3, obj4]
  end
  def getAndMove
    return @data.shift
  end
end
class MyOutStream
  def setAndMove(obj)
    puts obj.to_s
  end
end
p = ->{|x| s1} >> ->{|x| s2} >> ->{|x| s3}
p.setInStream(MyInStream.new)
p.setOutStream(MyOutStream.new)
p.run
```

Listing 2. Example for running a 3-stage pipeline. The to_s method converts an object into a string, and the puts statement outputs the string to standard output.

may choose a more highly concurrent implementation—e.g., one based on fine-grain locking.

Ideally, we should like to be able to check the correctness of commutativity annotations, but this is undecidable in general. For now, we simply trust the annotations. As noted in Section 1, there are several testing strategies that could, in future work, be used to increase confidence in the annotations [19, 20, 28].

For the most part, we expect commutativity annotations to be placed on library abstractions (sets, mappings, memoization tables, etc.) that are written once and frequently reused. To at least a certain extent, routine run-time checking may be less essential for such abstractions than it is for "ordinary" code. At the same time, it seems important to ensure that "ordinary" code does not conflict with AC ops at the level of reads and writes—e.g., by modifying in one thread an object that was passed by reference to an AC op in another thread. TARDIS does perform this checking.

In keeping with the dynamic nature of Ruby, commutativity annotations are executable rather than declarative in DPR: they can change when a class is dynamically re-opened and extended, or a method overridden.

Isolated futures A *future* is an object whose value may be computed in parallel with continued execution of its creator, up until the point that the creator (or another thread) attempts to retrieve its value. A future in DPR may be created via:

```
f = Future.new(->{|x1, x2, ...| op},
               arg1, arg2, ...)
```

The value of the future is retrieved with `f.get()`. In principle, a future will be deterministic if it is independent of all potentially concurrent computation. Rather than attempt to verify this independence, we require (and check) the stronger requirement that the future be *isolated*—that it represent a pure function, with arguments passed by deep copy (cloning) to prevent concurrent modification. Unlike the parallel tasks of `co` and `.all`, isolated futures need not be properly nested.

Pipelines Pipelines are a natural idiom with which to write programs that perform a series of computational stages for a series of input elements. Each stage may depend on (read values produced by) the same stage for preceding elements or previous stages for the current or preceding elements. If all stages execute in parallel using successive elements, and are otherwise independent, the computation remains deterministic in the strong sense of the word. A bit more formally, a pipeline P is modeled as a sequence of functions $\langle f_1, f_2, ..., f_n \rangle$, each of which represents a stage of P. A (finite or infinite) input sequence S is modeled as $\langle s_1, s_2, \ldots \rangle$. The determinism rule is then simple: stage f_a, operating on element s_i, must be independent of stage f_b, operating on element s_j, if and only if $a + i = b + j$. In the 3-stage pipeline illustrated in Listing 2, all tasks in a given cycle (vertical slice of the figure) must be mutually independent.

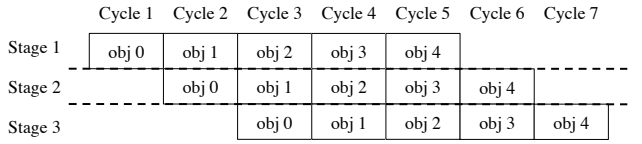

Figure 1. Example run of a 3-stage pipeline, on a sequence of 5 objects {obj0, obj1, obj2, obj3, obj4}.

DPR provides pipelines as a built-in class. A new (and empty) pipeline can be created by calling `Pipeline.new`, or by chaining together a series of lambda expressions:

```
p = ->{|x| op0} >> ->{|x| op1} >> ...
```

To avoid run-time type errors, each stage should output values of the type expected as input by the subsequent stage. Initial input and final output streams should be attached to a pipeline `p` using the `p.setInStream` and `p.setOutStream` methods. When running, the pipeline will then call the `getAndMove` method of the input stream and the `setAndMove` method of the output stream.

Stages may be executed in parallel during each *cycle* of the pipeline, with a barrier at the end of each cycle. The pipeline infrastructure moves the result of each stage to the next stage, feeds the first stage with the next value in the input stream, and stores the result to the output stream. To start the pipeline, a thread calls its `run` method. This method is blocking, and will execute until all items in the input stream have reached the output stream. Code for the pipeline of Figure 1 appears in Listing 2.

3. TARDIS Design

Our DPR run-time system, TARDIS, serves to verify adherence to the language rules that ensure determinism. The heart of TARDIS is a log-based data-race detector (described in Sec. 3.1) and a set of extensions (described in Sec. 3.2) to accommodate higher-level operations like reductions, AC ops, isolated futures, and pipelines.

We have prototyped DPR on top of the JRuby [17] virtual machine. The codebase is written in Java, and is compatible with Ruby 1.8. Within the virtual machine, TARDIS implements a Cilk-like work-stealing scheduler for lightweight tasks [2]. Though not extensively optimized, the scheduler introduces only minor overhead relative to standard Ruby threads.

For the constructs provided by DPR, the determinism checking of TARDIS is both *sound* and *complete*, provided that reductions are correctly implemented and that commutativity relationships for AC ops are correctly specified. That is, for any given input, the program will be reported to be deterministic if and only if all possible executions will have identical externally visible behavior. (Behavior observable with a debugger will also be identical if no use is made of reductions or AC ops.)

3.1 Log-based Data-race Detection

TARDIS logs reads and writes (access sets) in each task to identify conflicting accesses, which might lead to nondeterminism among supposedly independent concurrent tasks. At a task merge point, access sets from concurrent tasks are intersected to find conflicting accesses, and then union-ed and retained for subsequent comparison to other tasks from the same or surrounding concurrent constructs. More specifically, each task `T` maintains the following fields:

- **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

Each reference, as shown in Algorithm 1, is represented by a read/write bit, an object id, and a field number (while the Ruby garbage collector may move objects, an object's id and field numbers remain constant throughout its lifetime).

Algorithm 1 On read/write

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

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

When task T_p spawns n child tasks, as shown in Algorithm 2, two new access sets ($T_i.\mathit{current\_set}$ and $T_i.\mathit{local\_set}$) are created for each child T_i. At the same time, $T_i.\mathit{parent}$ is set to T_p so that T_i can find its merge-back point at the end of its execution.

Algorithm 2 On task split

Require: parent task T_p, number of child tasks n

1: **for** $i = 1 \ldots n$ **do**
2: $T_i \leftarrow$ new task
3: $T_i.\mathit{current\_set} \leftarrow \varnothing$
4: $T_i.\mathit{local\_set} \leftarrow \varnothing$
5: $T_i.\mathit{parent} \leftarrow T_p$

After the split operation, T_p waits for all child tasks to terminate. Algorithm 3 shows the work performed by TARDIS when each child T_i reaches its merge point. Since children may run in parallel, we 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 3, we use $\sqcap$ to represent "intersection" of access sets (really more of a join): $S_1 \sqcap S_2 \equiv \{\langle o, f\rangle \mid \exists\, \langle\langle o, f\rangle, t_1\rangle \in S_1,\ \langle\langle o, f\rangle, t_2\rangle \in S_2 : t_1 = \mathsf{write} \vee t_2 = \mathsf{write}\}$. We use $T_p.\mathit{local\_set}$ to store all accesses of all the concurrent siblings

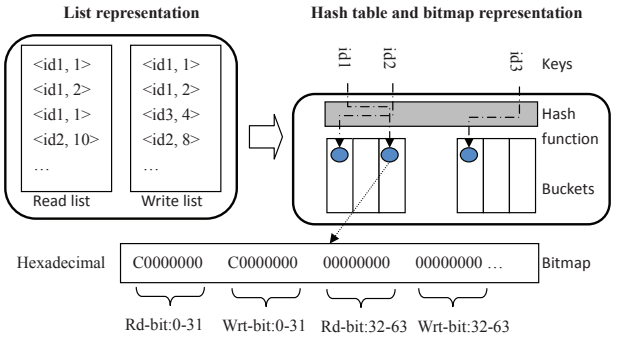

Figure 2. TARDIS's hybrid representation of access sets. An access set is started with list representation (left) and may convert to hash table representation (right). The lower part of the figure shows an example of the bitmap that represents object fields.

that merged before the current task T_c. Consequently, intersection is only performed between $T_c.\mathit{current\_set}$ and $T_p.\mathit{local\_set}$. After that, $T_c.\mathit{current\_set}$ is merged into $T_p.\mathit{local\_set}$ so that following tasks can be processed in a similar fashion. At the end of Algorithm 3, $T_p.\mathit{local\_set}$ is merged into $T_p.\mathit{current\_set}$ if T_c is the last child task merging back. $T_p.\mathit{local\_set}$ is also cleared so that it can be reused at the next merge point.

Algorithm 3 On task merge

Require: child task T_c

1: $T_p \leftarrow T_c.\mathit{parent}$
2: **sync** on T_p
3: **if** $T_p.\mathit{local\_set} \sqcap T_c.\mathit{current\_set} \neq \varnothing$ **then**
4: report a data race
5: $T_p.\mathit{local\_set} \leftarrow T_p.\mathit{local\_set} \cup T_c.\mathit{current\_set}$
6: **if** T_c is the last task to join **then**
7: $T_p.\mathit{current\_set} \leftarrow T_p.\mathit{current\_set} \cup T_p.\mathit{local\_set}$
8: $T_p.\mathit{local\_set} \leftarrow \varnothing$

Data Structure for Access Sets 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 fixed-sized lists are allocated for its *current_set*, one to store reads, the other writes. Accesses are recorded sequentially into the two lists. If a task executes "too many" accesses, the lists may overflow, at which point we convert them to a hash table. The 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 into the bitmap. Hash tables are more space efficient, however, because they eliminate duplicate entries.

The *local_set* for each task is always allocated as a hash table. As a result, each intersection / merge in the algorithm is performed either over a hash table and a list, or over two hash tables. TARDIS iterates over the list or the smaller of the hash tables, searching for or inserting each entry in(to) the other set.

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 [24]) 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 full 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.

Advantages and Limitations Data race detection in TARDIS is sound (no false positives) and complete (no false negatives) because DPR tasks are always properly nested, access sets are maintained for the entirety of each parallel construct, and the sets of tasks T_i and T_j are intersected if and only if T_i and T_j are identified as independent in the source code.

As noted in Section 1, the principal alternative to log-based race detection is per-location metadata, also known as *shadow memory*. Like TARDIS, a shadow-memory–based race detector instruments each shared memory read and write (or at least each one that cannot be statically proven to be redundant). Rather than logging the access, however, it checks metadata associated with the accessed location to determine, on the fly, whether the current access is logically concurrent with any conflicting access that has already occurred in real time. It may also update the metadata so that any logically concurrent access that occurs later in real time can tell whether it conflicts with the current access. Details of the metadata organization, and the costs of the run-time checks, vary from system to system.

Like TARDIS, a shadow-memory based detector can be both sound and complete. There is reason, however, to hope that a log-based detector may be faster, at least for properly nested tasks. If we consider the average time to access and update shadow memory to be C_{sm}, a program with N accesses will incur checker overhead of NC_{sm}. TARDIS performs a smaller amount of (entirely local) work on each memory access or annotated method call, and postpones the detection of conflicts to the end of the current task. It merges accesses to the same memory location (and, in Sec. 3.2, calls to the same annotated method) on a task-by-task basis. End-of-task work is thus proportional to the footprint of the task, rather than the number of its dynamic accesses. If the total number of elements (fields and methods) accessed by a task is K, there will be K entries in its task history. Suppose the average time to insert an item in a task history is C_{th}, and the time to check an element for conflicts is C_{ck}. Then the total task-history–based checker overhead will be $NC_{th} + KC_{ck}$. In general, it seems reasonable to expect $N >> K$ and $C_{th} < C_{sm}$, which suggests that task-history–based checking may be significantly faster than shadow-memory–based checking. Worst case, if there are T tasks and M accessed elements in the program, and $K \approx M$, total space for task-history–based determinism checking will be $O(TM)$, versus $O(M)$ for shadow-memory–based checking. 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 task-history–based detection may be more space efficient than shadow memory as well.

For practical implementations, another potential advantage of log-based checking is the separation of logging and history analysis. In a correct program, history analysis (correctness verification) is independent of normal execution, and can be off-loaded to a decoupled checker task. For workloads that may periodically underutilize a parallel system, TARDIS has an "out-of-band" mode that performs history analysis in separate, dedicated threads. By running these threads on cores that would otherwise be underutilized, the overhead of analysis can be taken off the program's critical path. One of the benchmarks discussed in Section 4 (blackscholes) has a periodic utilization pattern that lends itself to such out-of-band analysis. In the others, additional cores are usually more profitably used for extra application threads.

The most significant potential disadvantage of log-based determinism checking is that it can only report when operations that cause nondeterminism *have already happened*. In a buggy program, TARDIS may not announce a conflict until after side effects induced by the bug have already been seen. These are "legitimate" side effects, in the sense that they stem from the bug, not from any artifact of the checker itself, but they have the potential to be confusing to the programmer. To mitigate their impact, we rely on the natural "sandboxing" of the virtual machine to ensure that a bug never compromises the integrity of the checker itself—and in particular never prevents it from reporting the original race. Reporting multiple conflicts that happened concurrently may also be a challenge, especially if erroneous behavior leads to cascading conflicts.

For the sake of efficiency, TARDIS by default reports only the non-array variable name and data address (object id and offset) of a conflict. To assist in debugging, TARDIS can also be run in an optional (slower) *detail mode* that provides full source-level conflict information. Further discussion appears in Section 3.3.

3.2 Extensions for Higher-level Operations

Reductions In a properly implemented reduction object, `push` calls can occur concurrently with one another (and are internally synchronized if necessary), but initialization and `get` calls cannot be concurrent with each other or with `push`. These rules closely mirror those for reads and writes, allowing us to reuse the basic TARDIS infrastructure: A `push` is treated as a pseudo-read on the reduction object; initialization or `get` is treated as a pseudo-write.

Atomic Commutative Operations Ordering requirements among higher-level methods, as declared by the programmer, are stored in per-class commutativity tables (*cTable*s). Like virtual method tables, cTables are linked into an inheritance chain. Sub-classes inherit commutative relationships by pointing to their parent classes, and the relationship between any two given methods can be determined by following the inheritance chain.

In order to check determinism for programs with AC ops, the concept of access sets is extended to *task histories*. TARDIS represents task T's history as an *atomic_set* of reads and writes in T that are performed by AC ops, a "*normal_set*" of reads and writes in T that are *not* performed by AC ops, and a *commutative method list* (*cml*) of the AC ops themselves.

For each task T, *current_set* and *local_set* are extended to be task histories (*curr_hist* and *local_hist*). All operations on the sets in Section 3.1 are directed to the corresponding *normal_set* fields. Both *normal_set* and *atomic_set* use the same data structure described in Section 3.1. Each *cml* is a map from *receiver* objects to the set of AC ops that were called.

On each AC op call, as shown in Algorithm 4, TARDIS records the receiver object and the method in the current task's *cml*. On each read and write, TARDIS then records access information in either *normal_set* or *atomic_set*, as indicated by the thread-indexed Boolean array *in_AC_op*. (For simplicity, we currently assume that AC ops do not nest.)

Algorithm 5 shows the revised task merging and conflict detection algorithm. Line 4 reports conflicts between two normal concurrent memory accesses. Lines 6 and 8 report conflicts between normal accesses and those performed inside an AC op. Even with correctly implemented AC ops, these may be caused by, for example, an access in another thread to an argument passed to an AC op. Line 10 reports conflicts among higher-level operations, as detected

Algorithm 4 On method call

Require: receiver object o of class C, task T, called method m, thread id th

1: **if** m is an annotated method of C **then**
2: $in\_AC\_op[th] \leftarrow$ true
3: $T.curr\_hist.cml[o]$.add(m)
4: // call m
5: $in\_AC\_op[th] \leftarrow$ false

Algorithm 5 On task merge

Require: child task T_c

1: $T_p \leftarrow T_c.parent$
2: **sync** on T_p
3: **if** $T_p.local\_hist.normal\_set \sqcap T_c.curr\_hist.normal\_set \neq \varnothing$ **then**
4: report a normal R/W conflict
5: **if** $T_p.local\_hist.atomic\_set \sqcap T_c.curr\_hist.normal\_set \neq \varnothing$ **then**
6: report an AC-op vs. non-AC op conflict
7: **if** $T_p.local\_hist.normal\_set \sqcap T_c.curr\_hist.atomic\_set \neq \varnothing$ **then**
8: report an AC-op vs. non-AC-op conflict
9: **if** !verify($T_p.local\_hist.cml$, $T_c.curr\_hist.cml$) **then**
10: report a commutativity conflict
11: $T_p.local\_hist \leftarrow T_p.local\_hist \Cup T_c.curr\_hist$
12: **if** T_c is the last task to join **then**
13: $T_p.curr\_hist \leftarrow T_p.curr\_hist \Cup T_p.local\_hist$
14: $T_p.local\_hist \leftarrow \varnothing$

Algorithm 6 verify function for commutativity checking

Require: commutative method lists L_a, L_b

1: **for** each object o in L_a **do**
2: **if** $o \in L_b$ **then**
3: **for** each method m_a in $L_a[o]$ **do**
4: **for** each method m_b in $L_b[o]$ **do**
5: **if** m_a does not commute with m_b **then**
6: **return** false
7: **return** true

Algorithm 7 Task history merge ($H_a \Cup H_b$)

Require: task histories H_a, H_b

1: $H.normal\_set \leftarrow H_a.normal\_set \cup H_b.normal\_set$
2: $H.atomic\_set \leftarrow H_a.atomic\_set \cup H_b.atomic\_set$
3: **return** H

by the verify function, shown in Algorithm 6. The $\Cup$ operation of Lines 11 and 13 is presented as Algorithm 7.

Isolated Futures DPR requires the operation performed in an isolated future to be *pure*. To verify pureness, TARDIS uses dynamic extent (scope) information maintained by the language virtual machine. In an isolated future, TARDIS checks the depth of scope for each access. Any attempt to access an object shallower than the starting scope of the future constitutes a pureness violation, as does I/O or any other operation that alters the state of the virtual machine. Since arguments are passed by deep copy, no conflicts can occur on these.

A future called within the dynamic extent of a concurrent task (e.g., a branch of a co-begin) is exempted from the usual read-write conflict checking. Concurrent tasks nested within a future, however, must be checked.

Pipelines As discussed in Section 2.1, pipeline objects in DPR introduce a different task execution pattern to accommodate loop-carried dependencies. Conflict detection, however, is not significantly altered. Given an n-stage pipeline, tasks issued in the same pipeline cycle will be treated as n concurrent tasks issued by a co-begin construct.

3.3 Engineering Issues

Memory location identification Everything in Ruby is an object—even built-in scalars like integers and floats. In TARDIS, each memory location is identified as $\langle id, field\_no \rangle$ and IDs are required for most objects for determinism checking.

JRuby internally assigns 64-bit IDs to objects lazily, on demand. A CAS operation is performed for each ID allocation to avoid a data race. We optimized this mechanism to reduce contention and avoid the CAS operation by allocating a block of IDs to each newly created thread. Each thread can then assign IDs to created objects as a purely local operation. Stack frames are treated as objects of which each local variable is a field.

Ruby permits new fields to be assigned to an object or class at run time, but these introduce no significant complications: each is assigned a new, unique (for its class) field number.

Integration with the virtual machine TARDIS needs to work with JRuby's built-in types. Objects of primitive types (e.g., scalars) are read-only, and require no instrumentation. Assignment to a local integer variable, for example, is logged as a change to a field of the stack frame, which contains a reference to an (immutable) integer. Because Ruby arrays are resizable vectors, the determinism checker has to accommodate dynamic changes in length, without announcing conflicts on the length field. The possibility of resizing also leads to high overhead in our reference implementation of shadow-memory–based race detection, as the checker needs to automatically extend the shadow memory as well, with attendant synchronization. For some other built-in collection data types, such as sets and hash maps, we provide standard annotations of method commutativity.

For performance reasons, JRuby code may either be directly interpreted or translated into Java bytecode before (ahead of time compilation) or during (just-in-time compilation) program execution. TARDIS instrumentation is performed at a relatively low level to make sure that method invocations and all shared reads and writes are logged in both interpretation and translation modes.

Recording and reporting conflict details Ideally, on detection of nondeterminism, TARDIS would report as much descriptive information as possible to the user. In practice, such reporting would require that we log information not strictly required for conflict detection. To minimize the cost of checking in the common (deterministic) case, TARDIS provides two modes of operation. In *default* mode, it reports the object id, field number, and non-array variable name involved in a conflict. In *detail* mode, it reports the symbolic name of the object and the source file and line number of the conflicting accesses. In practice, we would expect programmers to rely on default mode unless and until a conflict was reported, at which point they would re-run in detail mode. Note that on a given input a program is, by definition, deterministic up to the beginning of the first parallel construct in which a conflict arises. If a default-mode run elicits a conflict message, a subsequent detail-mode run on the same input can be expected (a bit more slowly) to report the same conflict, with more descriptive information. (Sometimes, especially when the input is associated with asynchronous events, exactly "the same input" may be hard to replay.)

Reducing detection overhead In practice, most reads and writes are performed to task-local variables (fields of the current task's stack frames). TARDIS refrains from logging these, since they cannot ever be shared.

In the pseudocode of Algorithms 3 and 5, the access sets of children are "pushed back" (merged in) to those of their parents. Once all children have completed, the merged set is needed only if the parent is itself a child in some other parallel construct. To economize on memory, TARDIS discards sets that have been pushed back into a sequential context (or an isolated future), once all the children have completed.

4. Performance Evaluation

Our evaluation aims to quantify the following:

- Scalability for DPR, both with and without dynamic determinism checking, on a parallel system
- Overhead for TARDIS to verify determinism dynamically
- Comparative cost of dynamic race detection for TARDIS and a state-of-the-art shadow-memory–based detector
- Extra cost for TARDIS to verify determinism on AC ops, and to provide detailed conflict information

4.1 Evaluation Setup

We evaluate DPR using 13 applications, from a variety of sources. Most were originally written in some other parallel language, which we hand-translated into DPR. During the translation, we used TARDIS to assist us in verifying determinism. Despite our familiarity with both parallel programming in general and DPR in particular, TARDIS still identified 5 nondeterminism conflicts.

Eight of our applications are from standard parallel benchmark suites: PARSEC [1], the Java Grande Forum (JGF) [34] and the Problem Based Benchmark Suite (PBBS) [33]. These are listed in Table 1. The other 5 are as follows:

Delaunay. A triangulation program working on a set of points in the plane. Originally written for a local class assignment, the code is a straightforward divide-and-conquer version of Dwyer's classic algorithm [9]. Our runs triangulate a field of 16,384 randomly chosen points. Tasks at each level are created with co-begin, and may add edges to, or remove them from, a concurrent list with annotated AC ops.

Chunky-png. A popular Ruby "gem" to convert between bitmap and PNG image formats, obtained from `rubygems.org/gems/chunky_png`. We parallelized the gem with DPR, and use it for bitmap encoding of the provided benchmark image. Our code uses a concurrent hash table with annotated AC ops to store encoding constraints.

WordWeight. This locally constructed program was inspired by the RemoveDup benchmark from PBBS. The original program used a hash table to remove duplicate keys given a user-defined total order on all values. Our DPR version works on articles. It takes a plain text file, assigns a "weight" to each word in the file, and then uses a concurrent hash table to find the maximum weight for each word.

GA. A genetic algorithm for use in a decompiler, this locally-constructed benchmark transforms irreducible regions in a control flow graph into high-level program constructs with a minimum number of `goto` statements. A gene representation of a candidate solution is an array of bits, where each bit represents whether a specific edge will be removed in the restructuring process. For each generation, a parallel iterator computes the fitness of all genes; each such calculation is time consuming. The evaluation function is annotated as an AC op.

Benchmark	Source	AC ops	Input set
Blackscholes	PARSEC	No	simlarge
Swaptions	PARSEC	No	simsmall
Streamcluster	PARSEC	No	simsmall
Series	JGF	No	A (small)
Crypt	JGF	No	A (small)
SparseMatMult	JGF	No	A (small)
SOR	JGF	No	A (small)
BFS	PBBS	Yes	rMat, n = 100,000

Table 1. Workloads from benchmark suites.

GA-Java. A variant of GA in which the fitness function is implemented as an external Java kernel. Each kernel is sequential, but calls from different DPR threads can execute in parallel.

The DPR virtual machine runs on a 64-bit OpenJDK at version 1.8.8. 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. The JRuby virtual machine will by default compile a function to Java byte code after 50 invocations (just-in-time). The Java VM then typically JITs the byte code. We kept this default behavior in our evaluation.

4.2 Determinism Checking for Benchmarks Without AC Ops

Seven of our benchmarks employ no AC ops, and can be checked for determinism with a conventional data race detector. For these we compare TARDIS, in both default and detail mode, to best-effort reimplementations of the Cilk Nondeterminator [11] and SPD3 [27] race detectors. Both prior systems are shadow-memory based, and do not record source-code–level information. To support reductions, pipelines and futures, we extended these systems with mechanisms similar to those discussed in Section 3. In our reimplementation of SPD3, the shadow area of each object is protected by a sequence lock to accommodate concurrent access and potential resizing.[3]

Results appear in Figure 3. We tested all workloads with 1, 2, 4, 8 and 16 threads. All data were collected with a maximum JVM heap size of 4 GB. 1024 slots were allocated for both reads and writes in the list-based representation of access sets. The baseline for speed measurement is the DPR version of each benchmark, running on a single thread with instrumentation turned off. Each data point represents the average of 5 runs. Since Nondeterminator is a sequential detector, it is reported at 1 thread only. Our results for SPD3 and Nondeterminator do not include the time spent allocating shadow memory for objects created in the initialization phase of the benchmarks.

For applications other than Streamcluster, in which most parallelism is very fine-grained, DPR achieves speedups of 4–6× on 16 threads. TARDIS outperforms SPD3 by substantial margins in three applications, and by modest amounts in another. It is slightly slower in the other three. The large wins in Blackscholes, Crypt, and SparseMatMult stem from tasks with 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 unrepeated, or rarely repeated,

[3] The newer, commutative-operation version of SPD3 [35] was not available in time to incorporate in our experiments.

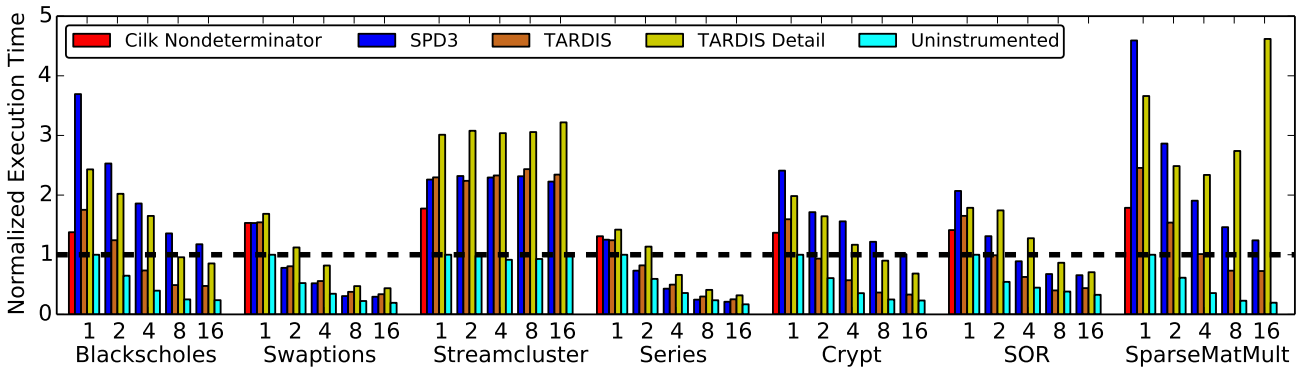

Figure 3. Normalized execution time (lower is better) of Nondeterminator, SPD3, TARDIS and uninstrumented DPR. Each workload is tested with 1, 2, 4, 8 and 16 threads. All speed results are normalized to the 1-thread uninstrumented case, which is also represented by the dashed line in the figure. Performance information for Nondeterminator is reported only at 1 thread, since it is a sequential algorithm.

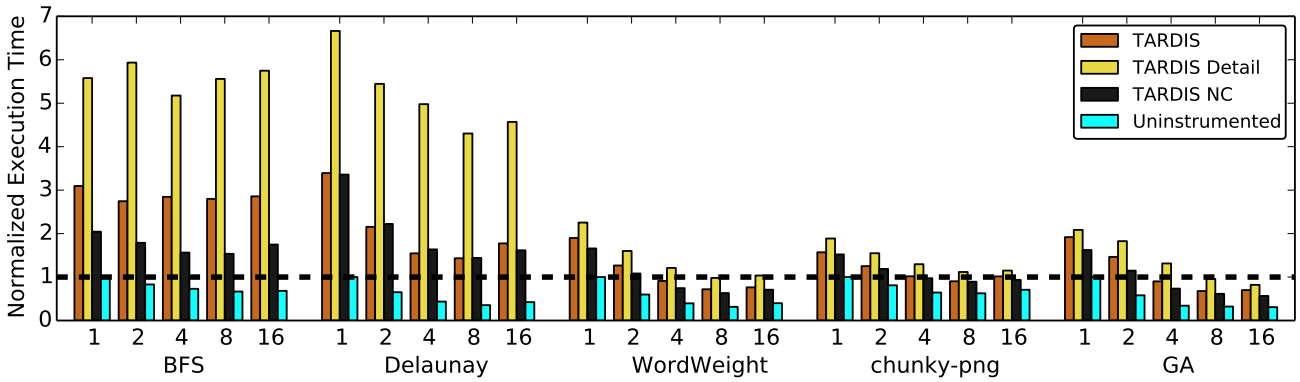

Figure 4. Normalized execution time of TARDIS (default mode), together with its detail mode, a mode with no AC operation checking (shown as "TARDIS NC"), and uninstrumented DPR. Each workload is tested with 1, 2, 4, 8 and 16 threads. All speed results are normalized to the 1-thread uninstrumented case, which is also represented by the dashed line in the figure.

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 instrumentation. Nondeterminator outperforms TARDIS and SPD3 at one thread in 6 out of 7 benchmarks, but is unable to scale up.

The detail mode of TARDIS is the slowest in all groups. Because of the extra information it must save, the working set of detail mode is substantially larger than that of default mode. In SparseMatMult, the blow-up is 23×. Cache misses caused by the large working set slow down the execution with 8 and 16 threads.

4.3 Determinism Checking for Benchmarks with AC ops

Our 5 remaining benchmarks (BFS, Delaunay, GA, WordWeight and Chunky-png) employ atomic commutative operations. For these, TARDIS is the only available precise determinism checker for DPR. To assess the cost of AC op checking, we run our benchmarks with TARDIS on and off, and also in an (incorrect) mode in which we fail to distinguish the AC ops. Loads and stores in those ops are then logged like any other accesses, leading to false reports of conflicts, but allowing us to measure the marginal overhead of AC op handling. Results appear in Figure 4.

Uninstrumented code (TARDIS off) achieves speedups of 0.5–4× across our benchmark set. With TARDIS enabled, WordWeight, chunky-png, and GA continue to achieve at least some speedup. In BFS and Delaunay, the extra space required by logs leads to significant cache pressure, with slowdowns of roughly 2–4× compared to the 1-thread uninstrumented case; in detail mode, this expands to 4–7×.

In comparison to no-AC mode, the checking of AC ops in BFS introduces about 40% overhead; for all other benchmarks the overhead is less than 10%. Profiling indicates that in BFS the access sets for AC ops and for normal accesses are almost equal in size, yielding an almost worst-case scenario for Algorithm 5, which implements set intersection by performing a lookup in the hash table of the larger set for each element of the smaller set. TARDIS actually outperforms its no-AC mode in some cases on Delaunay: moving some accesses to an AC set can reduce the cost of intersections, if it shrinks the smaller of an intersecting pair.

4.4 Scripting with an Accelerated Kernel

Scripting languages are typically chosen for their ease of use, rather than performance. After a DPR program has been built, it may be possible to accelerate it by replacing the most computationally in-

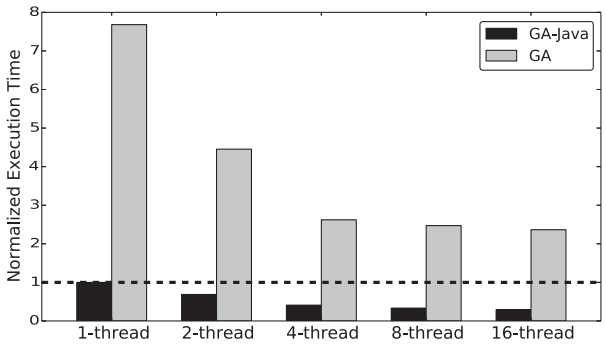

Figure 5. Normalized execution time of uninstrumented GA and GA-Java. Execution time is normalized with respect to the one thread execution time of GA-Java.

tensive parts with an external kernel, written in some performance-oriented language. In this case, it may still be attractive for the parallelism to be managed on the scripting side, with the external kernel kept sequential.

As an example of this style of programming, we created a version of the GA benchmark, GA-Java, that uses a pre-compiled Java kernel (easily called in JRuby) to perform the (sequential) evaluation function. As shown in Figure 5, GA-Java runs about 8 times as fast as the pure DPR version. It should be noted, of course, that TARDIS cannot verify determinism for external kernels, whose accesses are uninstrumented. Anecdotally, we found it easier and safer to translate an already-verified, sequential and independent kernel from DPR to Java than to develop a race-free Java kernel from scratch.

5. Conclusions and Future Work

In this paper, we discussed language design and run-time mechanisms for deterministic execution in a parallel scripting language. Specifically, we presented DPR, a parallel dialect of Ruby, and TARDIS, a dynamic determinism checker for our language. DPR's parallel constructs are useful tools in building or translating parallel workloads. In our JRuby-based implementation, DPR is able to achieve significant speedups on multicore machines for a variety of sample applications.

Tailored to DPR, TARDIS provides precise detection of races among both low-level reads and writes and high-level non-commutative operations. In comparable cases, TARDIS often outperforms existing state-of-the-art detectors—notably the SPD3 tool for Habañero Java. Relative to uninstrumented execution, TARDIS introduces (in default mode) a typical slowdown of approximately $2\times$ and a maximum of less than $4\times$. While this is probably still too slow to enable in production runs, it is eminently reasonable during testing and debugging. TARDIS also has a detail mode that helps programmers attribute detected conflicts to specific source-code constructs.

Topics for future work include:

Parallel scripting performance. Although DPR does display significant parallel speedup, it is neither as scalable nor (anywhere close to) as fast as a typical performance-oriented language. The raw speed of both sequential and parallel scripting can be expected to improve over time, though it is unlikely to ever rival that of compiled languages with static typing. Further study is needed to identify barriers to scaling in dynamic languages, and to determine which barriers may be addressable and which more fundamental.

Additional language constructs. As noted in Section 1, we are exploring such "structured" nondeterministic mechanisms as arbitrary choice and atomic asynchronous events. One could also consider additional deterministic constructs, or more refined definitions of commutativity. It is not yet clear at what point the set of parallel constructs would become "too messy" to fit well in a language that stresses convenience.

Commutativity checking. DPR currently treats commutativity annotations as axioms, so incorrect annotations can be a source of undetected errors. Though commutativity is undecidable in the general case, heuristic tools to test it could be very helpful.

Static analysis. Though many things cannot be checked until run time in a dynamic language, there are still opportunities for static optimization. We already identify a significant number of accesses that are guaranteed to be task-local, and need not be instrumented. Additional analysis could, for example, identify accesses whose instrumentation is provably redundant, and thus elidable.

Out-of-band processing of logs. Delayed, log-based detection of races raises the possibility that logs might be processed off the application's critical path. In our current implementation, this strategy is profitable only during I/O waits. With additional development, it might be possible on extra cores (for applications with poor scaling), or even in special hardware.

Acknowledgments

We thank Xi Wang, Lingxiang Xiang, Yang Tang and the anonymous reviewers for their feedback and suggestions on this work.

This work was conducted while Weixing Ji was a visiting scholar at the University of Rochester, supported by the Chinese Scholarship Council (award no. 2011603518). Li Lu and Michael Scott were supported in part by the U.S. National Science Foundation under grants CCR-0963759, CCF-1116055, CNS-1116109, CNS-1319417, and CCF-1337224. Weixing Ji was also supported in part by the National Science Foundation of China, grant no. 61300010.

References

[1] 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.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *J. of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[3] R. Bocchino Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *First Usenix Workshop on Hot Topics in Parallelism*, Berkeley, CA, Mar. 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. 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.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Berlin, Germany, June 2002.

[7] 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.

[8] C. Ding, B. Gernhart, P. Li, and M. Hertz. Safe parallel programming in an interpreted language. Technical Report #991, Computer Science Dept., Univ. of Rochester, Apr. 2014.

[9] R. A. Dwyer. A faster divide and conquer algorithm for constructing Delaunay triangulation. *Algorithmica*, 2, 1987.

[10] 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.

[11] 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.

[12] 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.

[13] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. River Trail: A path to parallelism in JavaScript. In *Object-Oriented Programming Systems, Langauges, and Applications (OOPSLA)*, Indianapolis, IN, Oct. 2013.

[14] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Salt Lake City, UT, Feb. 2008.

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

[16] W. Ji, L. Lu, and M. L. Scott. TARDIS: Task-level access race detection by intersecting sets. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, Houston, TX, Mar. 2013.

[17] JRuby: The Ruby programming language on the JVM. `jruby.org/`.

[18] B. Kasikci, C. Zamfir, and G. Candea. RaceMob: Crowdsourced data race detection. In *ACM Symp. on Operating Systems Principles (SOSP)*, Farminton, PA, Nov. 2013.

[19] D. Kim and M. C. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *32nd SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, San Jose, CA, June 2011.

[20] M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali. Exploiting the commutativity lattice. In *32nd SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, San Jose, CA, June 2011.

[21] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[22] 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.

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

[24] 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.

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

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

[27] 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.

[28] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Trans. on Programming Languages and Systems*, 19(6):942–991, Nov. 1997.

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

[30] 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.

[31] 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.

[32] E. Schonberg. On-the-fly detection of access anomalies. In *SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Portland, OR, June 1989. Retrospective appears in *ACM SIGPLAN Notices 39*:4 (Apr. 2004), pp. 313-314.

[33] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: The problem based benchmark suite. In *ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, Pittsburgh, PA, June 2012.

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

[35] E. Westbrook, R. Raman, J. Zhao, Z. Budimlić, and V. Sarkar. Dynamic determinism checking for structured parallelism. In *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, Salt Lake City, UT, Mar. 2014.

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

[37] 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.

[38] 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.