Designing for the Simple Case in a Parallel Scripting Language

by

Li Lu

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Professor Michael L. Scott

Department of Computer Science
Arts, Sciences and Engineering
Edmund A. Hajim School of Engineering and Applied Sciences

University of Rochester
Rochester, New York

2015
This dissertation is dedicated to my family.

My parents, Xiaoguang Lu and Cuiying Miao

and

My wife, Ran Tao
Biographical Sketch

The author was born in Neijiang, Sichuan, China. In 2005, he was admitted into Tsinghua University, Beijing, China. He graduated with a Bachelor of Engineering degree in Computer Science and Technology in 2009. In the same year, he entered the Doctor of Philosophy (Ph.D.) program at the University of Rochester, NY, USA. He pursued his research in parallel programming language semantics and implementations under the direction of Professor Michael L. Scott. During the Ph.D. program, the author received the Master of Science degree in Computer Science in 2011. In the summer of 2011, he worked as an intern researcher for Microsoft Research, Redmond, WA. In the summer of 2012, he worked as a software engineering intern for Google, Mountain View, CA.

The following papers were the result of work conducted during doctoral study.

Conference papers:


Other workshop papers:


The following open-source software package was the result of work conducted during doctoral study.

1. Rochester-BIT Implementation of DPR (RB-DPR):

https://github.com/rb-dpr/rb-dpr
Acknowledgments

I would like to thank my advisor, Professor Michael L. Scott, for his excellent guidance and support throughout my Ph.D. program. He offers me a tremendous amount of help with his knowledge and expertise in research, and always gives me support when I encounter any difficulties. He sets an example for me, not only in research, but also for life: I'm deeply impressed by his passion for work and life, by his attention to every detail, by his persistent chase for perfection, and by him feeling the responsibility of making the world a better place. So thank you very much Michael: This dissertation would just never exist without the help from you. It is a real honor to be a student of yours, and I'm very sure that this experience will definitely benefit my whole life.

I'm very thankful to my co-worker on the RB-DPR project, Professor Weixing Ji. Together, we built the implementation of DPR, and spent many days and nights discussing, polishing and debugging our artifact. Thank you for your hard and brilliant work, Weixing. It's a nice teamwork experience. Professors Chen Ding, Sandhya Dwarkadas, and Matthew Fluet served as my thesis committee. And I would like to thank them for their invaluable suggestions on my research. I feel honored to have them on my committee. I would also like to thank my mentors during the two internships, Edmund Nightingale from Microsoft Research and Xiao Zhang from Google, for their guidance during the unforgettable six months. I would like to thank my friends Xi Wang, Lingxiang Xiang, Yang Tang, and the
anonymous reviewers, for their feedback and suggestions on the papers published during my study.

I would offer my special thanks to my talented friends met in the Computer Science Department of the University of Rochester, in Tsinghua, and during the two internships. Together we had many valuable discussions and shared many great memories. Thanks to my wife Ran Tao for her years of support, and especially a beneficial discussion on an algorithm design issue in GMV: apparently this discussion is also a nice break for a chemical engineer.
Abstract

The fast development of parallel computer systems poses a challenge to programming language design and implementation. On the one hand, simple semantics are desirable; on the other hand, language implementations are expected to help programmers fully utilize complex parallel systems. Scripting languages are becoming popular because of their simplicity, but parallel performance has never been irrelevant for their design and implementation.

In this thesis, we propose the design and implementation of a deterministic parallel scripting language. Determinism is an appealing property for parallel programs, as it simplifies understanding, reasoning and debugging. It is particularly appealing in scripting languages, where ease of programming is a dominant design goal. We start by proposing a formal semantic framework for deterministic parallel programming. We suggest several history-based definitions of determinism, discuss some of their comparative advantages, prove containment relationships among them, and identify programming idioms that ensure them.

On top of the proposed formal semantic framework, we discuss the design of Deterministic Parallel Ruby (DPR), a parallel dialect of the Ruby programming language. DPR extends Ruby by adding simple deterministic parallel constructs while giving up on full generality (of conventional parallel programming models). We introduce the constructs of DPR, and a dynamic determinism checking mechanism (TARDIS) that verifies properties required for determinism. With our DPR
implementation, RB-DPR (RB stand for Rochester-Beijing Institute of Technology), 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 parallel scripting languages that are simple to use.
Contributors and Funding Sources

This work was supervised by a dissertation committee consisting of Professors Michael L. Scott, Chen Ding, and Sandhya Dwarkadas, of the Department of Computer Science, University of Rochester, and Professor Matthew Fluet of the Department of Computer Science, Rochester Institute of Technology. Professor Weixing Ji from Beijing Institute of Technology contributed to the idea of TARDIS, as did Professor Michael L. Scott. Professor Weixing Ji also collaborated in the implementation of RB-DPR, and some performance evaluation benchmarks. Professor Michael L. Scott provided Figure 3.1 and Professor Weixing Ji generated Figures 5.1, 5.2 and 5.3. All other work for the dissertation was completed by the student independently.

Graduate study was supported by the US National Science Foundation under awards numbers CCR-0963759, CCF-1116055, CNS-1116109, CNS-1319417, and CCF-1337224. The author has also been supported by multiple grants for conference travel from ACM SIGPLAN and the EC-funded Marie Curie Initial Training Network, TransForm.
# Table of Contents

**Biographical Sketch**

**Acknowledgments**

**Abstract**

**Contributors and Funding Sources**

**List of Tables**

**List of Figures**

## 1 Introduction

1.1 Background .................................................. 1

1.2 Motivation .................................................. 6

1.3 Organization of this Thesis ............................... 9

## 2 A Formal Semantic Framework for Determinism  

2.1 Introduction .................................................. 11

2.2 System Model ................................................. 16

2.3 Example Definitions of Equivalence ....................... 19
List of Tables

5.1 Workloads from benchmark suites . . . . . . . . . . . . . . . . . . . . 106
List of Figures

2.1 Relationships among source programs, abstract (program) executions and target executions. ........................................... 13

2.2 Containment relationships among definitions of determinism ................................................................. 23

2.3 Example program with a benign data race. ............................................................... 26

2.4 Example code fragment with parallel-for in a hypothetical language. ....................................................... 30

2.5 A sample CnC program’s flow chart. ........................................................................ 36

2.6 A sample epoch of a deterministic execution implementation. ..................................................... 39

3.1 Alternative models of parallelism .............................................................................. 46

3.2 A DPR code example for quicksort ........................................................................ 53

3.3 Code example for reductions ................................................................................. 57

3.4 Example code of a concurrent memoization table ........................................................................ 59

3.5 Relationships between annotated AC ops and non-AC ops ......................................................... 61

3.6 Example run of a 3-stage pipeline, on a sequence of 5 objects. ................................................................. 64

3.7 Code example of a 3-stage pipeline, on a sequence of 5 objects. ................................................................. 66

3.8 Task queue data structure for a thread ........................................................................ 69

3.9 Example code segment for StreamIt, setting up a trivial pipeline stage with one source and one sink. ........................................................................ 77
4.1 TARDIS’s hybrid representation of access sets. . . . . . . . . . . . 89

5.1 Normalized execution time (lower is better) of Nondeterminator,
SPD3, TARDIS and uninstrumented DPR. . . . . . . . . . . . . . 109

5.2 Normalized execution time of TARDIS (default mode), together
with its detail mode, no AC mode and uninstrumented DPR. . . 110

5.3 Normalized execution time of uninstrumented GA and GA-Java . 112
1 Introduction

Parallel systems are becoming ubiquitous in the modern computing world. Given such parallel systems, fully utilizing their computing power becomes an important goal. For programming language designers, the spread of parallel hardware platforms requires continuous improvements in existing parallel programming models: Programmers need more intuitive ways to express algorithms “in parallel”. They expect language implementations to make full use of the growing computing power, generating scalable performance on existing parallel systems, and on future systems. This thesis describes research that address some of these challenges for parallel scripting programming languages, from both language design and implementation perspectives.

This chapter provides background and an overview of this research.

1.1 Background

1.1.1 Parallel Programming Models

There are many factors to consider when designing a parallel programming language. In this thesis, we mainly focus on three critical ones among them: simplicity, generality, and performance. Generally speaking, programmers would expect
a parallel programming language to be simple to use, cover most significant use cases, and, on top of a parallel system, generate significant speed up.

Empirically, there is a trade-off between simplicity and generality. The comparison between shared memory programming models versus message-passing models is a representative example for such a trade-off. Message-passing programming models, such as MPI [MPI, 1994], can provide programmers with the ability to program on systems with multiple distributed memory spaces. On the other hand, shared memory programming models, like OpenMP [Dagum and Menon, 1998], can provide a programming interface for one globally visible shared memory. Message-passing programming models provide programmers the ability to directly model cross-memory-space communications. In contrast, it is much harder to explicitly model these communications with a shared-memory programming model. However, when talking about simplicity, things are different. With a message-passing model, performing operations on multiple memory spaces requires explicit data distribution and communication control. Though simplicity is a subjective concept for many programmers, it is arguably true that the message-passing model is less intuitive than the shared-memory model, where all memory locations are globally visible. There are also solutions on the middle ground, such as Partitioned Global Address Space (PGAS) languages. Co-Array Fortran [Numrich and Reid, 1998], UPC [Carlson et al., 1999], X10 [Charles et al., 2005], and Titanium [Aiken et al., 1998] are all PGAS languages. These languages work on distributed memory systems but offer the programmer a globally visible memory space. Having a globally visible memory space and an (almost) sequential programming model keeps the simplicity of the shared memory model, but PGAS languages can also work on distributed memory systems. However, due to the complexity of the underlying systems, PGAS languages are typically less intuitive than languages that only work for shared memory systems, like OpenMP.

When deciding between simplicity and generality, language designers need to
consider the targeted users and use cases for the programming language. Consider projects like the POSIX Pthreads (POSIX, 1995), Intel TBB (Threading Building Blocks) (Intel, 2011), and Intel CnC (Concurrent Collections) (Budimlić et al., 2010). Pthreads aims to provide a general and portable multi-thread programming interface for POSIX-conforming operating systems. Targeted users of Pthreads include system programmers, library builders, and other parallel programmers that are familiar with POSIX-conforming systems. The primary design goal of Pthreads leans toward the generality side, and thus its programming model is less intuitive than the other two models. Intel TBB provides commonly used parallel algorithms, data structures, and library routines. It uses implicit parallelism and provides pre-built parallel blocks, like concurrent data structures, synchronization constructs, and parallel algorithms to simplify parallel programming. TBB is less general than Pthreads: One can implement TBB with Pthreads, but it is not possible to implement Pthreads only with the constructs provided by TBB. The Intel CnC (Budimlić et al., 2010) aims to simplify parallel programming for non-professional programmers. CnC provides a data-flow style programming model for parallel algorithm designers and hides all parallel details from its programming interface, including thread creation, data distribution, and synchronization. For non-professional users, subjectively, CnC can provide a simpler programming interface than both Pthreads and TBB. However, CnC programs can only have a fixed execution pattern. Therefore, its expressiveness is the most limited among the three (but sufficient to cover most of its targeted use cases, like independent task parallelism). In fact, Intel’s own CnC implementation is based on TBB.

If simplicity is the primary goal of a parallel programming language, there are other applicable techniques that may help. Deterministic parallel programming, the main topic of this thesis, means the outcome of a parallel program is not affected by scheduling patterns. Determinism significantly simplifies understanding, debugging and reusing parallel programs. For synchronization, Transactional
Memory (TM) [Herlihy and Moss, 1993] uses the concept of “transactions” to guarantee atomicity. Transactions can replace conventional locks when performing synchronization, and can avoid deadlocks that happen in conventional, lock-based synchronization. We believe that the concept of “transaction”, or atomicity, can be a useful add-on feature for a simple parallel language.

In practice, language designers need to decide whether to support parallelism by adding libraries, or by redesigning the language. MPI, OpenMP, and Pthreads are all parallel libraries used with existing programming languages (Fortran/C/C++). Building a parallel library can add parallelism to an existing language while keeping the language semantics intact. However, the downside of this approach is that the “hosting” language of the parallel libraries may lack comprehensive support for parallelism. It has been argued that “threads cannot be implemented as a library” [Boehm, 2005]. In some languages, memory consistency models and/or language primitives are missing, making it hard to build and understand parallel programs. Modern languages like Java [Gosling et al., 2000], C# [Hejlsberg et al., 2003], C++11 [C++, 2011], and Scala [Odersky et al., 2004] all provide comprehensive language-level support for parallel programming.

In general, building a parallel language requires more effort than just building a parallel library. However, subjectively, parallel programming languages provide a cleaner and more comprehensive solution than parallel libraries. Hence, our work covered in this thesis focuses on designing and implementing a parallel language.

1.1.2 Scripting (Dynamic) Languages

Scripting languages are becoming more and more important in today’s programming world. People widely use scripting languages for web applications, coordination applications, software configuration and extension, and education. Most scripting languages emphasize flexibility, richness of expression, and ease of use. Toward these goals, today’s scripting languages often have some very appealing
features, including dynamic typing, both batch and interactive mode, sophisticated pattern matching and string manipulation, and high-level data structures. Popular scripting languages include Ruby (Ruby, 1995), Perl (Perl, 2002), or Python (Python, 1991) for general-purpose programming, PHP (PHP, 1995) or Javascript (Flanagan, 2002) for web programming, and R (R-Language, 1993) or Matlab (MATLAB, 1994) for mathematical or statistical computing.

The features of scripting languages are not for free: compared to conventional languages like C++ or Java, scripting languages have more significant run-time overheads. However, when deciding between performance and simplicity, most, if not all, scripting languages lean toward the simplicity side.

Favoring simplicity does not mean performance is irrelevant. The Ruby community is actively improving the MRI Ruby language virtual machine’s sequential performance, and websites and online blog posts demonstrate the improvements. Given this fact, it would be reasonable for people to expect scripting languages to generate significant speed up, or to “scale out”, on top of future parallel systems. However, performance on parallel systems has not yet become a major concern for scripting languages. Although some scripting languages do provide interfaces for creating threads in a program, both the programming interface and the performance require some further polish. Take the Ruby language as a representative example. On the programming interface side, Ruby provides threads and fibers to support parallelism, but these two constructs may appear to be “too hard” for general Ruby programmers. On the language implementation side, given a Ruby parallel program, current language virtual machine implementation may not be able to generate significant speedup on a parallel system. In fact, experience has shown that multi-thread Ruby programs may even slow down on the standard MRI Ruby 1.9 virtual machine.
1.2 Motivation

The goal of this research is to build a scripting language with “appropriate” parallel constructs. To design such a language, we may borrow concepts, idioms, and implementations from existing parallel programming models/languages. As discussed in Section 1.1.2 this is a practice applied by many existing scripting languages (like Ruby or Python). However, we may need more than this. In general, we need to comprehensively consider the tradeoffs among simplicity, generality, and language performance in a scripting language. All three factors relate to the semantics, programming interface, and runtime implementation (language virtual machine) of a parallel scripting language. More precisely, our goal is to design and implement a parallel scripting language that is simple and efficient, and still “feels” like a scripting language. We divide our goal into three subtasks, introduced in the following three subsections.

1.2.1 Formalize Determinism for Parallel Programming

As briefly noted in Section 1.1.1, determinism helps programmers to understand, build, analyze, and debug parallel programs. It has been proposed that parallel programming should be deterministic by default (Bocchino et al., 2009a). There are many existing projects to provide determinism for parallel programs, from a language perspective, a runtime/operating system perspective, or even an architecture perspective. However, the precise definitions of determinism for many existing projects are not the same; thus, this may create confusion on the semantic side of deterministic parallel language design. A formalized framework for deterministic parallel programming can be a helpful tool to resolve the confusion.

A formal framework may assist language design. Given a formalized framework of deterministic parallel programming, it is possible to discuss the relationship between commonly used programming idioms and possible definitions of
determinism. For language designers, a formalized framework makes it possible to reason about and prove the relationship between detailed language constructs and desired deterministic properties.

1.2.2 Design for the Simple Cases

The obvious objection to a parallel scripting language might be that parallel programming is too hard to “fit” with scripting. However, this is true only if the language attempts to cover the fully general case of parallel programming, and expose all its details to programmers. Parallel programming can also be easy, if the language directly supports the commonly used, important cases.

Although parallel programming is considered difficult for general cases, there are some simple and common cases of parallel programming that can be easy. Some programs are natively parallel, because the tasks they perform are completely independent. A simple example is the quicksort algorithm: after the partition process, the algorithm can concurrently work on two parts of the partitioned array, independently. Some programs may not be easily parallelized, as there are dependences within the program execution. Conversely, parts of the algorithm may be over-specified by its sequential version and may have a simpler parallel version. For example, if a linear search is applied to an unordered set, a conventional sequential algorithm may check all elements in this set one-by-one. The order to check each element does not matter, and the whole process can be parallelized.

In these cases, simple and deterministic parallel constructs are an ideal fit for parallel scripting: by giving up complete generality, the easy cases are very easy to express. Determinism, in turn, makes programs easy to understand.
**1.2.3 Enforce Determinism at Run Time**

Given a language design with deterministic parallel constructs, the most significant question left is how to build a language implementation according to this design. Specifically, the language implementation should be able to help programmers reason about whether all requirements of the deterministic parallel constructs are satisfied. In turn, satisfying all requirements of the parallel constructs guarantees determinism of the program execution.

Typical solutions for language implementations to reason about and guarantee determinism require either non-trivial changes to the language type and effect system (e.g., in Deterministic Parallel Java \cite{Bocchino2009b}), or a specialized execution pattern in the implementation (e.g., in CoreDet \cite{Bergan2010a}, RCDC \cite{Devietti2011}, and DMP \cite{Devietti2009}). The former approach significantly increases the difficulty of programming, and is therefore suboptimal for scripting languages. However, on the performance side, this approach allows the compiler to reason about the determinism of the program and therefore has less run-time overhead. The latter approach guarantees determinism below the language level, and chooses a single possibility from among the multiple possible executions permitted by the language. This approach often aims to provide determinism for “the general case”—for programs written in a wide range of languages, with various language constructs to express concurrency and synchronization. While this approach is certainly helpful for debugging, it does not help programmers to understand a specific program, and due to execution patterns, this type of approach brings in more significant run-time overhead.

An ideal determinism checker for a parallel scripting language may combine the advantages of both prior approaches: like the execution-pattern-based approaches, the determinism checker can avoid changing the type and effect system of the language, keeping a simple programming interface (and dynamic typing) for
scripting language programmers. Like the type and effect system based approach, a language-level determinism checker can help programmers to better understand a program. Meanwhile, since the determinism checker is specialized for a specific language, its run-time system can be tailored for all possible use cases of the language. Run-time overhead may be one major disadvantage of dynamic determinism checking and therefore may appear to be suboptimal for common, statically typed languages like Java. We believe this shortcoming is less significant under a scripting language background.

1.3 Organization of this Thesis

This thesis argues that parallel scripting programming languages can be simple and efficient. To support this statement, the design and implementation for a parallel dynamic language is given. Contributions of the design and implementation of this language include:

- A formal semantic framework for deterministic parallel programming, including formal definitions of determinism, of relationships among the definitions, and of potential programming idioms.

- A deterministic parallel dynamic language, Deterministic Parallel Ruby (DPR), with deterministic parallel constructs that have guaranteed determinism. The language focuses on the simple case for parallel scripting.

- An implementation of DPR called RB-DPR, with a run-time system, TARDIS, that can verify adherence to parallel construct rules and help in debugging DPR programs.

- Experiments, on the JRuby virtual machine, demonstrating that RB-DPR can achieve significant speedup on multicore machines for a variety of sample
applications. In particular, TARDIS introduces less overhead than existing state-of-the-art data-race detectors.

This thesis is organized as follows: In Chapter 2 we propose and discuss a formal semantics framework for deterministic parallel programming. In Chapter 3 we introduce the design and implementation of DPR. TARDIS’s design and implementation are discussed in Chapter 4. We report our evaluation results of DPR and TARDIS in Chapter 5. The thesis concludes, with possible future research directions, in Chapter 6.
2 A Formal Semantic Framework for Determinism

2.1 Introduction

Determinism, loosely defined, is increasingly touted as a way to simplify the design, verification, testing, and debugging of parallel programs—in effect, as a way to make it easier to understand what a parallel program does. Parallel functional languages have long enjoyed the benefits of determinism (Halstead, 1985). Recent workshops have brought together members of the architecture, programming languages, and systems communities to discuss determinism in more general languages and systems (Ceze and Adve, 2009; Adve et al., 2011; Berger, 2012; Bocchino and Devietti, 2013). Determinism has also featured prominently in recent workshops on pedagogy for concurrency (Midkiff et al., 2011; Shavit, 2009; Guy L Steele and Saraswat, 2009).

At the very least, determinism suggests that a given parallel program—like a sequential program under most semantic models—should always produce the same output when run with the same input. We believe, however, that it needs to mean more than this—that runs of a deterministic program on a given input should not only produce the same output: they should produce it in the same way. By analogy to automata theory, a deterministic Turing machine doesn’t just
compute a single-valued function: it takes a uniquely determined action at every step along the way.

For real-world parallel programs, computing “in the same way” may be defined in many ways. Depending on context, we may expect that repeated runs of a deterministic program will consume (more or less) the same amount of time and space; that they will display the same observable intermediate states to a debugger; that the behavior of distributed replicas will not diverge; that the number of code paths requiring separate testing will be linear in the number of threads (rather than exponential); or that the programmer will be able to straightforwardly predict the impact of source code changes on output or on time and space consumption.

*History-based semantics* has proven to be one of the most useful ways to model the behavior of parallel programs. Among other things, it has been used to explain the serializability of transactions [Papadimitriou 1979], the linearizability of concurrent data structures [Herlihy and Wing 1990], and the memory model that determines the values seen by reads in a language like Java [Manson et al. 2005] or C++ [Boehm and Adve 2008]. Memory models typically distinguish between *ordinary* and *synchronizing* accesses, and use these to build a cross-thread partial order among operations of the program as a whole. Recently Spear et al. (2008) and Dalessandro et al. (2010) have proposed that the various sorts of synchronizing accesses be unified under the single notion of an *atomic action*.

Informally, the parallel semantics of a given program on a given input is a set of *abstract executions*. Each execution comprises a set of *thread histories*, each of which in turn comprises a totally ordered sequence of *reads, writes*, and other *operations*—notably *external actions* like input and output. The history of a given thread is determined by the program text, the language’s (separately specified, typically operational) sequential semantics, the program’s input, and the values returned by reads (which may have been set by writes in other threads).
Figure 2.1: Relationships among source programs, abstract (program) executions and target executions. A language semantic maps source programs to program executions. A language implementation maps source programs to target executions.

An execution is said to be sequentially consistent if there exists a total order on reads and writes, consistent with program order in every thread, such that each read returns the value written by the most recent preceding write to the same location. Under relaxed memory models, a program is said to be data-race free if the model’s partial order covers all pairs of conflicting operations (read/write or write/write operations to the same memory location).

An implementation maps source programs to sets of low-level target executions on some real or virtual machine. The implementation is correct only if, for every target execution, there exists a corresponding abstract execution that performs the same external actions, in the same order. (The extent to which shorter sequences of target-level operations must correspond to operations of the abstract execution is related to, but separate from, the subject of this chapter; we do not address it further here.)

In the strictest sense of the word, a deterministic parallel program would be one
whose semantics, on any given input, consists of only a single abstract execution, to which any legal target execution would have to correspond. In practice, this definition may prove too restrictive. Suppose, for example, that I have chosen, as a programmer, to “roll my own” shared allocator for objects of some heavily used data type, and that I am willing to ignore the possibility of running out of memory. Suppose further that my allocator keeps free blocks on a simple lock-free stack. Because they access a common top-of-stack pointer, allocation and deallocation operations must synchronize with one another, and will thus be ordered in any given execution. Since I presumably don’t care what the order is, I may wish to allow arbitrary executions that differ only in the order realized, while still saying that my program is deterministic.

In general, we suggest, it makes sense to say that a program is deterministic if all of its abstract executions on a given input are equivalent in some well-defined sense. A language may be said to be deterministic if all its programs are deterministic. An implementation may be said to be deterministic (for a given, not-necessarily-deterministic language) if all the target executions of a given program on a given input correspond to abstract executions that are mutually equivalent. For all these purposes, the definition of determinism amounts to an equivalence relation on abstract executions.

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

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

In a similar vein, an implementation may be deterministic for only a subset of some standard programming language—i.e., for a smaller language. MIT’s Kendo system, for example, provides determinism for only those \( \langle \text{program, input} \rangle \) pairs that are data-race free—a property the authors call \textit{weak determinism} (Olszewski et al., 2009).

From an implementation perspective, history-based semantics differentiates between things that are required to be deterministic and things that an implementation might choose to make deterministic. This perspective draws a sharp distinction between projects like DPJ (Bocchino et al., 2009b), Prometheus (Allen et al., 2009), and CnC (Budimlić et al., 2010), which can be seen as constraining the set of abstract executions, and projects like Kendo, DMP (Devietti et al., 2009), CoreDet (Bergan et al., 2010a), and Grace (Berger et al., 2009), which can provide deterministic execution even for (some) \texttt{pthread}-ed programs in C. (Additional projects, such as Rerun (Hower and Hill, 2008), DeLorean (Montesinos et al., 2008), and DoublePlay (Veeraraghavan et al., 2011), are intended to provide deterministic \textit{replay} of a program whose initial run is more arbitrary.)

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

2.2 System Model

In a manner consistent with standard practice and with past work on atomicity-based semantics (Dalessandro et al. 2010), we define an execution of a program $P$, written in a language $L$, to be a 3-tuple $E_{P,L} : (OP, <_p, <_s)$, where $OP$ is a set of operations, and $<_p$ (program order) and $<_s$ (synchronization order) are irreflexive partial orders on $OP$. When we can do so without confusion, we omit $P$ and $L$ from our notation.

Each operation in $OP$ takes one of six forms: $(\text{read, name, val, tid, uid})$, $(\text{write, name, val, tid, uid})$, $(\text{input, val, tid, uid})$, $(\text{output, val, tid, uid})$, $(\text{begin\_atomic, tid, uid})$, or $(\text{end\_atomic, tid, uid})$. In each of these, tid identifies the executing thread. Uid is an arbitrary unique identifier; it serves to make every operation distinct and to allow the set $OP$ to contain multiple operations that are otherwise identical. In read and write operations, name identifies a program variable; in read, write, input, and output operations, val identifies a value read from a variable or from the program’s input, or written to a variable or to the program’s output. The domains from which thread ids, variable names, and values are chosen are defined by the semantics of $L$. These domains are assumed to be countable, but not necessarily finite.

Program order, $<_p$, is a union of disjoint total orders, one per thread. Specifically, if $o_1 = (\ldots, t_1, u_1)$ and $o_2 = (\ldots, t_2, u_2)$ are distinct operations of the same execution (i.e., $u_1 \neq u_2$), then $(t_1 = t_2) \rightarrow (o_1 <_p o_2 \lor o_2 <_p o_1)$ and $(t_1 \neq t_2) \rightarrow (o_1 \not<_p o_2 \land o_2 \not<_p o_1)$. (Here $\lor$ is exclusive or.)

For any given thread $t_i$, $OP|_{t_i}$ or $E|_{t_i}$ represents $t_i$’s thread history—its (totally ordered) sequence of operations. We use $OP|_s$ or $E|_s$ to represent an execution’s synchronization operations: begin\_atomic, end\_atomic, input, and output. We use
$OP|_e$ or $E|_e$ to represent the execution’s external operations: input and output. Clearly $OP|_e \subseteq OP|_s$. (We assume that I/O races are always unacceptable.)

Synchronization order, $<_s$, is a total order on $OP|_s$. It does not relate reads and writes, but it is consistent with program order. That is, for $o_1 = (\ldots, t_1, u_1)$ and $o_2 = (\ldots, t_2, u_2)$, $(o_1 <_s o_2 \land t_1 = t_2) \rightarrow (o_1 <_p o_2)$.

For convenience, we define $v_{in}$ and $v_{out}$, for a given execution $E$, to be the execution’s input and output vectors—the (possibly infinite) sequences of values contained, in order of $<_s$, in $E$’s input and output operations, respectively. For any given execution $E$, we use $ext(E)$ to represent the pair $\langle v_{in}, v_{out} \rangle$.

We use `begin_atomic` and `end_atomic` operations in our model to capture the synchronization operations of $L$, whatever those may be—thread fork and join, lock acquire and release, monitor entry and exit, volatile variable read and write, etc. For this reason, we require that `begin_atomic` and `end_atomic` operations appear in disjoint, unnested pairs, and never bracket input or output operations. That is, for every $b = (\text{begin\_atomic}, t, u_1)$ there exists an $e = (\text{end\_atomic}, t, u_2)$ such that $b <_s e$ and $\forall m \in OP|_s \setminus \{b, e\}$, $m <_s b \lor e <_s m$; likewise for every $e = (\text{end\_atomic}, t, u_2)$ there exists a $b = (\text{begin\_atomic}, t, u_1)$ such that $b <_s e$ and $\forall m \in OP|_s \setminus \{b, e\}$, $m <_s b \lor e <_s m$. We use $OP|_a$ or $E|_a$ to represent the execution’s atomic actions: the union of $E|_e$ and the set of minimal sequences of operations in each thread beginning with `begin\_atomic` and ending with `end\_atomic`.

Continuing with standard practice, we assume that the semantics of $L$ defines, for any given execution, a synchronizes-with order, $<_{sw}$, that is a subset of $<_s$—that is, a partial order on $OP|_s$. In a lock-based language, for example, the release method for lock $L$ might be modeled as $(\text{begin\_atomic}, t_1, u_1)$, $(\text{write}, L, 0, t_1, u_2)$, $(\text{end\_atomic}, t_1, u_3)$; an acquire might be $(\text{begin\_atomic}, t_2, u_4)$, $(\text{read}, L, 0, t_2, u_5)$.

\footnote{The release sequence must be bracketed with `begin\_atomic`...`end\_atomic`, even though there is only one operation inside, in order to induce cross-thread ordering.}
(write, L, 1, t2, u6), (end_atomic, t2, u7). If operation u3 precedes operation u4 in <s, we might require that it do so in <sw as well (since they operate on the same lock), but operations used to model methods of different locks might be unrelated by <sw.

Given <sw, we define happens-before order, <hb, to be the irreflexive transitive closure of <p and <sw. Finally, we assume that L defines, given <hb, a reads-sees-writes function W that specifies, for any given read operation r, the set of write operations \{w_i\} whose values r is permitted to return. In most languages, r will be allowed to see w if w is the most recent previous write to the same variable on some happens-before path. In some languages (e.g., Java), r may be allowed to see w if the two operations are incomparable under <hb. Languages may also differ as to whether atomic actions are strongly atomic—that is, whether nonatomic reads can see inside them (strongly atomic actions), or nonatomic writes be seen within them (Blundell et al., 2006) (Dalessandro et al., 2010, TR version appendix).

In any cut across <hb, we define the program state to be (1) the prefixes of v_in and v_out that have been input and output prior to the cut, and (2) the most recent values written to the program’s variables according to <hb. If a variable has not yet been written, its value is undefined (⊥); in a program with a data race, the most recent write may not be unique, in which case the variable’s value is indeterminate.

Two operations (read or write) conflict if they access the same variable and at least one of them writes it. An execution is data-race free if all conflicting operations are ordered by <hb.

In this chapter, we consider only well-formed executions. An execution E is well formed if and only if it satisfies the following three requirements.

Adherence to per-thread semantics: Given the code for thread t and the values returned by read and input operations, L’s (independently specified) sequential semantics determine the set of legal histories for t; E|_t must be
among them. Moreover, begin\_atomic and end\_atomic operations in $E|_t$ must occur in disjoint matched pairs, with only read or write operations between them (as ordered by $<_p$).

**Consistent ordering:** For all $t$, operations of $E|_t$ are totally ordered by $<_p$. Operations with different tids are unordered by $<_p$. $E|_s$ is totally ordered by $<_s$, which is consistent with $<_p$. Reads and writes do not participate in $<_s$. Paired begin\_atomic and end\_atomic operations are contiguous in $<_s$.

**Adherence to memory model:** All values read are permitted by $W$, the reads-see-writes function induced by $<_p$, $<_s$, $<_{sw}$, and $<_{hb}$, according to $L$’s semantics.

## 2.3 Example Definitions of Equivalence

In this section we suggest several possible definitions of equivalence for abstract executions. Two—Singleton and ExternalEvents—are intended to be extreme cases: the strictest and loosest definitions that strike us as plausible. Another—FinalState—is similar to ExternalEvents, restricted to programs that terminate. The other two—Dataflow and SyncOrder—are two of many possible in-between options.

**Singleton.** Executions $E_1 : (OP_1,<_p,<_s)$ and $E_2 : (OP_2,<_p,<_s)$ are said to be equivalent if and only if they differ only in the uids of their operations; that is, there exists a one-one mapping (bijection) between $OP_1$ and $OP_2$ that preserves $<_p$, $<_s$, and the content other than uid in every operation.

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

**Dataflow.** Executions $E_1 : (OP_1,<_p,<_s)$ and $E_2 : (OP_2,<_p,<_s)$ are said to be equivalent if and only if $\text{ext}(E_1) = \text{ext}(E_2)$ and there is a one-one mapping
between OP<sub>1</sub> and OP<sub>2</sub> that preserves (1) the content other than tid and uid in every operation, and (2) the reads-see-writes function W induced, under L’s semantics, by <<p, <s, <sw, and <hb.

Informally, Dataflow requires that reads see the same writes in both executions, and that the values in both reads and writes (including the input and output operations that “read” and “write” elements of v<sub>in</sub> and v<sub>out</sub>) be the same in both executions. Note that we do not require that the bijection preserve <p or <s, nor do we require that the executions be data-race free.

SyncOrder. Executions E<sub>1</sub>: (OP<sub>1</sub>, <<p<sub>1</sub>, <s<sub>1</sub>) and E<sub>2</sub>: (OP<sub>2</sub>, <<p<sub>2</sub>, <s<sub>2</sub>) are said to be equivalent if and only if there is a one-one mapping between E<sub>1</sub>|<a and E<sub>2</sub>|<a that preserves (1) <s, and (2) the content other than uid in every synchronization operation and in every read or write within an atomic action.

SyncOrder requires that there be a fixed pattern of synchronization among threads in E<sub>1</sub> and E<sub>2</sub>, with atomic actions reading and writing the same values in the same variables. Note that if executions are data-race free (something that SyncOrder does not require), then they are also sequentially consistent [Adve and Hill 1990], so E<sub>1</sub> ≡<sub>SynOrd</sub> E<sub>2</sub> ∧ E<sub>1</sub>, E<sub>2</sub> ∈ DRF → E<sub>1</sub> ≡<sub>Dataflow</sub> E<sub>2</sub>.

ExternalEvents. Executions E<sub>1</sub>: (OP<sub>1</sub>, <<p<sub>1</sub>, <s<sub>1</sub>) and E<sub>2</sub>: (OP<sub>2</sub>, <<p<sub>2</sub>, <s<sub>2</sub>) are said to be equivalent if and only if ext(E<sub>1</sub>) = ext(E<sub>2</sub)).

ExternalEvents is the most widely accepted language-level definition of determinism. It guarantees that abstract executions on the same input look “the same” from the perspective of the outside world.

FinalState. Executions E<sub>1</sub>: (OP<sub>1</sub>, <<p<sub>1</sub>, <s<sub>1</sub>) and E<sub>2</sub>: (OP<sub>2</sub>, <<p<sub>2</sub>, <s<sub>2</sub>) are said to be equivalent if and only if they both terminate and their program states at termination (values of variables and of v<sub>in</sub> and v<sub>out</sub>) are the same.
Like ExternalEvents, FinalState says nothing about how \( E_1 \) and \( E_2 \) compute. It requires only that final values be the same. Unlike ExternalEvents, FinalState requires agreement on variables other than output.

### 2.4 Discussion of Formal Definitions

**Singleton** is the strictest definition of equivalence, and thus of determinism. It is a common notion in the literature—corresponding, for example, to what Emrath and Padua called “internally determinate” ([Emrath and Padua](Emrath and Padua) 1988) and Netzer and Miller “internally deterministic” ([Netzer and Miller](Netzer and Miller) 1992). It requires a single execution for any given source program and input. Interestingly, while we have not insisted that such executions be sequentially consistent, they seem likely to be so in practice: a language that admits non-sequentially consistent executions (e.g., via data races) seems likely (unless it is designed in some highly artificial way) to admit multiple executions for some \( \langle \text{program, input} \rangle \) pairs.

By requiring abstract executions to be identical in every detail, **Singleton** rules out “benign” differences of any kind. It may therefore preclude a variety of language features and programming idioms that users might still like to think of as “deterministic.”

**Dataflow** relaxes **Singleton** by loosening the requirements on control flow. Equivalent executions must still have the same operation sets (ignoring \( \text{tid} \) and \( \text{uid} \)), but the synchronization and program orders can be different, so long as values flow from the same writes to the same reads. In the literature, **Dataflow** is essentially equivalent to Karp and Miller’s 1966 definition of “determinacy” ([Karp and Miller](Karp and Miller) 1966), which was based on a dataflow model of computation. Intuitively, **Dataflow** can be thought of as an attempt to accommodate programming languages and idioms in which the work of the program is fixed from run to run, but may be partitioned and allocated differently among the program’s threads.
SyncOrder also relaxes Singleton, but by admitting benign changes in data flow, rather than control flow. Specifically, SyncOrder requires equivalent executions to contain the exact same synchronization operations, executed by the same threads in the same order. It does not require that a read see the same write in both executions, but it does require that any disagreement have no effect on synchronization order (including output).

ExternalEvents is also a common notion in the literature. It corresponds to what Emrath and Padua called “externally determinate” (Emrath and Padua, 1988) and, more recently, to the working definition of determinism adopted by Bocchino et al. (Bocchino et al., 2009b). The appeal of the definition lies in its generality. If output is all one cares about, ExternalEvents affords the language designer and implementor maximum flexibility. From a practical perspective, knowing that a parallel program will always generate the same output from the same input, regardless of scheduling idiosyncrasies, is a major step forward from the status quo. For users with a strong interest in predictable performance and resource usage, debugability, and maintainability, however, ExternalEvents may not be enough.

FinalState is essentially a variant of ExternalEvents restricted to programs that terminate (and whose internal variables end up with the same values in every execution). It corresponds to what Netzer and Miller called “externally deterministic” (Netzer and Miller, 1992).

In the remainder of this section, we explore additional ramifications of our example definitions. We formalize containment properties: which definitions of equivalence, if they hold between a given pair of executions, imply which other definitions? Which definitions are incomparable? We also identify programming languages and idioms that illustrate these containments. Finally, we consider the practical issues of repetitive debugging and of deterministic implementation for nondeterministic languages.
Figure 2.2: Containment relationships among definitions of determinism, or, equivalently, abstract execution equivalence. Names of equivalence definitions correspond to ovals. Outlined numbers label the light, medium, and dark shaded regions. Bold letters show the locations of programming idioms.

2.4.1 Containment Properties

Figure 2.2 posits containment relationships among the definitions of determinism given in Section 2.3. The space as a whole is populated by sets \( \{X_i\} \) of executions of some given program on a given input, with some given semantics. If region \( S \) is contained in region \( L \), then all executions that are equivalent under definition \( S \) are equivalent under definition \( L \) as well; that is, \( S \) is a stricter and \( L \) a looser definition. (The regions can also be thought of as containing languages or executions: a language [execution] is in region \( R \) if for every program and input, all abstract executions generated by the language semantics [or corresponding to target executions generated by the implementation] are equivalent under definition \( R \).) We justify the illustrated relationships as follows.

**Theorem 2.4.1** Singleton is contained in Dataflow, SyncOrder, and External-Events.

**Proof**: Suppose \( E_1 \) and \( E_2 \) are arbitrary equivalent executions under Singleton. By definition, \( E_1 \) and \( E_2 \) are identical in every respect other than the uids of their
operations. None of the definitions of Dataflow, SyncOrder, or ExternalEvents speaks to uids. Each requires certain other portions of \( E_1 \) and \( E_2 \) (or entities derived from them) to be the same; Singleton trivially ensures this.

**Theorem 2.4.2** Singleton, Dataflow, SyncOrder, and FinalState are all contained in ExternalEvents.

*Proof:* For Singleton, this is proved in Theorem 2.4.1. For Dataflow, it follows from the definition: if \( E_1 \) and \( E_2 \) are Dataflow equivalent, then \( \text{ext}(E_1) = \text{ext}(E_2) \).

For SyncOrder, suppose \( E_1 : (\text{OP}_1, <_{p1}, <_{s1}) \) and \( E_2 : (\text{OP}_2, <_{p2}, <_{s2}) \) are arbitrary equivalent executions under SyncOrder. This means there is a bijection between \( \text{OP}_1 \) and \( \text{OP}_2 \) that preserves (among other things) both \( <_s \) and the content other than uids in each input and output operation. Since input and output operations are totally ordered by \( <_s \), and since \( v_{in} \) and \( v_{out} \) are defined to be the values in an execution’s input and output operations, in order of \( <_s \), we have \( \text{ext}(E_1) = \text{ext}(E_2) \).

For FinalState, suppose \( E_1 \) and \( E_2 \) are equivalent under FinalState. Then \( E_1 \) and \( E_2 \) both terminate, and with the same state. Their input and output vectors, included in their terminating states, must therefore be the same: \( \text{ext}(E_1) = \text{ext}(E_2) \).

**Theorem 2.4.3** There are sets of executions that are equivalent under Dataflow but not under SyncOrder.

*Proof:* This is the light gray region, labeled “1” in Figure 2.2. It corresponds to programs with benign synchronization races. Consider a program in which two threads each increment a variable under protection of a lock: \texttt{acquire(L); x++; release(L)}. Under plausible semantics, one possible execution looks as follows (ignoring uids), where \( <_p \) orders the operations of each thread as shown, and \( <_s \) orders the atomic actions of thread 1 before those of thread 2:
Call this execution $E_1$. Another execution (call it $E_2$) looks the same, except that the tids in various operations are reversed: thread 2 changes $x$ from 0 to 1; thread 1 changes it from 1 to 2. For *Dataflow*, the obvious bijection swaps the tids, and the executions are equivalent. For *SyncOrder*, there is clearly no bijection that preserves both synchronization order and the tids in each *begin atomic* and *end atomic* operation.

**Theorem 2.4.4** There are sets of executions that are equivalent under *SyncOrder* but not under *Dataflow*.

*Proof*: This is the medium gray region, labeled “2” in Figure 2.2. It corresponds to programs with benign data races. An example is shown in Figure 2.3. This is a racy program: neither the write nor the read of flag in $t_2$ is ordered by $<_{hb}$ with the write in $t_1$. Two abstract executions, $E_1$ and $E_2$ (not shown), may thus have different data flow: in $E_1$, the read of flag in $t_2$ returns the value 1, while in $E_2$ it returns the value 2. However, these two executions have the same synchronization order: in both, only the two outputs are ordered by $<_s$, and they are ordered the same in both executions. Thus $E_1 \equiv_{SyncOrder} E_2$ but $E_1 \not\equiv_{Dataflow} E_2$.

**Theorem 2.4.5** Singleton, *Dataflow*, and *SyncOrder* all have nontrivial intersections with *FinalState*.

*Proof*: *Singleton*, *Dataflow*, and *SyncOrder* clearly all contain sets of terminating executions that have the same final state. However, equivalent executions in *Singleton*, *Dataflow* or *SyncOrder* do not necessarily terminate. Suppose $E_1$ and
Initially flag == 0

t1: t2:
    flag = 1 flag = 2
    if (flag > 0)
        print "flag > 0"
        print "end"

Figure 2.3: Example program with a benign data race.

$E_2$ are arbitrary equivalent executions under Singleton, Dataflow or SyncOrder, and also under FinalState. We can make both executions nonterminating by adding an additional thread to each that executes an infinite but harmless loop (it might, for example, read an otherwise unused variable over and over). The modified executions will no longer be in FinalState since they do not terminate, but they will still be in Singleton, Dataflow, or SyncOrder, since the loop will neither race nor synchronize with any other part of the program.

Conversely, there are executions that have different data flows or synchronization orders, but terminate with the same state. Examples in FinalState $\cap$ (Dataflow $\setminus$ SyncOrder) and FinalState $\cap$ (SyncOrder $\setminus$ Dataflow) appear in the proofs of Theorems 2.4.3 and 2.4.4 respectively, and an example for FinalState $\setminus$ (SyncOrder $\cup$ Dataflow) is easy to construct (imagine, for example, a program that has chaotic data flow and synchronization, but eventually writes a zero to every program variable before terminating).

2.4.2 Programming Languages and Idioms

While equivalence relations and their relationships, seen from a theoretical perspective, may be interesting in their own right, they probably need to correspond to some intuitively appealing programming language or idiom in order to be of
practical interest. As illustrations, we describe five programming idioms in this section, corresponding to the dots labeled A, B, C, D, and E in Figure 2.2. Some of these programming idioms are supported by DPR, as discussed in detail in Chapter 3.

**Independent Split-Merge** (Point A $\in$ Singleton in Figure 2.2) Consider a language providing parallel iterators or *cobegin*, such as DPR, with the requirement (enforced through the type system or run-time checks) that concurrent tasks access disjoint sets of variables. If every task is modeled as a separate thread, then there will be no synchronization or data races, and the execution of a given program on a given input will be uniquely determined.

**Bag of Independent Tasks** (Point B $\in$ Dataflow $\setminus$ SyncOrder in Figure 2.2) Consider a programming idiom in which “worker” threads dynamically self-schedule independent tasks from a shared bag. The resulting executions will have isomorphic data flow (all that will vary is the tids in the corresponding reads and writes), but their synchronization orders will vary with the order in which they access the bag of tasks.

Significantly, this idiom remains in Dataflow $\setminus$ SyncOrder even if we require that tasks be added to the bag in groups, and all of them completed before any new tasks can be added. One might consider such a restricted model to be an alternative characterization of the Independent Split-Merge idiom, but we prefer to consider it a separate language—one in which the maximum degree of concurrency in the abstract execution is limited to the number of worker threads.

One might also expect that a program with deterministic sequential semantics, no data races, and no synchronization races would have only a single abstract execution for a given input—that is, that Dataflow $\cap$ SyncOrder would equal Singleton. We speculate, however, that there may be cases—e.g., uses of `rand()`—
that are easiest to model with more than one execution (i.e., with classically nondeterministic sequential semantics), but that we might still wish to think of as “deterministic parallel programming.” We have left a region in Figure 2.2 (the dark gray area labeled “3”) to suggest this possibility.

**Parallel Iterator with Reduction**  (Point C ∈ ExternalEvents \ (Dataflow ∪ SyncOrder) in Figure 2.2) Consider a language with explicit support for reduction by a commutative, associative function (the Reduction class in DPR is an example, discussed in Section 3.3). The order in which such a function is applied to a set of operands need not be fixed, leading to executions with different synchronization orders and data flows, but only a single result. It seems plausible that we might wish to call programs in such a language “deterministic.”

**Parallel Atomic Commutative Methods**  (Point D ∈ ExternalEvents \ (Dataflow ∪ SyncOrder) in Figure 2.2) In the split-merge and bag-of-tasks idioms above, we required that parallel tasks be independent. We may relax this requirement by allowing tasks to call methods of some shared object O, so long as the calls are atomic and (semantically) commutative. The memory allocator mentioned in Section 2.1 is an example of this idiom, as long as we ignore the possibility of running out of memory. Another example would be a memoization table that caches outputs of some expensive function. DPR supports this idiom by directly supporting annotations for Atomic Commutative (AC) operations.

If a program contains atomic, commutative method calls in otherwise independent tasks, the synchronization order for these calls may be different in different runs of the program on the same input. Data flow may also be different, because the internal state of the shared object may change with the synchronization order. Even the final state may be different, since commutativity is defined at a level of abstraction above that of individual variables. A given finite sequence of calls
is guaranteed to lead to the same output, however, regardless of permutation, because the calls are atomic and commutative.

**Chaotic Relaxation** (Point E ∈ SyncOrder \ Dataflow in Figure 2.2)

Imagine a language that is specially designed for programs of this kind. Programmers can specify an $\epsilon$ for the convergence condition, then design an iterative algorithm for an array of data. Different executions may have different data flows, because the program is full of data races. For chaotic relaxation, however, these data races do not change the limit toward which the computation converges. If final results are rounded to a level of significance determined by $\epsilon$ before being output, the results will be deterministic despite the uncertainty of data flow. And as long as the output operations (which constitute the only synchronization in the program) are strictly ordered, the program will be deterministic according to SyncOrder.

2.4.3 Repetitive Debugging

One of the principal goals of deterministic parallel programming is to facilitate repetitive debugging. The definitions in Section 2.3 vary significantly in the extent to which they achieve this goal.

In a Singleton system, a debugger that works at the level of abstract executions will be guaranteed, at any breakpoint, to see a state that corresponds to some consistent cut across the happens-before order of the single execution. This guarantee facilitates repetitive debugging, though it may not make it trivial: a breakpoint in one thread may participate in an arbitrary number of cross-thread consistent cuts; global state is not uniquely determined by the state of a single thread. If we allow all other threads to continue running, however, until they wait for a stopped thread or hit a breakpoint of their own, then global state will be
Figure 2.4: Example code fragment with parallel-for in a hypothetical language.

deterministic. Moreover (assuming a relatively fine-grain correspondence between target and abstract executions), monitored variables’ values will change deterministically, since Singleton requires all runs of a program on a given input to correspond to the same abstract execution. This should simplify both debugging and program understanding.

Under Dataflow, monitored variables will still change values deterministically, but two executions may not reach the same global state when a breakpoint is triggered, even if threads are allowed to “coast to a stop.” A program state encountered in one execution may never arise in an equivalent execution.

Consider the code fragment shown in Figure 2.4 (written in a hypothetical language). Assume that $f()$ is known to be a pure function, and that the code fragment is embedded in a program that creates two worker threads for the purpose of executing parallel iterators. In one plausible semantics, the elements of a parallel iteration space are placed in a synchronous queue, from which workers dequeue them atomically.

Even in this trivial example, there are four possible executions, in which dequeue operations in threads 0 and 1, respectively, return \{0, ⊥\} and \{1, ⊥\}, \{1, ⊥\} and \{0, ⊥\}, \{0, 1, ⊥\} and {⊥}, or {⊥} and \{0, 1, ⊥\}. These executions will contain exactly the same operations, except for thread ids. They will have different program and synchronization orders. Dataflow will say they are equivalent; Singleton will say they are not. If we insist that our programming model be determinis-
tic, Dataflow will clearly afford the programmer significantly greater expressive power. On the other hand, a breakpoint inserted at the call to f() in thread 0 may see very different global states in different executions; this could cause significant confusion.

Like Dataflow, SyncOrder fails to guarantee deterministic global state at breakpoints, but we hypothesize that the variability will be significantly milder in practice: benign data flow changes, which do not impact synchronization or program output, seem much less potentially disruptive than benign synchronization races, which can change the allocation of work among threads.

ExternalEvents and FinalState, for their part, offer significant flexibility to the language designer and implementor, but with potentially arbitrary differences in internal behavior across program runs. This would seem to make them problematic for repetitive debugging.

2.4.4 Deterministic Implementations

Generally speaking, given a deterministic parallel programming language, it should be straightforward to construct an implementation that achieves most of the concurrency that a programmer might expect on a given machine. This expectation is essentially an issue of liveness, and may be difficult to formalize, but the intuition is clear: if the language is capable of expressing only deterministic programs, then an implementation that captures the concurrency explicit in such programs will remain deterministic. In Independent Split-Merge programs, as supported by DPR (discussed in Chapter 3), an implementation is assured that synchronization (\( <_s \)) edges enter a task only at the beginning, and leave it only at the end, so scheduling decisions within a split-merge group can never violate happens-before.

The more interesting question is: in a language that admits nonequivalent abstract executions, how hard is it likely to be (how much run-time cost are we
likely to incur) to construct an implementation that achieves a high degree of concurrency (scalability) while still guaranteeing that all target executions will correspond to equivalent abstract executions? Here the answer may depend on just how much nondeterminism the language itself allows. As noted in Section 2.1, Kendo ([Olszewski et al. 2009]) provides determinism (of roughly the `SyncOrder` variety) only for programs written in the data-race-free subset of C. Specifically, it resolves each synchronization race deterministically, given that deterministic resolution of prior synchronization races and the lack of data races uniquely determines program order in each thread, up to the next synchronization operation. While still too slow for production use (reported overheads are on the order of 1.6×), this is fast enough for convenient repetitive debugging.

For programs with data races, there is no known way to achieve any of our definitions of deterministic implementation without special-purpose hardware or very high worst-case overhead (one can, of course, serialize the execution—we count that as “very high overhead”). CoreDet ([Bergan et al. 2010a]), dOS ([Bergan et al. 2010b]), and Determinator ([Aviram et al. 2010]) all achieve roughly Singleton semantics on conventional hardware. Unfortunately, all impose common-case overheads of roughly 10×, making them unsuitable for production use and undesirable for debugging. Recent work on the DoublePlay system ([Veeraraghavan et al. 2011]) suggests that it may be possible to execute arbitrary programs deterministically while limiting overhead to a relatively modest amount (comparable to that of Kendo) for executions whose behavior does not depend on data races.

2.5 Related Work

This section surveys existing techniques for deterministic parallelism and related topics. We mentioned some related work in the previous sections of this chapter; here we provide additional detail. Generally speaking, there are two types of
deterministic parallel systems: one aims to eliminate nondeterminism at the pro-
gram level, providing a deterministic parallel language; the other aims to eliminate
nondeterminism at the target execution level, providing a deterministic parallel
implementation.

2.5.1 Deterministic Programming Models

A deterministic language can guarantee all programs written in this language will
be executed deterministically. However, to make this guarantee, there may be
some compromise in the power of the language. This means that there may be
some programs that cannot be expressed. A race-free language may be an ini-
tial step to exclude nondeterministic program executions. Data races introduce
nondeterminism, and are considered as bugs by some mainstream languages like
C++. But from a language design perspective, this is not enough. Due to the lim-
itation of a deterministic parallel language’s constructs, even some data-race-free
programs are not able to be expressed in a deterministic parallel language. Pro-
grams with synchronization races, such as those associated with lock acquisition,
are often not allowed by some deterministic parallel languages.

The key to guaranteeing determinism at the language level is reasoning about
independence for concurrent operations. Some programming models, like Intel
CnC, limit expressiveness, so that it is not possible to create concurrent tasks
that are not disjoint in a correct program. Some, like Deterministic Parallel Java,
prove task independence statically, by requesting concurrent effects to be explic-
itly declared in the program. There are multiple other works that are intended
to support deterministic parallel programming, at different levels. For verifiers
and checkers, Burnim and Sen (2009) propose an assertion-based system, to help
programmers verify the state of parallel programs. SingleTrack (Sadowski et al.,
2009) uses a dynamic analyzer to verify non-interference data communications (ac-
cesses). Some previous work on “safe” parallelism is also related to deterministic
parallel programming. “Safety” in these projects generally means equivalence to a program’s sequential execution. Safe futures \cite{Welc2005} uses the future construct to support concurrent programming. Its run-time system dynamically executes concurrent tasks in parallel and guarantees safety by instrumenting reads and writes to detect concurrent conflict operations. The BOP \cite{Ding2007} system can speculatively parallelize program executions given programmer hints. The runtime then detects conflicting operations happening in concurrent tasks, and, once such conflicts are detected, falls back to a sequential execution of the same program. So instead of guaranteeing determinism, the BOP system sticks to the sequential semantics of the program. The Grace \cite{Berger2009} system can eliminate concurrency errors in fork-join parallel programs. Its implementation is similar to BOP’s. One significant difference between these two projects is that Grace directly works on parallel programs, while BOP works on sequential programs with programmer’s hints.

**Deterministic Parallel Java (DPJ)** DPJ is one representative deterministic parallel programming language. It’s based on Java, but introduces several concepts to guarantee determinism. Specifically, it uses a specially designed type and effect system to guarantee race freedom. Unless explicitly specified, both data races and synchronization races are completely eliminated in DPJ.

DPJ uses the concept of *region* to reason about memory. In theory, regions partition the heap. Each region has a specified name, defined in the source code. As regions are symbols for heap partitions, it is possible to have *disjoint constraints* on multiple regions, which means that regions with different names should comprise completely disjoint parts of the heap. The programmer has to assign regions for all an object’s fields. Since the language is heavily based on Java, data operations are operations on object fields. Therefore, each write operation is associated with a region. Regions may be nested because of object composition. DPJ uses
Region Path Lists, a list style representation, to describe nested regions.

In DPJ, each method is associated with an effects list, defined by the programmer. An effects list for a method specifies all “effects” this method has. Effects are modeled as reads and/or writes to a region. For example, a method that reads two data fields’ values, performs a multiplication, and then updates a third field can be expressed as “reads $R1, R2$; writes $D$”, where $R1, R2$ are the regions of the two data fields, while $D$ is the region of the updated field. There are some special cases for such effect summary lists. For example, constructors do not need such lists, because the language will prohibit any data access to an object until its constructor has been executed. Another special case accommodates conventional sequential Java codes: an omitted effect list (for non-constructor methods) means a method may affect the whole heap.

To express parallelism, the programmer can use two constructs predefined by DPJ: cobegin and foreach. As in most other parallel programming models, statements in the same cobegin section will be executed in parallel, while a foreach section will execute different iterations of a for loop in parallel.

The DPJ system performs verification statically. First, the compiler checks whether each effect list matches the actual effect of that method. Then, it checks whether the parallel part of each method is safe, based on the effects and region disjoint constraints. This whole process is intra-procedural. It can prove the determinism of legal DPJ programs.

Intel Concurrent Collections (CnC)  Intel proposes CnC as a new parallel programming model. CnC is a very different programming model than DPJ, but is very similar to some data flow programming languages. Programmers can specify both data and control dependences in a CnC program, with its predefined “collections”. These “collections” may carry some concurrent semantics and can be executed in parallel, in a deterministic way.
Figure 2.5: A sample CnC program's flow chart. An initial image set and tags are specified as input. The program then uses 3 classifiers to detect faces in the image set.

CnC provides three kinds of collections: step collections for computational steps; item collections for data items; and tag collections for control tags. A CnC program consists of a data flow graph that has multiple collections, whose relationships are specified by their interconnections. Such relationships are dependences of a parallel program. In the programmer-defined data flow graph, step collections define algorithms for a parallel program: for example, an image processing function. Item collections model the data of a program, such as an image in an image processing program. Concurrency of a CnC program comes from the assumption that elements in the same input data collection are different and the tasks to process them are independent. Tag collections store control information and decide which element in a data collection should be processed by “prescribing” a step collection. An instance in a step collection will be executed if and only if its tag (unique id) is in the tag collection that prescribes the step collection, or that step collection is not prescribed by any tag collection. This is useful when step collections only need to process the data that satisfy certain conditions. For example, an image processing function only needs to process the image that has a face in it. Both item collections and tag collections can be accessed and edited
by step instances at run time, by the CnC predefined put and get methods.

CnC eliminates nondeterminism by introducing concurrency only among independent objects. In Budimlić et al. (2010), it is proved that featherweight CnC, a simplified version of CnC, is deterministic. The definition of determinism used in this paper is similar to FinalState, introduced in Section 2.3. This conclusion is based on several assumptions: first, there’s no interactive IO during the execution. The whole program should be in a “function” style, which means the program should accept input only at the very beginning and generate output at the very end of program execution. Second, the featherweight CnC implementation should output all its data collections and all elements in those collections in a fixed order. This will guarantee that a CnC program will surely generate the same output, rather than only the same result, on the same input. Third, to reach the final state, featherweight CnC programs should always terminate.

2.5.2 Deterministic Implementations

A deterministic implementation can guarantee that any program run on it, no matter which kind of language it’s written in, will run deterministically. This works well to simplify repetitive debugging, and seems to be a general solution to deterministic parallel programming. But in practice, it also has some limitations. One of the most significant limitations is that the deterministic implementation does not really help the programmer to eliminate any nondeterminism (and potentially bugs) hiding in the source code: it just hides those problems at run time. Some problems may be hidden temporarily because they do not happen in a particular history. When the program needs some future changes or optimizations—or most trivially, when the input changes—these hidden problems may appear again. So deterministic implementations do not really help the programmer to understand parallel programs: they just help the programmer to deterministically avoid or trigger potential troubles on a certain input.
Put another way, when doing deterministic execution, if the desired output never occurs after several attempts, the programmer may be puzzled: is this because the program will never generate the expected output, or because of a data race in the program, or because of a synchronization race? From this perspective, deterministic implementation is not good enough to help programmers fully understand or verify the correctness of parallel programs.

That said, deterministic implementations are still helpful for reasoning and debugging parallel programs. In general, deterministic implementations guarantee determinism by fixed execution patterns. Trivially, a deterministic implementation could serialize the entire program execution, the same way, every time it runs. Implementation-wise, this means executing the whole program in a sequential way and completely losing concurrency. Most existing projects overcome this limitation by using an “epoch based” execution pattern. In this execution pattern, parallel program executions are sliced deterministically into several epochs. The run-time system iteratively executes two modes, a parallel mode, and a serial mode. In parallel mode, the runtime issues independent concurrent epochs in parallel and performs a barrier on all issued epochs before transferring into serial mode. A thread reaches the barrier by finishing its execution, trying to execute a synchronization operation, or trying to communicate with other threads. In serial mode, threads are scheduled in a serial fashion and the order is deterministic. In this way, thread interference can be resolved deterministically.

This execution pattern generates deterministic program executions, with no assumptions on race freedom of the program. Any existing data race or synchronization race would force the epoch to reach a barrier, and perform potentially non-commutative operations under the serial mode. However, identifying these operations may be expensive. Some projects introduce specialized hardware to overcome run-time overheads.
Figure 2.6: A sample epoch of a deterministic execution implementation. The 3 threads perform operations concurrently in parallel mode, and then enter serial mode by either performing synchronization operations or finishing the epoch. In serial mode, synchronization operations are performed in a deterministic order. In this example, $T_1$ and $T_3$ finish their parallel mode by performing synchronization operations, and $T_2$ finishes by using up its whole epoch.

**CoreDet** ([Bergan et al., 2010a](#)) CoreDet is a pure software implementation for deterministic parallel programming. It uses a combination of compiler and run-time techniques. The runtime has two key components: an ownership-based inter-thread communication detector and a versioned memory buffer.

CoreDet utilizes the execution pattern discussed above. When threads do not communicate with the others, the CoreDet run-time system will execute them in parallel. When there are some threads that “request” a communication, the run-time will serialize them. During execution, for each epoch, each thread is granted a quantum (a permission to execute a fixed number of instructions). They start under the *parallel mode*, where all threads run in parallel. Once a thread runs out its quantum or requests communication, its parallel execution mode immediately terminates. When all threads’ parallel execution modes terminate, the whole system will start serial mode, where all communications are handled by the system, in a deterministic way. Detecting communications is important in Core-
Det, because this will affect the overhead and scalability of the implementation. CoreDet works on shared memory systems, and the “communications” performed in CoreDet are memory references to the same location among different threads. There are three ways to detect communications in a thread: DMP-O, DMP-B and DMP-PB. DMP-O uses an ownership-based detector, with low overhead but poor scalability. DMP-B uses a versioned memory buffer to “track” communications, and has higher overheads but better scalability. However, the tradeoff is that DMP-B only guarantees a relaxed memory ordering. Combining DMP-B and DMP-O together, DMP-PB is a hybrid implementation that is in a middle ground with both overhead and scalability.

To accommodate the run-time system of CoreDet, some compiler support is needed. The main support methods include instrumenting every statement to do counting (for quanta), and every data operation to guarantee determinism.

CoreDet can provide deterministic executions for programs from different languages. This is a very appealing feature. However, nothing comes for free. In the original paper, it is shown that CoreDet has good scalability, but has 100%–1000% overheads compare to non-deterministic implementations. Run-time overhead is the main problem for such deterministic implementations.

**RCDC** ([Devietti et al., 2011](#)) RCDC is a deterministic parallel implementation proposed by the Sampa group in University of Washington. It is follow-up work to the Deterministic Multi-Processing (DMP) project. Both of them distinguish from the CoreDet project because they have hardware-level support for determinism, while CoreDet is completely in software. RCDC is different from DMP because it supports a more relaxed memory model: DMP uses Total Store Order (TSO) ([Weaver and Gremond, 1994](#)) but RCDC uses Data-Race-Free (DRF) ([Adve and Hill, 1990](#)). Implementing a relaxed memory model will have less overhead for the system, because there are fewer orders to guarantee. At the
same time, DRF still supports data-race-free based memory models, such as Java and C++. The hardware layer of RCDC only needs to provide two mechanisms: a software-controlled store buffer for isolated thread execution in parallel mode, and an instruction counter for splitting thread executions into epochs. In simulation results, RCDC’s relative overhead, when being compared to nondeterministic executions, is at most 60%, which is significantly better than pure software implementations, such as CoreDet.

**Determinator (Aviram et al., 2010)** Determinator is a proof of concept operating system designed by DeDiS group at Yale University. It is an operating system level deterministic implementation. It has a microkernel that allows no data sharing, and an upper level programming interface that provides a private workspace model for parallel programming. Its kernel will also deny the user code’s direct access to hardware resources that may cause nondeterministic results, such as real-time clocks, cycle counters, and writable shared memory. On top of that kernel, Determinator gives each thread its own copy of the shared memory. Each thread operates on its own copy, and then “checks in” its changes at a synchronization point. Read-write races are eliminated because each read can only see the causally prior write, and write-write races can be detected by the system and then reported to the user.

**Kendo (Olszewski et al., 2009)** The Kendo project takes a different approach to deterministic implementations. Rather than guaranteeing deterministic execution for all parallel programs on the platform, it guarantees this only for a subset. More precisely, Kendo guarantees deterministic execution of data-race-free programs. The limited application range seems to be a significant disadvantage of the project, but in practice this is still valuable. Normally, to achieve deterministic execution, both data races and synchronization races need to be resolved deterministically. Resolving synchronization races generally introduces much less
overhead. To resolve synchronization races, the run-time system needs to instrument all synchronization operations. However, to resolve data races, the run-time system has to instrument every read and write operation which may cause a race. Therefore, by limiting the range of applicable programs, Kendo introduces significantly less run-time overhead than other pure software solutions (the average for run-time overhead on a variety of benchmarks is only 16%). Since data races are considered as bugs in more and more programming languages, it might be true that the limitation of Kendo would be less significant with future parallel programs.

The idea of Kendo is simple: it enforces deterministic orders on all synchronization operations (mostly lock acquires and releases). Specifically, it employs deterministic lock acquisition and release algorithms, utilizing logical times. In this way, synchronization races can be resolved in a deterministic way. (And, as expected, programs with data races may not be deterministic under Kendo.)

### 2.5.3 Deterministic Replay

To make understanding, reasoning and debugging of parallel programs easier, there is another type of solution: deterministic replay. Deterministic replay is a different technique to both deterministic programming languages and deterministic implementations (we use the term “deterministic parallel programming” when referring to both techniques). Deterministic replay uses some special mechanism to record a parallel execution, and then replays this execution history later, to guarantee all executions have the same output. The recording mechanism can either be hardware based or software based; either will generate an event history log when the program on a given input is executed the first time. From the second time on, on the same input, the system will switch to replay mode, and execute the record, rather than the actual binary, to generate exactly the same event history.
From some perspectives, deterministic replay is similar to deterministic parallel programming, as both of them are trying to make parallel program executions to be deterministic. Intuitively, deterministic replay makes it possible to ensure that future executions will be the same as some chosen past execution, but deterministic parallel programming is focused on generating the same executions on the same input all the time. Deterministic replay is based on user choices: the user can choose which specific execution to record, and when to replay the recorded execution. But until there is an explicit operation to initiate the replay mode, the program is not deterministic. This is not the case with deterministic parallel programming, where the system is always deterministic: a system only generates equivalent target executions on the same input, either because all program executions are equivalent (deterministic language), or because all races are resolved in the same way (deterministic implementation). Due to this difference, deterministic parallel programming and deterministic replay focus on different implementation techniques: deterministic parallel programming focuses on how to guarantee that all target executions are equivalent, while deterministic replay focuses on how to record and replay a parallel program execution with low overhead.

Example systems for deterministic replay include Instant Replay (LeBlanc and Mellor-Crummey, 1987), DeLorean (Montesinos et al., 2008), Rerun (Hover and Hill, 2008), and DoublePlay (Veeraraghavan et al., 2011). One possible approach to implement a deterministic replay system is to store the total order of the whole program execution. However, this approach is not scalable and not storage-efficient for logs. The Instant Replay system relaxed the total order by keeping a separate log for each shared object (synchronization object). In this way, run-time overhead can be reduced. It is also possible for a replay system to use the two-stage epoch-based execution pattern to execute a parallel program, in a manner similar to CoreDet. Instead of simply keeping the total order, the runtime will make a log for each unrecorded execution, and replay the whole program execution with the
log. A possible improvement over this, implemented by DoublePlay, is to launch a parallel program execution in the background while sequentially simulating the program in the foreground, and using the parallel (probably faster) execution to speculate the log.
3 Deterministic Parallel Ruby

Dynamically typed, “scripting” languages, such as Python, R, and Ruby, are increasingly popular, in domains ranging from e-commerce and web services to education and even scientific research. Scripting languages typically run on a custom virtual machine, and are designed for ease of programming, even at the expense of raw performance. This is not to say, however, that performance is irrelevant. All else being equal, programmers will almost always choose faster execution. In practice, they may use a scripting language for initial software development, and then rewrite key components in a statically typed language like C or Java when better performance is required—as Twitter did, for example, when it switched from Ruby to Java in 2011 (Metz 2013). Better performance in the original scripting language might delay the need for such rewrites.

With the rise of multi-core and many-core machines, it seems inevitable that programmers will want to use their extra cores to increase system performance, even in a scripting language. One possible path to such “scale-out” is to fork additional processes (each with its own virtual machine), connect those processes with sockets, and communicate via messages (Fig. 3.1(a)). While this approach is common in scripting languages today, several decades of experience suggest that many programmers would rather write multithreaded programs within a single address space, and communicate via shared memory (Figs. 3.1(b) and (c)). From
Figure 3.1: Alternative models of parallelism: (a) separate processes; (b) general-purpose threads and shared memory; (c) independent split-merge tasks.

From a conceptual standpoint, shared memory supports “implicit” communication: a writing thread doesn’t need to say what part of a shared structure to “send” to a reading thread, and the reader doesn’t need to say when to “receive” it. From a performance standpoint, the cost of thread creation and management within an address space is generally quite a bit less than that of separate virtual machines.

Unfortunately, fully general shared-memory threads still suffer from the problem of nondeterministic interleaving: if we think of the operations in a thread as if they were playing cards, then the overall program is a merging—a shuffling together—of the decks that represent the threads, and program behavior may depend on the order of cards in the merge. Consider, for example, the program in Figure 3.1(b). The picture suggests that the right-hand thread has written variable X before the left-hand thread reads X. If the left thread runs a little faster, though, and the right thread runs a little slower (perhaps because the operating system’s scheduler has made different decisions about what to run on which core when), then the read may happen before the write, and the left thread may see the old value of X. The same issue arises with Y. In effect, the threads are racing toward their conflicting accesses, and program behavior depends on
the outcome of the race. The need to reason about interleavings makes parallel programming—at least in the general case—dramatically more complicated than sequential programming. Arguably, this complexity is simply incompatible with the ease of use that is scripting’s raison d’être.

We believe that the answer lies in deterministic execution. Determinism means that whenever two parts of a program may execute in parallel, the order in which their operations interleave will have no observable impact; that is, program behavior will be determined solely by the source code and the input—never by relative execution speed. The simplest way to ensure determinism is to insist that concurrent tasks (potentially parallel parts of the program’s execution) are mutually independent. In Figure 3.1(c), for example, execution has split into a pair of tasks. The left-hand task is allowed to read a variable that was written before the split, and the right-hand task is allowed to write a variable that will be read after the tasks have merged again, but neither task is permitted to write a variable that the other reads or writes.

As described in the previous chapter, some researchers have developed languages in which determinism is enforced at compile time, using extensions to the type system [Bocchino et al. 2009b]. In the simple case, any given program object is owned by at most one task at any given point in the source code; any attempt to access the object in another task results in a compile-time error. Unfortunately, the ownership type rules are quite complex, and entirely incompatible with scripting, where variables are typically undeclared, and not given types until run time.

Our work takes a different approach. We interpret every split and merge as an assertion on the programmer’s part that the subtasks are independent—much as the use of an arithmetic operator constitutes an assertion that the operands are numbers, or an array subscripting operation constitutes an assertion that the subscript is in bounds. And just as these other assertions are verified by the
virtual machine at run time, so too must we verify independence.

Our Deterministic Parallel Ruby (DPR) adopts a handful of well-known concurrent constructs from the literature and leverages Ruby’s support for functional programming (lambda expressions in particular) to embed these constructs cleanly in the language syntax. For any given program and input, the constructs dictate a fixed graph of task splits and merges. To support a more abstract notion of “independence”, DPR allows the programmer to describe the commutativity of higher-level operations. Meanwhile, DPR also has parallel constructs specially designed for some common cases in parallel programming.

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.

3.1 Language Overview

The main design goal of DPR is to demonstrate that a parallel scripting language can still preserve the simplicity of a scripting language. DPR makes use of Ruby’s syntax and dynamic features to keep a consistent programming style with the original language. Relatively (and subjectively) simple changes to the Ruby source code can allow programmers to make good use of multicore hardware.
3.1.1 Implicit Parallelism

Parallel programming is often considered difficult by many programmers. Many languages require the programmer to understand and arrange too many low-level details, clutter the syntax, mix design details from different levels of abstraction, and combine algorithm design with performance tuning on specific platforms.

Like some existing parallel programming models (OpenMP, Cilk, and DPJ, for example), DPR uses implicit parallelism to decouple algorithm design and parallel infrastructure design. DPR provides language constructs that can create concurrent (unordered) tasks, and can capture simple cases in parallel programming. DPR constructs can be nested. Depending on the implementation, concurrent tasks may be executed in parallel, though this is not guaranteed. The implementation is responsible for low-level parallelism details, including thread scheduling, task division and distribution, and load balancing. This separates the problem of designing parallel algorithms and specifying parallel details. Cross platform porting for DPR programs should be easy, and platform-specific optimizations are possible.

Standard Ruby includes traditional shared-memory threads, of the sort depicted in Figure 3.1(b). These should not be used directly by the DPR programmer. They can be used to implement DPR’s concurrent constructs, allowing us to run on any existing Ruby virtual machine. Alternatively, DPR constructs can be implemented inside the virtual machine, in order to improve their performance. Either way, a modest number of existing Ruby threads (typically, one per core) operate as workers, repeatedly executing the smaller, more lightweight tasks.

3.1.2 Determinism

In DPR, different constructs satisfy different definitions of determinism. Among the possible definitions of determinism proposed in Chapter 2, Singleton requires
that a program always generate exactly the same execution given the same input, while the most relaxed definition, *ExternalEvents*, requires only the same output (external events) on the same input. For DPR programs, the type of determinism guaranteed is always bounded by the “same output on the same input” (*ExternalEvents*) definition. As discussed in Section 2.4, satisfying different definitions of determinism means guaranteeing different sets of properties. Given the same input, depending on the DPR constructs used, program behavior varies from the same result (from the output) to fully equivalent execution histories. We discuss determinism for each parallel construct in detail in the following sections of this chapter.

While determinism is desirable, it comes at a price. For the sake of determinism, DPR sacrifices a certain degree of generality: there are parallel programs one simply cannot write in DPR, some are even deterministic under other programming models. The class of programs that remains is very large, however, and provides ample opportunity to make good use of multicore hardware. Moreover, all correct DPR programs are deterministic by default.

The most fundamental restriction made by DPR is on the possible ways to create parallelism. In DPR, programmers are not supposed (and not allowed) to program the parallel execution details, such as threads. The type of concurrency is deliberately limited by properly-nested, independent, fork-join tasks. We observe that in practice properly-nested independent parallel constructs can significantly simplify programming. As we shall see in the following chapter, they also facilitate dynamic determinism enforcement. In addition, even given only proper nesting, programmers can (easily) express a range of commonly used parallel algorithms.

To guarantee determinism, each parallel construct has its own special usage restrictions. Some of these restrictions may be broken by careless programming and cause subtle bugs (nondeterminism). Ideally, a DPR runtime should catch bugs that directly break the determinism of a program, at least in debug runs.
However, some run-time verifications are hard to implement (some are even not computable). In DPR, we identify some usage restrictions as requirements. It is required that a DPR implementation should check if each parallel construct’s requirement is satisfied, at least in debug mode. In order to guarantee correctness of a DPR program, the programmer should also satisfy all assumptions on the programmer’s side, but these assumptions may not be checked during run time. For language designers, deciding the set of requirements is a subjective task. In DPR, we mainly consider the following factors:

1. The requirement is a necessary condition to guarantee determinism for a DPR program.

2. The likelihood for programmers to accidentally break the requirement is (subjectively) high.

3. It is possible to dynamically detect any violations to the requirement, and the complexity to implement such a detector is (subjectively) acceptable.

### 3.2 Independent Tasks

The most basic parallel constructs in DPR are co-begin and unordered iterators, which create properly nested “split-merge” tasks. Programmers can use the co statement for co-begin, and the .all method for unordered iterators. We interpret the use of these constructs as an assertion on the programmer’s part that the tasks are independent—an assertion which, as we discuss in the following chapter, needs to be checked 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 execution histories that have different external behaviors (Chapter 2).
Determinism  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 interleaving: a program based on only co and .all satisfies the strong Singleton definition of determinism.

3.2.1 Co-begin

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.

Usage  The co-begin construct can be used as follows:

\[
\text{co} \rightarrow\{ \text{task}_1 \}, \rightarrow\{ \text{task}_2 \}, \ldots
\]

The list of \(\lambda\)-closures represents concurrent tasks.

Requirements  Each lambda closure should be independent to other concurrent tasks declared in the same co statement. Note that the DPR language runtime may execute any combination of concurrent tasks in parallel, according to its underlying implementation.

Assumptions on the Programmer’s Side  None.

Example  The co statement can be used to express “task parallelism” such as that arising in divide-and-conquer algorithms that work on disjoint memory locations.
def qsort(a)
    return a if a.length <= 1
    m = a.length/2
    pivot = a[m]
    left = mid = right = nil
    co ->{ left = qsort(a.map{|v| v < pivot?v:nil}.compact) },
         ->{ mid = a.map{|v| v == pivot?v:nil}.compact },
         ->{ right = qsort(a.map{|v| v > pivot?v:nil}.compact) }
    return left + mid + right
end

Figure 3.2: A DPR code example for quicksort

For example, Figure 3.2 shows a quicksort algorithm. (This algorithm makes extra copies of the input array. With a bit more code, an in-place version is certainly possible.)

3.2.2 Unordered iterators

The .all method is intended for data parallelism. It is reminiscent of the built-in .each, but as with co, .all 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.

Usage An unordered iterator, invoked by the all method, can be used similarly to Ruby’s each method, followed by a block that would be applied to each element of the collection (represented by x in the following example):
\r.all \{|x| \ op\} \\

**Requirements** Each iteration is a task, and needs to be independent of all other iterations. The DPR runtime may group, reorder, and execute any combinations of tasks in parallel.

**Assumptions on the Programmer’s Side** None.

**Example** Unordered iterators can be used to speed up independent loop iterations in a DPR program. In our experience of building DPR benchmark programs, unordered iterator is the most commonly used parallel construct for data-parallel workloads.

As a simple example, a graphics application might use .all to adjust the color of each pixel in an image:

```
myPic.all \{|p| adjustColor(p) \}
```

### 3.2.3 Design Notes

Independent tasks set up the basic concurrent structure of DPR programs. For implementation, since both constructs can be used in nested fashion, it is important to decide the granularity of parallelization. For unordered iterators, in cases where each iteration runs in a comparable or even shorter time than scheduling this iteration, the run-time system may need to group a certain number of tasks together to amortize the overhead of task scheduling. Explicitly requiring programmers to manually group tasks is generally undesirable, since this will couple DPR programs to parallel execution details. To solve this problem, DPR provides a .setGrain method for each array/range object to allow programmers to suggest the best granularity to group iterations. DPR implementations may utilize this information for a better scheduling policy.
3.3 Atomic Commutative Operations

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

3.3.1 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.

Usage A reduction object can be created by:

```
objName = Reduction.new(Reduction::TYPE)
```

TYPE can be selected from:

¹We 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.
ADD Compute the sum of all pushed elements.

MULTIPLY Compute the product of all pushed elements.

MAX Compute the maximum value of all pushed elements.

MIN Compute the minimum value of all pushed elements.

Concurrent tasks can add new elements into the reduction by calling the push method, and then obtain the final result of the reduction by calling its get method.

Requirements The get call must not be concurrent to any push calls.

Assumptions on the Programmer’s Side The operation performed must be commutative and associative semantically. Note that built-in operations may not be commutative or associative with certain types (floating point operations may not be associative when precision requirements are high). It is the programmer’s responsibility to make sure the operations performed are actually commutative and associative.

Determinism 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 ExternalEvents definition of determinism: synchronization order and data flow 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.

Example In Figure 3.3 we create a simple accumulator and accumulate values concurrently. The output of this program should be $0 + 1 + 2 + 3 + 4 = 10$. 
r = Reduction.new(Reduction::ADD)
(0..5).all { |i|
  r.push(i)
}
puts r.get()

Figure 3.3: Code example for reductions

3.3.2 User-defined Atomic Commutative Operations (AC Ops)

DPR allows the programmer to identify Atomic Commutative operations (AC ops) 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.

Usage One can annotate a set of methods to be AC ops by using the `setACOps` meta-programming method in a class body:

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

It is assumed that all AC ops commute with one another unless explicitly indicated otherwise with method `setNoCommute`:

```
setNoCommute :m0, :m1
```

If a class is inherited by some other class, its AC ops, as well as all their relationships, will be carried to the subclass. A subclass can always mark special cases by `setNoCommute` calls. However, in a subclass, once an inherited method is overridden, AC op annotation on the overridden method, together with all annotated relationships, will be reset.
Requirements Each AC op must be independent of any other concurrent method calls that are not AC ops.

Assumptions on the Programmer’s Side 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 is provided by DPR. Alternatively, the designer of a class with AC methods may choose a more highly concurrent implementation—e.g., one based on fine-grain locking. For simplicity, we currently require that AC ops do not nest.

It is the programmer’s responsibility to guarantee that all commutativity annotations are correct. 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. There are several testing strategies that could, in future work, be used to increase confidence in the annotations (Rinard and Diniz 1997; Kulkarni et al. 2011; Kim and Rinard 2011).

Determinism With AC ops, a DPR program no longer guarantees Singleton. However, determinism is still satisfied in a relaxed fashion, where the program will always generate the same output on the same input (ExternalEvents). The data flow or synchronization behavior of a program may change across different executions even given the same input.

Example We expect most AC op annotations to happen inside library routines, specifying high-level commutativity information for reusable objects.

Consider a concurrent memoization table that stores the values computed by some expensive function $f$. We show an example in Figure 3.4. The class definition might look something like in this figure. The lookUpOrCompute method will
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
  setACOps :lookUpOrCompute
end

Figure 3.4: Example code of a concurrent memoization table

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.

**Design Note**  We design AC ops for the nature of scripting languages, where high-level data structures (like variable-length arrays, maps, and sets) with high level semantics are heavily used. By raising up the levels of abstractions, we can capture conflicts on method level, rather than fine-grained read/write level (including object metadata). We considered several alternative designs for the programming interface of AC ops. In our initial design, we let the programmer specify the commutativity relationship of all pairs of methods in a class. Although
this simplifies language implementation work, in practice it is undesirable. For a class with \( n \) methods, this requires \( \binom{n}{2} \) annotations. Practically, in Ruby, even the most basic class, `RubyObject`, has more than 20 built-in methods. It will cost significant effort for programmers to specify commutative relationships for all methods in a class, and repeat this for each class that has AC ops. In our experience, we noticed that most AC ops are commutative with each other. We can use this fact to effectively reduce the burden of AC op annotations. In DPR, AC op annotations divide all methods of a class into two sets: one set with and only the AC ops \( (S_{acop}) \) and one with all other (un-annotated, normal) methods \( (S_{normal}) \). By DPR’s design, all methods in \( S_{acop} \) are commutative with each other by default, unless explicitly exempted by the programmer. Intuitively, annotating a method \( m \) to be an AC op means replacing \( S_{acop} \) and \( S_{normal} \) by \( S'_{acop} = S_{acop} \cup \{m\} \) and \( S'_{normal} = S_{normal} \setminus \{m\} \), respectively.

We also need to consider the relationships between AC ops and un-annotated methods. Trivially, an AC op is commutative with a normal method if they are independent. DPR does not allow the case where an AC op and a concurrent normal method are not independent. This is because no assumption should be made on the atomicity of normal methods, neither on the commutativity of the two methods. To guarantee determinism, the language has to make the most conservative assumption.

The semantics of AC op annotations need to be specially designed for the object-oriented features of Ruby. With inheritance, a subclass obtains all non-private methods of its superclass. Intuitively, annotated relationships on inherited methods should be preserved in subclasses. On method overriding, we decided that the newly defined method should not preserve the predefined commutativity relationship. The rationale is as follows: There is no guarantee on the overriding method to behave like the overridden one. Therefore, keeping the relationships unchanged after override may introduce bugs. For example, AC op `foo` in class
Figure 3.5: Relationships between $S_{acop}$ and $S_{normal}$. The two blocks represent the same object, with methods called in two concurrent tasks.

A may commute with another AC op \texttt{bar} in the same class, because they operate on completely disjoint memory locations. However, when class A is inherited by a subclass B, in its overriding method \texttt{foo}, some new feature is added, and now it no longer commutes with \texttt{bar} because both methods operate on the same field of the class, without proper synchronization. Programmers may easily overlook this kind of error, and introduce nondeterminism during execution. To prevent similar cases, DPR requires re-annotations for any override methods. Note that DPR (and Ruby) does not support method overloading.

Another challenge in designing AC ops for a scripting language comes from the language’s dynamic features. In Ruby, a class can be reopened and its methods may be changed dynamically. Extended methods in a reopened class are by default non-AC ops, and only become AC ops if explicitly annotated. For all dynamically changed methods, similar to the rationale for method overriding, commutative relationships are no longer preserved.
3.4 Specialization

In addition to independent tasks, reductions, and AC ops, DPR also provides pipelines and isolated futures. Isolated futures have their origin in the concept of futures in Multilisp [Halstead, 1985], but assume that the call is independent of the rest of the program; in DPR, we require that the function be pure. Pipelines allow programmers to model parallel streaming programs—some of which may have loop-carried dependences.

3.4.1 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. 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.

Usage An isolated future can be created by:

\[
\text{f} = \text{Future.new}(\rightarrow\{|x_1, x_2, \ldots| \text{op}\}, \text{arg}_1, \text{arg}_2, \ldots)
\]

When the result of an isolated future is needed, one can call \text{f.get} to retrieve it.

Requirements The \(\lambda\)-expression performed by an isolated future must be pure—a function that only takes its arguments as input and only outputs its returned value with no side effects.
Assumptions on the Programmer’s Side  None.

Design Notes  In DPR, the result of a future should be explicitly obtained by the \texttt{get} method. It is also possible to allow obtaining a future’s result implicitly, the first time it is referenced. The implicit approach is used in some functional programming languages, like in Multilisp. DPR chooses the explicit way of obtaining future results (similar to Java) based on (a) easy identifications of continuations, and (b) easy language implementations. For (a), with an explicit \texttt{get} method call, programmers are clear about the potential concurrency of an isolated future. This explicit concurrency may help programmers to reason about the isolation requirement, which is not required by Multilisp. For (b), adding an extra \texttt{get} call simplifies potential language implementations: it is possible to provide an isolated future implementation without changing the existing Ruby language virtual machines. With existing Ruby MRI \cite{Ruby1995} or JRuby \cite{JRuby2001} language virtual machines, we did not find a way to integrate implicit future result acquisition, without significant changes to the language VMs. This integration may be easier with some speculation-based systems, such as Ruby BOP \cite{Dingetal2014}.

The requirement on pureness is more of a compromise than a choice. Futures in general create fork-join style concurrency patterns, which may not be properly nested. In practice, if concurrent tasks are not properly nested, it would be much harder to design a low-overhead run-time mechanism to verify their independence. Requiring functions performed in a future to be pure preserves the convenience of futures for many use cases, but reduces the requirements on language implementations at the same time. Pureness is checked during the run time.
3.4.2 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 (*Singleton*).

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.

**Usage** A pipeline can be constructed by connecting a sequence of lambda functions by the “>>” operator:

```
p = ->{|x|  o0}  >>
      ->{|x|  o1}  >>  ...
```

Each lambda function represents a stage of the pipeline.

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.

**Requirements**  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 $\langle s_1, s_2, ... \rangle$. Stage $f_a$, operating on element $s_i$, must be independent of stage $f_b$, operating on element $s_j$, if and only if $a + i = b + j$.

For each pipeline, input and output streams need to be properly set prior to the `run` call. (Note: If `run` is called before both streams are set, it will immediately return `false`.)

To avoid run-time type errors, each stage should output values of the type expected as input by the subsequent stage.

**Assumptions on the Programmer’s Side**  None.

**Example**  Figure 3.7 sets up a 3-stage pipeline, and feeds it with a input stream of objects \{obj0, obj1, obj2, obj3, obj4\}. This pipeline computes relative forces for consecutive objects, generates velocity, then updates the position for each object. The execution pattern of this pipeline is shown in Figure 3.6.
class MyInStream
  @@data = [obj0, obj1, obj2, obj3, obj4]
  def getAndMove
    return @@data.shift
  end
end

class MyOutStream
  def setAndMove(obj)
    puts obj.to_s
  end
end

p = ->{|x| computeForce(x)} >> ->{|x| generateV(x)} >> ->{|x| updatePos(x)}
p.setInStream(MyInStream.new)
p.setOutStream(MyOutStream.new)
p.run

Figure 3.7: Code example of a 3-stage pipeline, on a sequence of 5 objects {obj0, obj1, obj2, obj3, obj4}. 
3.5 The DPR Memory Model

To the best of our knowledge, there is no existing work to fully formalize the memory consistency model of any scripting languages (largely due to the immature status of parallel scripting languages). We do believe, however, that having such formalized memory models for scripting languages would be helpful. In this section, we discuss the memory model of DPR. Our discussion is based on the system model introduced in Section 2.2.

In DPR, two operations are *conflicting* if and only if either 1 or 2 holds:

1. At least one of the two operations is not an AC op. They operate on the same memory location, and at least one of them is a write.

2. Both operations are AC ops, but do not commute with each other.

In DPR, operating on “the same memory location” means two operations either work on the same field of the same object or both work on the metadata of the same object (including instance variable table, collection length information, and so on). In any given DPR program execution, there exists a total order on atomic commutative operations, consistent with both program order and the semantics of the operations.

Compare to other languages’ memory models discussed in Section 3.7.1, DPR’s memory model is simpler thanks to its (limited) split-merge style parallelism. In DPR, program behavior is undefined if a future conflicts with other operations in the program, or two branches of the same independent tasks parallel construct conflict with one another, or pipeline input $s_i$ on stage $f_a$ conflicts with pipeline input $s_j$ on stage $f_b$ where $a + i = b + j$. In the absence of such conflicts, the program is deterministic: every read (outside an AC op) has a unique most recent preceding write, whose value it returns, and every AC op has a unique
most recent preceding op with which it does not commute, and whose behavior therefore governs its own.

3.6 RB-DPR: Implementing DPR

All of the DPR 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. RB-DPR provides such an integrated implementation—specifically, within the JRuby [JRuby 2001] virtual machine.

RB-DPR has two major parts: a back-end with a thread pool and a work-stealing scheduler, and a front-end to implement each DPR parallel construct. We use a work-stealing scheduler to dynamically balance loads among threads, and use a thread pool to reduce the overhead for thread creation and destruction. The front-end uses parallel contexts to manage concurrent tasks created in parallel constructs. Each parallel context contains an (unordered) set of tasks and handles the launching and merging of these tasks. Tasks are closures that are basic scheduling units for the scheduler. When a new task is posted to a parallel context by a worker thread, the context enqueues the task to that thread’s task queue. To allow nested parallel constructs, a task may generate new parallel contexts. The RB-DPR runtime keeps the nested depth for each parallel context.

3.6.1 Back-end Implementation

The back-end scheduler of RB-DPR applies the concept of “work-stealing” from the Cilk language runtime [Blumofe et al. 1995]. In the RB-DPR scheduler, each thread has a local task queue, and can dequeue (and execute) tasks from this
Figure 3.8: This is an example task queue data structure for a worker thread. Each node in the linked lists is a separate task. As shown in the figure, the thread itself and other threads dequeue tasks from two sides of the task queue.

queue. It is possible that during an execution, some threads may drain their task queues, while the others are still running. In this case, the thread draining its own queue can dequeue tasks from others’ queue, “stealing” tasks from others. In this way, the run-time system can automatically balance the load among multiple threads. Our current library implementation has a similar task queue design as Cilk 5 [Frigo et al., 1998]. Each task queue $tq$ is a variable-length array, while $tq[i]$ is a sub-queue that contains all tasks in nesting level $i$ of the worker thread. To submit new tasks created by a task with nesting level $i$, the worker thread enqueues all new tasks to sub-queue $tq[i+1]$. A task queue is empty if and only if all of its sub-queues are empty.

During run time, a worker thread $T$ repeats the following operations until the whole program has finished:

1. $T$ tries to dequeue a task $t$ from its own task queue. If this operation
succeeds, go to step 3.

2. $T$ becomes a work stealer and tries to steal a task $t$ from a randomly chosen worker thread. If this operation succeeds, go to step 3, or else, redo this step.

3. $w$ executes $t$. Note that since we allow nested parallel constructs, it is possible that some new tasks are created during the execution of $t$.

In Step 1, the local dequeue operation always dequeues the task that has the deepest nesting level. If local task queue is empty, a thread becomes a stealer and try to steal a task from a pseudo-randomly picked thread. When stealing a task from other threads (in Step 2), the remote dequeue operation always returns the task that has shallowest nesting level. This is consistent with the Cilk runtime’s behavior. The main intuitions behind this, as explained by Blumofe et al. (1995), are: (a) the local thread and stealers access task queues from different directions, reducing contention on the local dequeue operation, and (b) a stealer will preferably steal the task with potentially the largest granularity, which reduces the total number of stealing operations.

One design choice for the thread pool is how to manage the “main” thread. The main thread is the thread that launches with the program, and then manages the creation of the thread pool. A simple solution is to use the master-slave model that is commonly used in parallel programming: After creating all worker threads in the thread pool, the main thread will only wait for an exit. To maximize the use of all existing threads, in RB-DPR, main threads are also managed by the thread pool. After the main thread creates the thread pool, it will add itself to the pool. This poses some challenges in the implementation. All worker threads are extended RB-DPR worker threads, while the main thread is a normal Java thread created by the language virtual machine. Therefore, the RB-DPR scheduler stores extended metadata for the main thread.
3.6.2 Implementing Parallel Constructs

On top of the RB-DPR backend, we implement DPR parallel constructs. Generally speaking, each construct other than atomic commutative operations (AC ops and reductions) creates specific execution patterns. Reductions handle concurrent put calls and the synchronized get call. AC ops create special commutative “annotations”, and store the commutative relationships.

**Independent Tasks** The implementation of `co` statement is straightforward. For each `co` statement, we create a parallel context object, then add all $\lambda$-closures in the statement into the context, with each closure as a separate task. The execution then spins until all tasks in this parallel context are finished.

Parallel iterators can be implemented in a similar fashion as `co` statements. Each `.all` call is mapped to a unique parallel context. Each iteration (or a fixed number of iterations) in the call forms a task and tasks are sequentially added to the `.all` call’s parallel context. In this way, the overhead of enqueuing all tasks is bounded by $O(n)$ ($n$ is the total number of tasks), and the whole task distribution process can only be executed sequentially. When $n$ is large, the task distribution process may become a potential bottleneck. A binary tree shaped task distribution would help to reduce the overhead of enqueuing tasks and parallelize the task distribution process. In practice, however, we noticed that a sequential, linear task distribution does not cause any significant problem when the granularity of tasks is carefully picked by the programmer and the total number of cores is not large (24 cores).

**Reductions** In a reduction object $R$, there are per-thread storage spaces $R(tid)$ to store the internal status of the reduction. Thread local storage guarantees independence among all push operations. On each `push(val)` operation from thread $tid$, $R(tid)$ is updated with $R(tid)' = R(tid) \ OP \ val$, where $OP$ is the chosen
operator of the reduction (set by `Reduction::TYPE`, as introduced in Section 3.3.1). The `get` method of a reduction will apply the chosen operator \( op \) on all thread local data elements, to produce the final result of this reduction. If there are \( n \) threads in total, the result \( r \) can be computed by

\[
    r = OP_{i=1}^{n} R(i).
\]

If \( n \) were large, it is beneficial to implement this step in a tree-shaped fashion. Note that according to the specification of DPR, on the same reduction object, `push` operations must happen before any `get` operations. Therefore, the implementation of `get` does not need to consider the case of concurrent `push` calls.

**AC Ops** AC ops are special annotations for commutativity relationships. Commutativity relationships are stored in *commutative tables* (\( cTables \)). A \( cTable \) \( T \) is a symmetric table, indexed by AC ops. For any two AC ops \( m_i \) and \( m_j \), \( T(m_i, m_j) \) and \( T(m_j, m_i) \) stores a boolean value, indicating if method \( m_i \) is commutative with method \( m_j \). On each `setACOps` call, the language runtime performs the following tasks:

1. Find the \( cTable \) \( T \) for the current class.
2. For each pair of the arguments in the `setACOps` call \( (m_1, m_2) \), set \( T(m_1, m_2) \) and \( T(m_2, m_1) \) to true.

Each `setNoCommute` call is performed in the same fashion as `setACOps` calls, only setting \( T(m_1, m_2) \) and \( T(m_2, m_1) \) to false.

Some methods of built-in collection types (like arrays and sets) are implemented as AC ops in RB-DPR. More discussions can be found in Section 4.3 “Integration with the virtual machine”.

**Isolated Futures** Similar to `co` statements, for each `Future`, we also create a parallel context and add the \( \lambda \)-closure in the future as a separate task. Different to `co` statements, we do not wait for the task to finish at the end of the enqueue
process. We only wait for and return the result for the `Future` when its `get` method is called.

**Pipelines** To support intuitive usage of the `>>` operator, we override it in RB-DPR’s `Pipeline` class. Pipelines are implemented in a cycled execution pattern. On each cycle, a pipeline object executes the current iterations in all of its stages concurrently, with each stage as a separate task, and blocks until all of them finish. After this barrier, the pipeline object marks the result of each stage as the input of the next stage (if this stage exists), stores the result of the last stage by calling the `getAndMove` method of the output stream, and moves in a new element from the input stream by calling the `getAndMove` method. The pipeline finishes after the `setAndMove` method of the input stream returns the first `nil`. After this, the whole pipeline is drained.

### 3.7 Related work

In this section, we introduce related work to the language design of DPR, including memory consistency models, and DPR’s parallel constructs in different existing parallel programming languages/libraries.

#### 3.7.1 Language Memory Consistency Models

A language’s memory consistency model (often called “memory model”) formally specifies its legal behaviors (reads-see-writes relationships) for a multithread program. We argue that, to design a parallel language comprehensively, it is particularly important to consider the memory consistency model of the language.

A language memory model must accommodate needs from two sides: For language users, it must be (subjectively) easy enough to reason about, eliminating counter-intuitive execution results like out-of-thin-air values. For language
implementors, it must enable optimization on various platforms. *Sequential consistency* (Lamport, 1979) is the model that has been widely accepted today as an intuitive (but potentially expensive) memory model. An execution is sequentially consistent if and only if a total order on all of its operations exists, such that the total order is consistent with its program orders, and each read returns the value of the most recent write to the same location. The Java (Manson et al., 2005) memory model guarantees sequentially consistent program behaviors for data-race free programs. The C++ (Berger et al., 2009) memory model guarantees sequential consistency if the program is data-race free and does not perform any non-sequentially consistent loads or stores on atomic variables. For racy programs, these two models have different solutions. The C++ memory model leaves racy programs’ behavior as undefined, while the Java model validates such executions by committing actions, and rules out executions that fail the validation. The C++ model enables more chances for platform-specific optimizations, while the Java model guarantees racy programs would not break the “sandboxing” mechanisms inside its language virtual machine.

### 3.7.2 DPR Parallel Constructs

**Co-begin and Unordered Iterators** DPR programs are properly-nested with co-begin statements and parallel iterators. Constructs to create properly-nested parallelism can be found in many parallel programming languages.

Co-begin is a parallel construct that has been widely used in parallel programming language/libraries. It is introduced by Algol 68 (Van Wijngaarden et al., 1969) as *collateral clauses*. In OpenMP, programmers can use `#pragma omp parallel` to notate statements that should execute in parallel, and then (explicitly) join all statements after they are all finished. In Cilk, programmers can do so by using the `spawn` and `sync` statements. In deterministic parallel Java,
co-begin statements behave in the same way as in DPR, with only syntactical differences.

Unordered iterators are similar to doAll loops (Wolfe, 1995), which appear in many parallel programming languages. In a doAll loop, each iteration of the loop can execute in a concurrent fashion. Designs similar to DPR’s unordered iterators include the parallel for pragma in OpenMP, FORALL loops in High Performance Fortran, the default for loops in Fortress, the upc forall statements in Unified Parallel C (Carlson et al., 1999), the cilk for statement in the Cilk++ (Leiserson, 2009) language, and the foreach statement in deterministic parallel Java.

Reduction Reductions are often used in collecting and post-processing results from multiple parallel tasks. They appear in several languages. As in DPR, the programmer has to decide which field to collect and define the combine function \( f(x_1, x_2) \) to perform during the reduction. In order to apply the combine function in an arbitrary order, \( f \) has to be commutative. If \( f \) is also associative, it is possible to recursively apply \( f \) to the results of the previous sub-tasks \( (f(result_1, result_2)) \), resulting in a tree-shaped reduction pattern.

OpenMP supports reductions by providing a reduction() pragma. For example, \#pragma omp parallel reduction(+:i) \ will sum up the value of i among all parallel tasks. MPI also uses reductions (MPI_Reduce) to collect results from different memory spaces. Like DPR, both MPI and OpenMP restrict the combine functions to be language-predefined. The MapReduce programming model (Dean and Ghemawat, 2008) uses reductions to perform summary operations after the independent filtering or sorting stage (map stage), and supports user-defined combine functions. In Fortress (Allen et al., 2005), it is also possible to provide user-defined combine functions, by user-defined “big operators”.
**Atomic Commutative Operations**  The idea of AC op in DPR is in a fashion reminiscent of *boosting* (Herlihy and Koskinen, 2008) in transactional memory systems. Transactional boosting aims to “transform a large class of highly-concurrent linearizable objects into highly-concurrent transactional objects”, as noted in its original paper. In transactional boosting, as in AC ops, and in contrast to conventional transactional memory systems, operations are no longer modeled by low-level read/write operations. Instead, a transaction contains a series of high-level operations like method calls (similar to the AC ops in DPR). These operations, once in a transaction, appear as atomic steps. Unlike AC ops in DPR, transactional boosting requires that each high-level operation should have an inverse. Once some transactions need to abort (potentially due to run-time conflicts), the system would apply the inverse operations for each performed operation. For example, in a transactional boosting system, on transaction abort, a `list.insert(A)` call will be rolled back by its predefined inverse operation, `list.remove(A)`, if A was not in the list prior to the insert. AC ops in DPR never abort and therefore do not require inverses.

Commutativity verification has been used to help reason about the correctness of parallel programs. In the Galois (Kulkarni et al., 2007) and DPR systems, commutativity relationships among methods are specified by programmers. However, where Galois uses these relationships to infer a parallel schedule for semantically sequential tasks, DPR use these relationships to verify the determinism of the program. Concurrent to our DPR work, the concept of “commutative race” was proposed by Dimitrov et al. (2014). This work proposes a specification to define commutative properties among method calls (including their parameters and return values). On top of this specification, a vector clock based dynamic detection algorithm is applied to detect concurrent operations that are not commutative in a specific execution.
Figure 3.9: Example code segment for StreamIt, setting up a trivial pipeline stage with one source and one sink.

**Future**  MultiLisp (Halstead, 1985) introduced the future construct in its original paper. A future indicates that an expression can be evaluated concurrently to its continuation and its evaluation has to finish when the result of the expression is used. Some non-functional languages, like Java, also support futures. Futures may appear to be similar to co-begin statements, since both may explicitly create task parallelism. However, futures can create non-properly nested, fork-join style parallelism, while parallelism created by co-begin statements is all properly-nested. In addition, co-begin statements enforce a join operation to guarantee all branches are finished, but futures do not. A future may perform the join operation lazily, only when its result is used. For example, a future $f$ may be used in only one branch of its continuation. In this case, other branches in $f$’s continuation do not need to wait the future to finish.

**Pipelines**  There is existing work in designing and implementing pipeline-style stream process languages. The Ptolemy project (Eker et al., 2003) uses data-flow graphs and pipelines to model concurrent real-time embedded systems. The StreamIt (Thies et al., 2002) project proposes a streaming language that uses pipeline constructs to build stream/signal processing programs. Pipelines are built in a data flow fashion, chaining components (stages) together. Typical chaining operations include serial chaining (implies program order), parallel chaining (implies task independence and concurrency), and loops. Each stage of the pipeline is specially programmed for its operations. Pipelines can be further chained together.
or reused to build more complex programs. In comparison to the pipeline objects in DPR, StreamIt aims to provide comprehensive support to signal processing programs. Therefore, its programming model is more specialized to streaming uses. It has more freedom in expressing pipeline data flow and better support in modeling input and output. DPR focuses on the simplicity side, instead of the generality side, of stream programming. In DPR, the \( \gg \) operator for \( \lambda \)-functions and existing pipelines provides functionality comparable to the basic \texttt{add} statement in StreamIt to build a pipeline, while the \texttt{getAndMove} method supported by input streams can be seen as an iterator method for the input queues in StreamIt. However, due to the limited expressiveness of DPR on streaming programs, it should not be seen as a pure stream-processing language.

When implementing pipelines, the runtime may have significant impact on the actual performance of a program. The SEDA architecture (Welsh et al., 2001) proposes a staged event-driven processing system for on-line services. A SEDA system controls and schedules system resources dynamically for stages of hosting applications, according to asynchronous events. The Capriccio (von Behren et al., 2003) project provides a scalable run-time thread library for SEDA-like staged systems. Given the wide range of applications for scripting languages in web programs, DPR programs may run on top of systems like SEDA or Capriccio. In this scenario, the pipeline construct in DPR may assist in making scheduling decisions for stage-based runtimes. We leave this integration as future work.
4 Dynamic Enforcement of Determinism

As discussed in Chapter 2, nondeterminism in a parallel program comes from two types of races: data races and synchronization races. Conflicting read/write operations that are not ordered by happens-before cause data races. Conflicting synchronization operations on the same synchronization object form synchronization races. Correct DPR programs are data race free, and only atomic commutative operations (reductions and AC ops) can induce a (benign) synchronization race. Enforcement of determinism in DPR thus amounts to verifying that:

1. There are no (low-level) data races

2. Higher-level operations are called concurrently only when they commute.

The first subsection below provides an overview of TARDIS. Algorithmic details appear in Section 4.2.

4.1 TARDIS Overview

TARDIS combines a log-based data-race detector (Sections 4.1.1 and 4.2.1) with a set of extensions (described in to accommodate higher-level operations like reductions, AC ops, isolated futures, and pipelines (Sections 4.2.2 and 4.2.2).
4.1.1 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 (Flanagan and Freund 2009; Nistor et al. 2009; Effinger-Dean et al. 2012; Muzahid et al. 2009; Devietti et al. 2012). Some track lock sets, looking for conflicting accesses not covered by a common lock (Savage et al. 1997; Zhou et al. 2007). Still others take a hybrid approach (Yu et al. 2005; Xie and Xue 2011; O’Callahan and Choi 2003; Choi et al. 2002). In general, data race detection introduces significant run-time overhead. Recent work has suggested that this overhead might be reduced by crowdsourcing (Kasikci et al. 2013).

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 (static) type system, as in DPJ, or by explicit assignment of “ownership” (Heumann et al. 2013), but we believe the resulting complexity (and static exposition) 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) (Feng and Leiserson 1997) 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 (Cheng et al., 1998). Mellor-Crummey’s offset-span labeling (Mellor-Crummey 1991) 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 (Raman et al., 2012), which runs in parallel and provides precise data race detection for arbitrary split-merge (async-finish) programs.\(^1\)

Like most recent race detectors, SPD3 relies on shadow memory to store metadata for each shared memory location. On each read and write to a shared location, 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, 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 Chapter 5, it can provide a significant performance advantage. Second, it is easily extended to accommodate higher-level operations. Several past researchers, including Schonberg (Schonberg 1989) and Ronsse et al. (Ronsse and

\(^1\) In recent work, concurrent with our own, the Habañero Java group has extended SPD3 to accommodate higher-level commutative operations (Westbrook et al., 2014). 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.
De Bosschere, 1997; Ronsse et al., 2003 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.

4.1.2 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 user-defined atomic commutative operations (AC ops). As in Galois (Kulkarni et al., 2007), commutativity relationships among methods in DPR are specified by programmers. Where Galois uses 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 that have been declared to commute with one another.

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 undecidable for general cases. 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 [Rinard and Diniz 1997, Kulkarni et al. 2011, Kim and Rinard 2011]. They are, for the most part, orthogonal to the work reported here. Integrating them with TARDIS is a subject for future work.

For futures, 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.

4.1.3 Comparative 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 4.1.1 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:

$$O = N \times C_{sm}$$

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. 4.2.2 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:

$$O = N \times C_{th} + K \times C_{ck}$$

In general, it seems reasonable to expect $N \gg K$ and $C_{th} < C_{sm}$, which suggests that task-history–based checking may be significantly faster than shadow-memory–based checking. Worst case, for space consumption, 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 de-coupled 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 Chapter 5 (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 4.3.

4.2 TARDIS Algorithm

4.2.1 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. $T.current_set.add((id, field_no), t)$
When task $T_p$ spawns $n$ child tasks, as shown in Algorithm 2, two new access sets ($T_i.current_set$ and $T_i.local_set$) are created for each child $T_i$. At the same time, $T_i.parent$ 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: $T_i.current_set \leftarrow \emptyset$
4: $T_i.local_set \leftarrow \emptyset$
5: $T_i.parent \leftarrow T_p$

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 $\cap$ to represent “intersection” of access sets (really more of a join): $S_1 \cap S_2 \equiv \{(o, f) \mid \exists (\langle o, f \rangle, t_1) \in S_1, (\langle o, f \rangle, t_2) \in S_2 : t_1 = write \lor t_2 = write\}$. We use $T_p.local_set$ to store all accesses of all the concurrent siblings that merged 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 3, $T_p.local_set$ 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.

**Data Structure for Access Sets** Because the number of memory operations issued by tasks varies dramatically, we use an adaptive, hybrid implementation of
Algorithm 3 On task merge

Require: child task $T_c$

1: $T_p \leftarrow T_c.p\text{arent}$
2: sync on $T_p$
3: if $T_p.local\_set \cap T_c.current\_set \neq \emptyset$ then
4: report a data race
5: $T_p.local\_set \leftarrow T_p.local\_set \cup T_c.current\_set$
6: if $T_c$ is the last task to join then
7: $T_p.current\_set \leftarrow T_p.current\_set \cup T_p.local\_set$
8: $T_p.local\_set \leftarrow \emptyset$

access sets to balance performance and memory consumption. As illustrated in Figure 4.1, 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, a list may overflow, at which point we convert it to a hash table. The table implements a mapping from object ids 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 usually 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
Figure 4.1: 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.
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 [Muzahid et al., 2009]) 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.

4.2.2 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** 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.

Ordering requirements among higher-level methods, as declared by the programmer, are stored in per-class commutativity tables (cTables). 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 4.2.1 are directed to the corresponding normal_set fields. Both normal_set and atomic_set use the same data structure described in Section 4.2.1. 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.

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 by the verify func-
Algorithm 4 On method call

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

1: \( \text{if } m \text{ is an annotated method of } C \text{ then} \)
2: \( \text{in}_\text{AC}_\text{op}[th] \leftarrow \text{true} \)
3: \( T.curr\_hist.cml[o].add(m) \)
4: // call \( m \)
5: \( \text{in}_\text{AC}_\text{op}[th] \leftarrow \text{false} \)

Algorithm 5 On task merge

Require: child task \( T_c \)

1: \( T_p \leftarrow T_c.parent \)
2: \( \text{sync on } T_p \)
3: \( \text{if } T_p.local\_hist.normal\_set \cap T_c.curr\_hist.normal\_set \neq \emptyset \text{ then} \)
4: \( \text{report a normal R/W conflict} \)
5: \( \text{if } T_p.local\_hist.atomic\_set \cap T_c.curr\_hist.normal\_set \neq \emptyset \text{ then} \)
6: \( \text{report an AC-op vs. non-AC op conflict} \)
7: \( \text{if } T_p.local\_hist.normal\_set \cap T_c.curr\_hist.atomic\_set \neq \emptyset \text{ then} \)
8: \( \text{report an AC-op vs. non-AC-op conflict} \)
9: \( \text{if } \neg \text{verify}(T_p.local\_hist.cml, T_c.curr\_hist.cml) \text{ then} \)
10: \( \text{report a commutativity conflict} \)
11: \( T_p.local\_hist \leftarrow T_p.local\_hist \cup T_c.curr\_hist \)
12: \( \text{if } T_c \text{ is the last task to join then} \)
13: \( T_p.curr\_hist \leftarrow T_p.curr\_hist \cup T_p.local\_hist \)
14: \( T_p.local\_hist \leftarrow \emptyset \)
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 ($H_a \uplus H_b$)

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 $H$

tion, shown in Algorithm 6. The $\uplus$ 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 3.7.2, pipeline objects in DPR introduce a different task execution pattern to accommodate loop-carried dependences. 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.

4.3 Engineering Issues for TARDIS

Generally speaking, TARDIS is integrated into RB-DPR by adding instrumentations to all memory operations, parallel construct method calls, and method calls that are marked as AC ops. With these instrumentations, TARDIS can perform sound and complete determinism verification. There are, however, certain engineering challenges.

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.

In JRuby, a static scope is generated for every method/block by the parser that defines which variables are accessible in a particular scope. At run time, a dynamic scope, acting as a stack frame, is generated and pushed into the running stack by the language virtual machine that stores the variable values. Dynamic scopes in JRuby are dynamic representations of static scopes and they are instanced out
on a per-call basis. For determinism checking, we also assign a unique ID to each
dynamic scope, as a scope holds all the object references to parameters and local
variables. Any concurrent tasks within a scope could access the scope with an
offset to read/write these references. Each memory location can then be identified
as an \((id, field\_no)\) pair.

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 re-
quire 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. In RB-DPR, the resizing
operation is implemented as an atomic operation and marked with commutativity
relationships. The possibility of resizing 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 synchroniza-
tion. 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. This is achieved by storing more run-time information on each instrumentation. 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.
Working with different parallel granularities  

RB-DPR partitions each unordered iterator into chunks, each of which contains multiple iterations. Loop chunks are then distributed among working threads and executed in parallel. Although there are coarse grained partitions of parallel loops, TARDIS can still find all the races in the program through one execution. The reason for this is because: (1) iterations in a same chunk are executed one by one by the same working thread and the access set of each iteration is intersected and merged at the end against other iterations in a same chunk; and (2) a union of all the access sets generated by different iterations in a chunk is calculated at the end of a chunk and the intersections are performed among these unions from different chunks.

This coarse grained partition with fine grained execution reduces the memory consumption required by the metadata of TARDIS. In practice TARDIS does not need to maintain the parent-child relationship for each iteration after its execution, because the information for conflict detection is already carried in its access sets.

4.4 Existing Data Race Detection Algorithms

In Section 4.1 we contrasted TARDIS, abstractly, with other styles of race detectors – those based on shadow memory in particular. In Chapter 5 we present performance comparisons. The current section provides more detail on these alternative approaches.

In general, dynamic data race detectors may introduce significant run-time overhead, and low overhead is desired. It is also critical to understand whether a detector will report false positives and/or false negatives during race detection. A false positive can be considered a “false alarm” of a data race: the race detector may report a “race” that does not actually exist in the program execution. Having false negatives means the detector may miss some true data races. In general, a dynamic race detector is sound if it reports no false positives and is complete if
on a given input, it reports all races in all possible executions of the program.

Dynamic race detectors are mainly designed to discover races in existing program source code and guide programmers to fix them. In this use case, it is critical to have a high “hit rate”, generating a low ratio of false positives. This is because reporting a false positive may waste significant numbers of human-hours to study, understand, and verify the reported race. False negatives are of course not desirable. But for debug uses, leaving some false negatives would not cause any consequences worse than not running the race detector.

As discussed in Section 4.1.1, most existing dynamic race detectors use shadow memory. To perform dynamic race detection, metadata for each memory location $l$ is stored in a shadowed location $Sh(l)$, where the shadow function is a one-to-one mapping $Sh : L \mapsto L_{shadow}$. $L$ is the set of all instrumented memory locations in a program, and $L_{shadow}$ represents all shadow memory elements. The race detector needs to instrument each memory operation, query and update its metadata in shadow memory, and detect any concurrent accesses to the same location.

Race detectors need to identify concurrent operations with regard to synchronization operations. Typically, a race detector is expected to support lock acquisitions and releases. The race detector may use synchronization orders to infer the happens-before order and then reason about concurrency. Given two threads, $T_a$ and $T_b$, if statement $S_1$ in $T_a$ is a release operation on lock $m$, and statement $S_2$ in $T_b$ is an acquire operation on $m$, and $S_1 <_s S_2$, then $T_{a,S_1} <_{hb} T_{b,S_2}$, where $T_{a,S_1}$ stands for the prefix of $T_a$ until (and including) $S_1$, and $T_{b,S_2}$ means the continuation from (and including) $S_2$.

4.4.1 Race Detection for General Programming Models

Given lock acquisitions and releases, one heuristic is to trace the locking pattern for each location on each access. If any two operations on the same location are
performed by two threads, and not protected by the same lock, the race detector can declare a race. Savage et al. (1997) and Zhou et al. (2007) describe lockset-based race detectors. Each location $l$’s shadow memory $Sh(l)$ stores the set of locks $M_l$ that “protect” this location. $M_l$ is initialized to all possible locks of a program. Each thread $t$ maintains its own set of acquired locks in lock set $M_t$. On each successful lock acquisition on lock $m$ from thread $t$, $M'_t = M_t \cup \{m\}$. And on each lock release on $m$, $M'_t = M_t - \{m\}$. On each access on location $l$ by thread $t$, the runtime system checks if $M_t \cap M_l = \emptyset$, and declares a race if so. Then, $M_l$ is updated to $M_t \cap M_l$. For example, suppose two threads both trying to access location $l$, but accidentally grab two different locks. The lock set algorithm will declare a race since after both threads’ accesses, the resulting lock set $M_l = \emptyset$.

The advantage of a lockset-based algorithm is that it is relatively simple and has low overhead. However, the most significant disadvantage is its high rate of false positives. False positives exist because not all happens-before orders are introduced directly by a lock. There are counter-examples including memory initializations, read-only locations, and privatization (Spear et al., 2007). Under these cases, races may not exist, but the run-time system may still declare a race since conflicting accesses are not protected by the same lock. Take privatization for example: the privatization process itself may be correctly synchronized. After privatizing location $l$, a thread may safely access this location and then publish $l$ after the access. The whole process can safely be performed by multiple threads and does not introduce a race. However, since the access on $l$ is not protected by any locks, it may be declared as a race by a lockset-based race detector.

Another approach to detect a race is to directly reason about the happens-before order of the program (Flanagan and Freund, 2009; Nistor et al., 2009; Effinger-Dean et al., 2012; Muzahid et al., 2009; Devietti et al., 2012). This leads to vector-clock race detectors. In a vector-clock data race detector, on an execution with $n$ threads, thread $i$ is assigned an $n$-dimensional vector $V_i$, \( V_i = (v_{i,1}, v_{i,2}, \ldots, v_{i,n}) \).
where \( V_i(j) \) \((j \neq i)\) indicates the most recent time stamp of thread \( j \) known to happen before the current time in thread \( i \). \( V_i(i) \) is the current time stamp of thread \( i \). Thread vector clocks \( \{V_i\} \) are updated on each lock acquire and release, consistent with the synchronization order. The relationship of two vector clocks can be decided via element-wise comparison: two vector clocks, \( V_1 \) and \( V_2 \), are \textit{not} concurrent if and only if either of the following two items holds:

1. For each index \( i \) such that \( 0 < i \leq n \), \( V_1(i) \geq V_2(i) \) and at least one of those relationships is strictly greater.

2. For each index \( i \) such that \( 0 < i \leq n \), \( V_1(i) \leq V_2(i) \) and at least one of those relationships is strictly smaller.

Otherwise they are concurrent. To trace memory accesses, shadow memory location \( Sh(l) \) stores the read and write vector clocks of location \( l \), namely \( V_{lr} \) and \( V_{lw} \). On each memory access to location \( l \) by thread \( i \), \( V_{lw} \) and \( V_{lr} \) may be checked against \( V_i \). On a read, the detector checks \( V_{lw} \) against \( V_i \), and then updates \( V_{lr}(i) = V_i(i) \). On a write operation to \( l \), the detector checks both \( V_{lr} \) and \( V_{lw} \) against \( V_i(i) \), and updates \( V_{lw}(i) = V_i(i) \). If any of the checked pair(s) shows that a concurrent thread has accessed this location, the race detector will declare a race.

One shortcoming for vector-clock race detection is the relatively high run-time overhead. Vector-clock methods in general require more memory to maintain the metadata. The overhead on each memory operation instrumentation is also higher than in the lockset-based method, since checking the concurrency of two vector clocks is generally more expensive than intersecting lock sets. The FastTrack project \([\text{Flanagan and Freund} 2009]\) proposed an optimized algorithm that not only significantly reduces the space consumption of vector clocks, but also reduces the checking overhead for most reads and writes. The algorithm compresses vector clocks for both threads and memory locations into one dimension and only
expands a vector clock into its full size when multiple concurrent reads may also be concurrent with an incoming write to the same location. In practice, it is observed that the compressed vector clocks are sufficient for most memory instrumentations. Therefore, the algorithm can significantly outperform conventional vector-clock race detection.

Vector-clock race detectors can accurately reason about happens-before order during run time. Therefore, in theory, a vector-clock race detector could be sound (no false positives). However, it is still hard to build a complete vector-clock race detector for general parallel programs. The problem is that vector-clock race detectors work only on the happens-before order of the current execution; they cannot infer all possible happens-before orders of a program. In other words, they are not complete, even for a given input.

4.4.2 Language Specific Race Detection

If the race detector is tailored for a specified programming model, there are potential chances to make the detector both sound and complete. The language may restrict the possible patterns of concurrency, which simplifies precise race detection. For example, programming languages with only properly-nested independent tasks (like DPR with only its independent task constructs) guarantee Singleton determinism (discussed in Section 2.4.2). Detectors for such languages need only to check if all concurrent tasks are independent and can be complete because the happens-before order they work on is the only happens-before order of the program (on a given input).

Compared to general race detectors, language specific detectors may need to work on concurrent “tasks”, rather than just on “threads”. Conventional race detectors are designed for general, fork-join based parallel programs and often work on a limited number of threads. Concurrent tasks in a language may be
significantly more numerous than threads. This poses a challenge to the design and implementation of low-overhead language-specific race detectors. Detectors should also deal with the case where concurrent tasks are nested.

The Nondeterminator race detector (Feng and Leiserson, 1997) for the Cilk programming language pioneered dynamic race detection for a specified programming model. The Nondeterminator detector is sound and complete for the basic Cilk language, when there are no locks in the program. It is a shadow memory based sequential race detection algorithm. It handles dynamic concurrent task creation by using the SP-bags algorithm. The S-bag for a given task indicates other tasks with which the given task is serial (ordered). The P-bag indicates other tasks with which the given task is parallel. During execution, a Cilk program is serialized into a sequential execution. The S-bag of a procedure $F$ is created on task spawn, and the P-bag of $F$ is merged back to its S-bag on sync. When a parallel procedure returns, both its S-bag and P-bag will be merged into its parent task’s P-bag. For each memory location $l$, $Sh(l)$ stores the task id of $l$’s last accessor (reader and writer, marked by procedure id). On each memory access to $l$, the race detector checks $Sh(l)$ to see if the last accessors are conflicting to this access and in P-bags. The detector will declare a race if so. The runtime also keeps updating the shadow memory after each non-racy access, replacing the accessor by the current task’s id.

The SPD3 race detector for the Habañero Java project is a parallel race detector for the async-finish (properly nested) parallel programming model. In a Habañero Java program, statements inside an async {} block are concurrent to the continuation of this block. Statements inside a finish {} block are guaranteed to finish at the end of the block. SPD3 is both sound and complete. It allows race detection to be parallel by using a Dynamic Program Structure Tree (DPST). A DPST describes the serial/parallel relationship of all tasks in a Habañero Java program. It has three kinds of nodes: F node, representing the code block of a
finish statement; A node, representing the code block of a async statement; and step node, representing a maximal sequence of statements with no async or finish statements. When executing inside an async or finish code block, on each async or finish statement, new A or F nodes are created as child nodes of the DPST node representing the current block. This behavior builds up the tree structure of DPST. Shadow memory is still used in SPD3. The shadowed location $Sh(l)$ of a memory location $l$ stores the task ids of the most recent accessors of $l$. On each access to location $l$ by task $T$, the concurrency relationship between $T$ and $Sh(l)$ is decided by checking through the DPST. More precisely, the race detector synchronously checks the lowest common ancestor (LCA) for $T$ and each element of $Sh(l)$ and declares a race if the LCA of both tasks is a async node. In this fashion, the SPD3 detector can precisely detect all conflicting memory operations that are concurrent in a Habañero Java program.
5 Performance Evaluation

Our evaluation of RB-DPR aims to quantify the following:

• Scalability of the implementation, 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

5.1 Evaluation Setup

We evaluate RB-DPR using 13 applications, from a variety of sources. Most were originally written in some other parallel language, which we hand-translated into RB-DPR. During the translation, we used TARDIS to assist us in verifying determinism. Despite our familiarity with both parallel programming in general and RB-DPR in particular, TARDIS still identified 5 nondeterminism conflicts.
Eight of our applications are from standard parallel benchmark suites: PARSEC (Bienia et al., 2008), the Java Grande Forum (JGF) (Smith et al., 2001) and the Problem Based Benchmark Suite (PBBS) (Shun et al., 2012). These are listed in Table 5.1. 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 (Dwyer, 1987). 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](http://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.

**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 RB-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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Source</th>
<th>AC ops</th>
<th>Input set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackscholes</td>
<td>PARSEC</td>
<td>No</td>
<td>simlarge</td>
</tr>
<tr>
<td>Swaptions</td>
<td>PARSEC</td>
<td>No</td>
<td>simsmall</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>PARSEC</td>
<td>No</td>
<td>simsmall</td>
</tr>
<tr>
<td>Series</td>
<td>JGF</td>
<td>No</td>
<td>A (small)</td>
</tr>
<tr>
<td>Crypt</td>
<td>JGF</td>
<td>No</td>
<td>A (small)</td>
</tr>
<tr>
<td>SparseMatMult</td>
<td>JGF</td>
<td>No</td>
<td>A (small)</td>
</tr>
<tr>
<td>SOR</td>
<td>JGF</td>
<td>No</td>
<td>A (small)</td>
</tr>
<tr>
<td>BFS</td>
<td>PBBS</td>
<td>Yes</td>
<td>rMat, n = 100,000</td>
</tr>
</tbody>
</table>

Table 5.1: Workloads from benchmark suites.
5.2 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 (Feng and Leiserson [1997]) and SPD3 (Raman et al. [2012]) 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 4.2. In our reimplementation of SPD3, the shadow area of each object is protected by a sequence lock to accommodate concurrent access and potential resizing. 

Results appear in Figure 5.1. We tested all workloads with 1, 2, 4, 8 and 16 threads. All data were collected with a maximum JVM heap size of 4GB. 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.

1The newer, commutative-operation version of SPD3 (Westbrook et al. [2014]) was not available in time to incorporate in our experiments.
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, 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 $23 \times$. Cache misses caused by the large working set slow down the execution with 8 and 16 threads.

5.3 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 5.2.
Figure 5.1: 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 5.2: 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.
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.

5.4 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 intensive 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 5.3, 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.

5.5 Summary

In our performance evaluation, we observed that RB-DPR can generate significant (4–6×) speedup on top of our evaluation platform. The overhead of the run-time determinism checker, TARDIS, is about 2–4× for general detection, and 4–7× for detailed conflict information. Our implementation also works with accelerated application kernels, and can act as a simple parallel scripting infrastructure to “glue” those pre-built kernels.
6 Conclusion and Future Work

6.1 Conclusion

In this thesis, we have argued that parallel programming can be simple and efficient. To prove this claim, we introduced a deterministic parallel scripting language, DPR, its implementation, RB-DPR, and a dynamic determinism checker, TARDIS. DPR’s deterministic parallel constructs are based on the formal framework of deterministic parallel programming, proposed in Chapter 2. DPR inherits and extends the simplicity of the Ruby scripting language. In our experience, DPR’s parallel constructs are useful tools to either build parallel programs from scratch or to parallelize existing sequential programs. RB-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 read/write operations and high-level user-defined operations. In comparable cases (data race detection), TARDIS often outperforms existing state-of-the-art data race 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.

6.2 Future Work

The research introduced in this thesis, together with the open-sourced RB-DPR implementation, reveals multiple exciting future research directions. Topics for future work include:

Parallel scripting performance Although DPR displays significant parallel speedups on some workloads, 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 are more fundamental.

In practice, it would be appealing to implement DPR and TARDIS on top of a faster and/or more “scalable” language virtual machine. The JRuby+Truffle project (Seaton, 2014) reveals promising sequential performance improvement over conventional JRuby. Currently, the project is being integrated into the JRuby codebase. There is no foreseeable obstacle that prevents JRuby+Truffle working with RB-DPR. In the future, it would be possible to fully integrate DPR, TARDIS with JRuby+Truffle, providing a faster language virtual machine for DPR, while preserving the dynamic determinism checking feature.

There are challenges, however, for integrating DPR with an improved language virtual machine, like Truffle+JRuby. It is possible that a language VM is fast with sequential programs but has insignificant speedup on parallel systems. We observed this problem with the Ruby MRI YARV virtual machine (Ruby 1.9).
fundamental solution to this problem is to deeply understand the barriers that have significant negative impact on a VM’s parallel performance. One needs to distinguish if the barrier comes from the language design, its dynamic features, or only its implementation.

**Additional language constructs** In general, it would be interesting to explore more parallel language constructs, both deterministic and non-deterministic. Note that the concept of “determinism” is not the ultimate goal for parallel language design, but an appealing property for a simple parallel language. In some use cases, it is irrelevant if there are some changes in program output (on the same input). This provides potential chances to introduce non-deterministic constructs into DPR. However, programmers would also expect the nondeterminism to be “restricted” or “structured”, so that the difficulty brought in by nondeterminism is controllable. One may consider some “structured” nondeterministic mechanisms, such as arbitrary choice and atomic asynchronous events (e.g., for GUI input handling). 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. Chaotic relaxation, irregular (graph) algorithms and better support for streaming are all possible directions to further improve DPR.

**Implementations on large scale systems** Some parallel constructs of DPR fit well with existing large-scale cluster infrastructure software: the unordered iterators, together with reductions and AC ops, may be a good fit, for example, for map/reduce run-time systems. The pipeline construct may fit well in streaming systems. It would be interesting to explore the use of DPR on top of existing map/reduce or streaming systems, such as Apache Hadoop (Hadoop, 2005) or Storm (Storm, 2011). This would allow programmers to directly program the un-
derlying cluster infrastructure by using a simple scripting language, with intuitive deterministc parallel constructs and dynamic determinism checking.

**Better support of AC Ops** As noted in Section 3.3.2 a practical approach to annotate commutativity is useful. We designed the current annotation interface for AC ops according to our experience in annotating built-in Ruby libraries, such as maps and sets. To build even more concurrent libraries, an AC op interface that is more convenient, or more expressive, is required. It may be possible to provide interfaces that let programmers directly “program commutativity relationships”. For example, it may be useful to organize AC ops by putting them into groups and then allowing programmers to specify relationships among groups. Or, to further reduce the burden of the programmers, the language runtime may infer commutativity relationships from known patterns or existing commutativity relationships.

On the verification side, 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.

**Improving the TARDIS implementation** In our performance evaluation, we noticed that the slowdown of our TARDIS implementation is less than 4×. It would be very appealing to further reduce this overhead. Once the run-time overhead is well controlled, it might be possible to make TARDIS an always-on mechanism, checking every program on the VM. This would significantly improve the practicability of the dynamic checker. Efforts on several fronts may help to achieve this goal:

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

**Access set representation** Experiments have shown that TARDIS’s performance is sensitive to the memory consumption of task histories. It is possible to explore more space-efficient set representations, as well as adaptive techniques more sophisticated than simply switching to a hash table on list overflow.

**Out-of-band nondeterminism checking** 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.

To sum up, we believe that combining scripting languages and parallel programming is a viable approach toward simpler and more efficient parallel programming languages. Other future work directions based on this research are certainly possible.
Bibliography


Adve, Vikram, Luis Ceze, and Bryan Ford. 2011. The Second Workshop on Determinism and Correctness in Parallel Programming. Newport Beach, California, USA.


Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP ’95, pages 207–216. Santa Barbara, California, USA.


Bocchino, Robert and Joseph Devietti. 2013. The Fourth Workshop on Determinism and Correctness in Parallel Programming. Houston, Texas, USA.


SIGOPS Workshop on Parallel and Distributed Debugging, PADD ’88, pages 89–99. Madison, Wisconsin, USA.


Guy L Steele, Jr and Vijay A Saraswat. 2009. Workshop on Curricula for Concurrency. Orlando, Florida, USA.


LeBlanc, Thomas J. and John M. Mellor-Crummey. 1987. Debugging Parallel


Midkiff, Sam, Vijay Pai, and Deborah Bennett. 2011. Workshop on Integrating Parallelism Throughout the Undergraduate Computing Curriculum. San Antonio, Texas, USA.


and Practice of Parallel Programming, PPoPP ’03, pages 167–178. San Diego, California, USA.


Shavit, Nir. 2009. Workshop on Directions in Multicore Programming Education. Washington, DC, USA.


Veeraraghavan, Kaushik, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. DoublePlay: Par-


'Symposium on Code Generation and Optimization', CGO '11, pages 201–212. Seattle, Washington, USA.
