Systematic and Scalable Testing of Concurrent Programs

Jiří Šimša

CMU-CS-13-133

December 16th, 2013

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Thesis Committee:
Randal E. Bryant, Co-chair
Garth A. Gibson, Co-chair
David G. Andersen
Andre Platzer
Junfeng Yang (Columbia University)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the U.S. Army Research Office under award number W911NF-09-1-0273, the U.S. Department of Energy under award number DE-FC02-06ER25767 (PDSI), the National Science Foundation under awards number CCF-1019104 and CNS-1042543, the U.S. Department of State through the International Fulbright Science and Technology Award, the MSR-CMU Center for Computational Thinking, the Intel Science and Technology Center for Cloud Computing, and members of the CMU PDL Consortium: Actifio, American Power Conversion, EMC Corporation, Facebook, Fusion-io, Google Hewlett-Packard Labs, Hitachi, Huawei Technologies Co., Intel Corporation, Microsoft Research, NEC Laboratories, NetApp, Inc., Oracle Corporation, Panasas, Samsung Information Systems America, Seagate Technology, Symantec Corporation, VMware, Inc., and Western Digital.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Concurrent Programming, Systematic Testing, State Space Explosion, State Space Exploration, State Space Estimation, Parallelization, State Space Reduction

For my parents.

Abstract

The challenge this thesis addresses is to speed up the development of concurrent programs by increasing the efficiency with which concurrent programs can be tested and consequently evolved. The goal of this thesis is to generate methods and tools that help software engineers increase confidence in the correct operation of their programs. To achieve this goal, this thesis advocates testing of concurrent software using a systematic approach capable of enumerating possible executions of a concurrent program.

The practicality of the *systematic testing* approach is demonstrated by presenting a novel software infrastructure that repeatedly executes a program test, controlling the order in which concurrent events happen so that different behaviors can be explored across different test executions. By doing so, systematic testing circumvents the limitations of traditional ad-hoc testing, which relies on chance to discover concurrency errors.

However, the idea of systematic testing alone does not quite solve the problem of concurrent software testing. The combinatorial nature of the number of ways in which concurrent events of a program can execute causes an explosion of the number of possible interleavings of these events, a problem referred to as *state space explosion*.

To address the state space explosion problem, this thesis studies techniques for quantifying the extent of state space explosion and explores several directions for mitigating state space explosion: parallel state space exploration, restricted runtime scheduling, and abstraction reduction. In the course of its research exploration, this thesis pushes the practical limits of systematic testing by orders of magnitude, scaling systematic testing to real-world programs of unprecedented complexity.

Acknowledgments

When I came to Carnegie Mellon University in 2007, my life changed more than I could have imagined. Compared to my previous studies in the Czech Republic, the classes were harder, my competition was smarter, and there was always more work to be done. The first couple of years at CMU were quite challenging and I believe that if it had not been for the support and advice of my advisors, Randy Bryant and Garth Gibson, it is very unlikely I would be writing this acknowledgment.

Having said that, Randy and Garth have done more than help me endure the rocky beginning of my Ph.D. journey. Over the course of the past six years, they have become my role models and their high standards, research vision, and professional humility made me the researcher I am today. I am extremely thankful for their unwavering support and for the advice they have imparted throughout the years.

This thesis would not be the same without my thesis committee. Consequently, I would like to thank professors David Andresen, Andre Platzer, and Junfeng Yang for not hesitating to serve on the committee and for providing me with constructive feedback.

The motivation for this thesis and the technical prowess required for its implementation stems in part from my internships at Microsoft Research, Argonne National Labs, and Google. I would like to thank the individuals with whom I have interacted for their mentorship and insightful technical feedback: Walfredo Cirne, Byron Cook, Jason Hickey, Robert Kennedy, Sam Lang, Dushyanth Narayanan, David Oppenheimer, Rob Ross, Satnam Singh, Eno Thereska, Todd Wang, and John Wilkes.

In addition to my advisors, thesis committee members, and internship mentors, I have been very fortunate to collaborate with a number of exceptional individuals and I would like to acknowledge all of them: Luis Von Ahn, Ben Blum, Heming Cui, Ashutosh Gupta, Hao Li, Yi-Hong Lin, Stephen Magill, Milo Polte, Andrey Rybalchenko, Jennifer Tam, Wittawat Tantisiriroj, Viktor Vafeiadis, and Xinan Xu.

I would also like to thank my fellow students with whom I have shared the ups and downs of the life of a Ph.D. student and exchanged conversations about everything ranging from research to sports to philosophy: Michael Abd-El-Malek, Michael Dinitz, Sam Ganzfried, José Pablo González-Brenes, Alex Grubb, Mladen Kolar, Jayant Krishnamurthy, Elie Krevat, Daniel Lee, Iulian Moraru, Abraham Othman, Ippokratis Pandis, Swapnil Patil, Amar

Phanishayee, Jeffrey Stylos, Vijay Vasudevan, Kevin Waugh, Dan Wendlandt, Lin Xiao, Erik Zawadski, and Noam Zielberger.

In the course of my research career, I have been a member of two special families: the Parallel and Distributed Systems Laboratory (ParaDiSe) at Masaryk University and the Parallel Data Laboratory (PDL) at Carnegie Mellon University. I am thankful to ParaDiSe for providing me with the initial training to carry out computer science research and for that I would like to acknowledge Jiří Barnat, Luboš Brim, Ivana Černá, Mojmír Křetínský, Pavel Moravec, Radek Pelánek, and Jan Strejček. PDL has been instrumental in helping me to become a more seasoned researcher and to establish connections to technical leaders in the industry. For that I would like to acknowledge the people that make PDL tick: William Courtright, Chuck Cranor, Joan Digney, Greg Ganger, Mitch Franzos, Karen Lindenfelser, and Michael Stroucken.

This list would not be complete without mentioning people I know outside of work who made me laugh and helped me remember not to take life too seriously. This includes my love Kjersti Cubberley, my past roommates: Katrina Chan, Brianna Kelly, Olyvya Molinar, Panagiotis Papasaikas, James Tolbert, Kalin Vasilev, and Nancy Wang, members of the European mafia in Pittsburgh: Cagri Cinkilic, Roxana and Vlad Gheorghiu, Kyriaki Levanti, Marina Moraiti, Panayiotis Neophytou, Michael Papamichael, Maria Tomprou, Ceren Tuzmen, Selen Uguroglu, Evangelos Vlachos, and Panagiotis Vouzis, my hipster friends: Amanda El-Tobgy, Nick "Gunther" Fedorek, Kate Lasky, Peter Leeman, Seth Nyer, Cyrus Omar, Cameron Scott, Tomasz Skowronski, and Dave Zak, and the lads from AC Mellon.

Finally, I would like to thank my parents Vladimír and Marie and my sister Radka for their unconditional love and support. In my culture, families are tight-knit and it is not common to pursue a career if it requires putting distance between yourself and your family. My family is no different. Yet, despite knowing that my studies abroad would decrease the number of opportunities to see one another, they supported my decision to leave home and continued to be there for me when I needed them the most. For that, I am deeply grateful.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Motivation and Scope

The sum of the computational power of the 500 most powerful computers in the world has been doubling every 15 months for the past 20 years reaching 223 Petaflop/s in 2013 (Figure 1.1). Notably, this trend has continued in spite of the silicon industry reaching the CPU frequency wall, an obstacle that has been overcome by massive parallelization and distribution of computation: from single processors to symmetric multiprocessing (SMP) to massive parallel processing (MPP) to cluster computers (Figure 1.2). To leverage the computational power of such architectures, computer programs have undergone a similar transition: from sequential to concurrent computation.

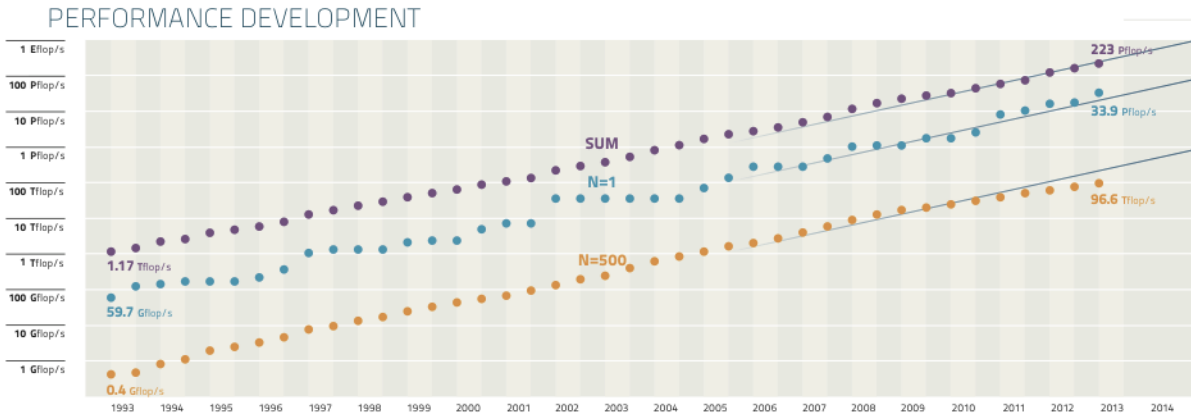


Figure 1.1: Evolution of Top500 Performance [102]

Unfortunately, the transition from sequential to concurrent programs unleashed not only the power of concurrent processing but also the terror of concurrency errors. Unlike sequential programs, whose behavior is determined by their inputs, concurrent programs behavior can additionally depend on the relative speed of concurrently executing threads. The concurrent nature of the computation leads to a combinatorial explosion of the number of ways in which concurrent events of a program can execute.

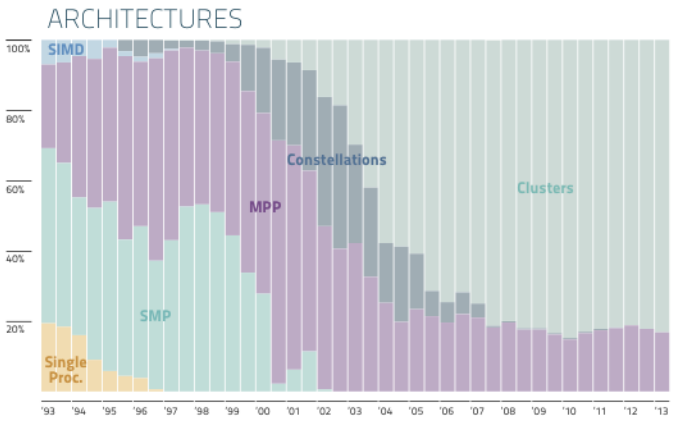


Figure 1.2: Evolution of Top500 Architectures [102]

This has proven a challenge for software engineers, who may overlook concurrency errors as they fail to fathom all scenarios in which their program can possibly execute.

Although software testing research has made great leaps forward, making use of techniques such as symbolic execution [83], it has struggled to generate methods and tools that would help software engineers build increasingly complex programs that keep up with the performance trends of new computing architectures. In particular, most of the existing concurrent software is still tested using ad-hoc methods, collectively referred to as *stress* testing [56], which rely on chance when it comes to discovering concurrency errors. Because of the absence of a high confidence testing mechanism, software engineers are reluctant to make changes to stable versions of their programs, slowing down the rate at which software evolves.

The challenge this thesis addresses is to speed up the development of concurrent programs by increasing the efficiency with which concurrent programs can be tested and consequently evolved. The goal of this thesis is to generate methods and tools that help software engineers increase confidence in the correct operation of their programs. To achieve this goal, this thesis advocates testing of concurrent software using a systematic approach capable of enumerating possible executions of a concurrent program.

1.2 Thesis Statement

"Existing concurrent software can be tested in a systematic and scalable fashion using testing infrastructure which controls nondeterminism and mitigates state space explosion."

On a theoretical level, given a program, one can view its set of possible behaviors as a random variable X and ad-hoc testing as a method for sampling observations from the probabilistic distribution of X. In case there exists an erroneous program behavior and ad-hoc testing executes concurrent events in a way that reveals the error with probability p, then, assuming independence of individual samples, ad-hoc testing

is expected to discover the erroneous behavior using p^{-1} samples. In other words, ad-hoc testing is good at detecting likely errors, but ineffective at discovering corner case errors that occur with very low probability. Further, if the probability distribution of the random variable X is uniform, then ad-hoc testing is expected to encounter all behaviors using $\mathcal{O}(n \log n)$ samples [22], where n represents the total number of possible program behaviors. Unfortunately, probabilistic distributions of real-world program behaviors are highly non-uniform, rendering ad-hoc testing an inefficient mechanism for exploring the state space of possible program behaviors.

This thesis focuses on an alternative approach to testing of concurrent programs, which avoids the aforementioned shortcomings of ad-hoc testing. This approach, known as *systematic* testing, controls the order in which concurrent events of a program happen in order to guarantee that different executions of the same test explore different interleavings of concurrent events. By doing so, systematic testing circumvents the limitations of ad-hoc testing. In particular, unlike ad-hoc testing, systematic testing is guaranteed to encounter all interleavings of concurrent events using n samples, where n represents the total number of possible interleavings of concurrent events.

However, reducing the number of samples required to enumerate all interleavings of concurrent events down to its theoretical minimum does not quite solve the problem of concurrent software testing. The combinatorial nature of the number of ways in which concurrent events of a program can execute causes an explosion of the number of possible interleavings of these events, a problem referred to as *state space explosion*. For a typical real-world program, the number of all possible interleavings of concurrent events can exceed the estimated number of atoms in the universe (10^{78} – 10^{82}), rendering exhaustive enumeration of all interleavings impossible.

To address the state space explosion problem in the context of concurrent software testing, this thesis first studies techniques for quantifying the extent of state space explosion and then explores several directions for mitigating state space explosion: parallel state space exploration, restricted runtime scheduling, and abstraction reduction. In the course of its research exploration, this thesis pushes the practical limits of systematic testing by orders of magnitude, scaling systematic testing to real-world programs of unprecedented complexity.

1.3 Thesis Contributions

This thesis makes several contributions. First, this thesis presents a novel software infrastructure for systematic testing of real-world programs and uses this infrastructure as a vehicle to drive forward its research on systematic testing. The infrastructure enables systematic testing of unmodified binaries of concurrent programs and implements state of the art algorithms for mitigating state space explosion.

Second, this thesis pioneers research on estimating the extent of state space explosion, a problem referred to as *state space estimation*. In particular, this thesis presents a number of techniques that create and refine an estimate of the number of thread interleavings

as the space of all possible thread interleavings is being explored. Further, this thesis demonstrates the benefits of state space estimation by using it to implement intelligent policies for allocation of scarce resources to a collection of systematic tests.

Third, this thesis demonstrates how to extend an inherently sequential state of the art algorithm for exploration of the space of all possible thread interleavings [51] to parallel execution on a large computational cluster. This effort results in a scalable version of the algorithm that achieves strong scaling on computer clusters consisting of up to one thousand of machines, speeding up the exploration of a set of thread interleavings by several orders of magnitude.

Fourth, this thesis explores integration of systematic testing with techniques for *restricted runtime scheduling*. In particular, this thesis presents an eco-system created by integrating its software infrastructure with a novel deterministic and stable multithreading thread runtime [39], which reduces the number of possible thread interleavings. There is synergy between systematic testing and restricted runtime scheduling: systematic testing helps to check the set of thread interleavings that are allowed by the restricted scheduler, while the restricted scheduler reduces the number of thread interleavings systematic testing needs to check.

Fifth, this thesis demonstrates how modeling program behavior at higher levels of abstraction, referred to as *abstraction reduction*, mitigates the extent of state space explosion. In general, the granularity of events interleaved by a thread interleaving can range from machine instructions to high-level function calls. Abstraction reduction mimics the behavior of software engineers that trust implementations of common coordination and communication libraries, such as POSIX threads (`pthreads`) [120] or Message Passing Interface (MPI) [103]. In particular, abstraction reduction treats common coordination and communication events as atomic, shifting the focus of systematic testing from testing the internals of common coordination and communication libraries to testing the programs that use them.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 covers both the theoretical and practical foundations this thesis builds upon. Chapter 3 presents the software infrastructure developed in support of the research carried out by this thesis. Chapter 4 presents the design, implementation, and evaluation of mechanisms for state space estimation and resource allocation policies based on state space estimates. Chapter 5 presents the design, implementation, and evaluation of an algorithm for parallel exploration of a set of thread interleavings. Chapter 6 presents the design, implementation, and evaluation of an eco-system formed by integrating our systematic testing infrastructure with a restricted runtime scheduling system. Chapter 7 presents the design, implementation, and evaluation of an approach that leverages abstraction to mitigate state space explosion. Chapter 8 provides an overview of related work and Chapter 9 discusses future work and draws conclusions.

Chapter 2

Systematic Testing Background

This chapter first describes state of the art algorithms for systematic testing (§2.1), focusing on algorithms pertinent to systematic testing of concurrent programs, and then presents an overview of the evolution of tools for systematic testing of concurrent programs (§2.2).

2.1 Theory

To gain confidence in the correct operation of a concurrent program through testing, a software engineer must verify as many program behaviors as possible in the time allotted for testing. The first step towards effective testing of a concurrent program is the ability to evaluate different scenarios in which concurrent events of the program may occur. The next step is then achieving efficiency through automation of the testing process, which should avoid evaluation of redundant or infeasible outcomes and at the same time fully utilize the available computational resources.

Traditional approaches to testing concurrent programs fail to meet these criteria. Ad-hoc approaches such as stress testing [56] evaluate a program using a range of tests. The intent of a stress test, typically either a large collection of concurrent stimuli or a small one repeated many times, is to drive a program into as many different scenarios as possible. However, stress testing is not guaranteed to exercise all possible scenarios.

To achieve better coverage, researchers have developed and evolved exhaustive approaches such as theorem proving [31, 43, 49, 95, 125] or model checking [14, 33, 35, 72]. Although these approaches have the potential to provide guarantees about all possible scenarios, their practicality is limited by the false positives and false negatives generally introduced during modeling [9], the expert knowledge needed to prove theorems about the program [84], or the effort needed to create and verify accurate models of existing software and update these models as the software changes [24].

In parallel with the above exhaustive approaches, research in software verification [61, 71] have produced practical tools for *systematic* testing of concurrent software. Unlike stress testing, systematic testing is able to enumerate different scenarios and

unlike the above exhaustive approaches, systematic testing operates over the actual implementation, striking a balance between coverage and practicality.

The idea behind systematic testing is to systematically resolve nondeterminism, enumerating different scenarios the program under test could experience. The sources of nondeterminism can be divided into two broad categories: *scheduling* nondeterminism and *input* nondeterminism.

Scheduling nondeterminism arises in concurrent programs when the relative speed of concurrently executing threads of computation can affect the program behavior, while input nondeterminism arises in both sequential and concurrent programs when program behavior depends on program inputs. Systematic testing of scheduling nondeterminism and input nondeterminism are two orthogonal problems.

This thesis is concerned with systematic testing of scheduling nondeterminism and the theory presented in the remainder of this section assumes that scheduling is the only source of nondeterminism. Obviously, this is rarely the case for real-world programs whose behavior can depend on a plethora of inputs such as user data, file contents, or timing. Chapter 3 describes how our infrastructure for systematic testing of concurrent programs addresses input nondeterminism.

The remainder of this section gives an overview of stateful exploration [71, 105], stateless exploration [61], partial order reduction (POR) [60], and dynamic partial order reduction (DPOR) [51], state of the art algorithms for systematic testing of concurrent programs.

2.1.1 Stateful Exploration

Stateful exploration [12, 71, 105] is a technique that targets systematic testing of nondeterministic programs. The goal of stateful exploration is to explore the state space of possible program states by systematically resolving program nondeterminism, explicitly recording program states to avoid re-visitation and to guarantee completeness.

Algorithm 1 gives a high-level overview of stateful exploration. To keep track of the exploration progress, stateful exploration maintains two collections of program states. The *visited* collection records program states that have been visited, while the *reachable* collection records program states that are known to be reachable.

The program states contained in the *visited* and *reachable* collections are stored explicitly: for traditional hardware architectures, a program state thus consists of the relevant context stored in the hardware registers, main memory, and on disk. To navigate a program to a particular program state (Algorithm 1, Line 6), stateful exploration simply loads the previously stored context.

The CHILDREN(*node*) function (Algorithm 1, Line 7) resumes execution from the program state *node*. Once a nondeterministic choice is encountered, a set of program states corresponding to all possible outcomes of the choice is returned.

The main drawback of stateful exploration is that for real-world programs, explicitly storing the program states can require considerable space [29, 105, 165], which limits the size of the state space that can be fully explored. To mitigate this problem, researchers

Algorithm 1 StatefulExploration(*root*)

Require: *root* is the initial program state.
Ensure: All program states reachable from *root* are explored.

```
1: visited ← NewSet
2: reachable ← NewSet
3: Insert(root, reachable)
4: while visited ≠ reachable do
5:     node ← an arbitrary element of reachable \ visited
6:     navigate execution to node
7:     for all child ∈ Children(node) do
8:         if child ∉ reachable then
9:             Insert(child, reachable)
10:        end if
11:    end for
12:    Insert(node, visited)
13: end while
```

have devised a number of schemes such as compression and hashing [71], selective caching [13], or parallel processing [27].

In spite of the advances in stateful exploration, the majority of practical tools for systematic testing for concurrent programs [61, 82, 107, 160, 163] are based on an alternative approach called stateless exploration.

2.1.2 Stateless Exploration

Stateless exploration [61] is a technique that targets systematic testing, particularly suitable for systematic testing of concurrent programs. Although stateless exploration pursues the same goal as stateful exploration, stateless exploration does not store program states explicitly. Instead, stateless exploration represents a program state implicitly using the sequence of nondeterministic choices that lead to the program state from the initial program state.

To keep track of the exploration progress, stateless exploration abstractly represents the state space of different program states using an *execution tree*. Nodes of the execution tree represent nondeterministic choices and edges of the execution tree represent program state transitions. A path from the root of the tree to a leaf then uniquely encodes a program execution as a sequence of nondeterministic choices.

Abstractly, enumerating the branches of the execution tree corresponds to an enumeration of different sequences of program state transitions. Notably, the set of explored branches of a partially explored execution tree identifies which sequences of program state transitions have been explored. Further, the information collected by past executions can be used to generate schedules that describe in what order to sequence program state transitions of future executions in order to explore new parts of the state space.

Algorithm 2 StatelessExploration(*root*)

Require: *root* is the initial program state.
Ensure: All program states reachable from *root* are explored.

```
1:  frontier ← NewStack
2:  Push({root},frontier)
3:  while frontier not empty do
4:      while Top(frontier) not empty do
5:          node ← an arbitrary element of Top(frontier)
6:          remove node from Top(frontier)
7:          navigate execution to node
8:          if Children(node) not empty then
9:              Push(Children(node),frontier)
10:         end if
11:     end while
12:     Pop(frontier)
13: end while
```

Algorithm 2 gives a high-level overview of stateless exploration. The algorithm maintains an exploration *frontier*, represented as a stack of sets of nodes, and uses depth-first search to explore the execution tree. Depth-first search is used to achieve space complexity that is linear in the depth of the execution tree.

To navigate a program to a particular program state (Algorithm 2, Line 7), stateless exploration recreates the initial program state and then carries out the sequence of nondeterministic choices identified by the execution tree. The Children(*node*) function (Algorithm 2, Line 8) behaves the same as before.

The main drawback of stateless exploration is that different sequences of nondeterministic choices can represent identical concrete program states, resulting in redundant exploration. To address this problem, stateless exploration is typically augmented with state space reduction techniques [51, 60].

2.1.3 Partial Order Reduction

Partial order reduction (POR) [60] is a technique that targets efficient systematic testing of concurrent programs. The goal of POR is to reduce the inefficiency of state space exploration resulting from exploration of equivalent program states.

Algorithm 3 gives a high-level overview of stateless exploration based on partial order reduction. Similar to stateless exploration, the partial order reduction algorithm maintains an exploration *frontier*, represented as a stack of sets of nodes, and uses depth-first search to explore the execution tree.

The PersistentSet(*node*) function (Algorithm 3, Line 3) uses static analysis to identify what subtrees of the execution tree need to be explored. In particular, it inputs a node of the execution tree and outputs a subset of the Children(*node*) set that needs

Algorithm 3 PARTIALORDERREDUCTION(*root*)

Require: *root* is the initial program state.
Ensure: All non-equivalent program states reachable from *root* are explored.

```
1: procedure DepthFirstSearch(root, frontier)
2:     remove node from Top(frontier)
3:     if PersistentSet(node) not empty then
4:         Push(PersistentSet(node), frontier)
5:         for all child ∈ Top(frontier) do
6:             navigate execution to child
7:             DepthFirstSearch(child, frontier)
8:         end for
9:         Pop(frontier)
10:    end if
11: end procedure
12: frontier ← NewStack
13: Push({root}, frontier)
14: DepthFirstSearch(root, frontier)
```

to be explored in order to explore all *non-equivalent* sequences of program states; two sequences of program states are considered equivalent, if using them to evaluate a program property yields the same result. Further details behind the computation of the PERSISTENTSET function are unnecessary for understanding the content of this thesis are omitted for brevity. The interested reader is referred to Godefroid's seminal treatment [60].

The main drawback of partial order reduction is that static analysis of complex programs is often costly or infeasible and results in larger than necessary persistent sets. To address this problem, static analysis can be replaced with dynamic analysis [51].

2.1.4 Dynamic Partial Order Reduction

Dynamic partial order reduction (DPOR) [51] is a technique that targets efficient state space exploration. The goal of DPOR is to reduce the inefficiency of partial order reduction resulting from the use of static analysis. Namely, when stateless exploration explores the execution tree, DPOR relies on dynamic analysis to decide how to augment the existing exploration frontier.

Algorithm 4 gives a high-level overview of stateless exploration based on dynamic partial order reduction. Similar to stateless exploration, the dynamic partial order reduction algorithm maintains an exploration *frontier*, represented as a stack of sets of nodes, and uses depth-first search to explore the execution tree.

The UPDATEFRONTIER(*frontier*, *node*) function (Algorithm 4, Line 3) uses dynamic analysis to identify which subtrees of the execution tree need to be explored. In particular, the function inputs the current exploration frontier and the current node and

Algorithm 4 DynamicPartialOrderReduction(*root*)

Require: *root* is the initial program state.
Ensure: All non-equivalent program states reachable from *root* are explored.

```
 1: procedure DepthFirstSearch(root, frontier)
 2:     remove node from Top(frontier)
 3:     UpdateFrontier(frontier, node)
 4:     if Children(node) not empty then
 5:         child ← arbitrary element of Children(node)
 6:         Push({child}, frontier)
 7:         for all child ∈ Top(frontier) do
 8:             navigate execution to child
 9:             DepthFirstSearch(child, frontier)
10:         end for
11:         Pop(frontier)
12:     end if
13: end procedure
14: frontier ← NewStack
15: Push({root}, frontier)
16: DepthFirstSearch(root, frontier)
```

computes the happens-before [88] and dependence [60] relations between the transitions on the path leading to the current node. This information is then used to infer which nodes need to be added to the exploration frontier in order to explore all *non-equivalent* sequences of program states. Unlike the PersistentSet function, the UpdateFrontier function can add nodes to an arbitrary set of the exploration frontier stack, an aspect of DPOR that prevents straightforward parallelization of the execution tree exploration (cf. Chapter 5). Further details of the computation of the UpdateFrontier function are unnecessary for understanding the content of this thesis are omitted for brevity. The interested reader is referred to the original DPOR paper [51].

2.2 Practice

Research in software verification has produced a number of tools for systematic testing of concurrent programs including VeriSoft [61, 62], CMC [105], FiSC [164, 165], eXplode [163], MaceMC [82], ISP [118, 148], CHESS [106, 107], MoDist [160], dBug [135, 136, 137], and ETA [138].

All of the these tools are built around the same idea: for a given concurrent program and its initial program state, systematically and automatically enumerate different scheduling scenarios searching for errors. What differentiates these tools is 1) the operating systems and programming languages they target, 2) the techniques they use for exploring and reducing the state space of possible scenarios, 3) the type of properties they check for, and 4) the effort needed to deploy the tool.

TOOL	OPERATING SYSTEM	PROGRAMMING LANGUAGE	STATE SPACE EXPLORATION	STATE SPACE REDUCTION	PROPERTIES CHECKED	DEPLOYMENT EFFORT
VeriSoft [61]	*	C and C++	stateless	POR	generic propositions	manual annotation
CMC [105]	*	C and C++	stateful	state hashing, abstraction	user-provided propositions	manual annotation
FiSC [165]	Linux	*	stateful	state hashing, search heuristics	generic propositions	manual annotation
eXplode [163]	*	*	stateless	state hashing	user-provided propositions	none
MaceMC [82]	*	Mace [81]	stateless	state hashing, search heuristics	user-provided temporal properties	none
ISP [118]	*	MPI [103]	stateless	DPOR	generic propositions	none
CHESS [107]	Windows	*	stateless	state hashing, context bounding [106]	generic propositions	none
MoDist [160]	Windows	*	stateless	DPOR, search heuristics	user-provided propositions	none
dBug [135]	POSIX [119]	*	stateless	DPOR	generic propositions	none
ETA [138]	*	Actors [2]	stateless	DPOR	user-provided temporal properties	none

Table 2.1: Overview of Systematic Testing Tools

Table 2.1 presents an overview of the key attributes of these tools. The TOOL column lists the tools in a chronological order. The OPERATING SYSTEM column identifies what operating systems, if any, is the tool restricted to. The asterisk character is used when the tool works independently of the operating system. The PROGRAMMING LANGUAGE column identifies what programming languages, if any, is the tool restricted to. The asterisk character is used when the tool works independently of the programming language. The STATE SPACE EXPLORATION column identifies what type of state space exploration the tool uses. The STATE SPACE REDUCTION column identifies what type of state space reduction the tool uses. The PROPERTIES CHECKED column identifies what properties the tool checks for. In particular, this column specifies if the tool checks for 1) user-provided propositions in addition to generic propositions and 2) temporal propositions [99] in addition to other propositions. Finally, the DEPLOYMENT EFFORT column identifies the effort needed to deployment the tool.

The remainder of this section offers a summary of all of these tools with the exception of dBug and ETA. The dBug and ETA tools were built by this thesis author in support of the research carried out by this thesis and are described in detail in Chapter 3.

2.2.1 VeriSoft

The first practical tool for systematic testing of concurrent programs – VeriSoft [61] – was developed by Patrice Godefroid in 1997. VeriSoft can explore different serializations of concurrent library function calls of multithreaded C and C++ programs and requires manual instrumentation of the program. Since its creation in 1997, VeriSoft has been successfully applied to prove safety properties and to find bugs in a number of programs ranging from critical components of a telephone switch [63] to complete releases of call-processing software [30].

2.2.2 CMC

In 2002, a research group at Stanford developed the C Model Checker (CMC) [105], which can explore different serializations of event handlers in C and C++ programs. CMC was the first systematic testing tool to introduce fault injection as a mechanism to test the error handling of corner case scenarios. The main conceptual difference between VeriSoft and CMC is the state space exploration technique. VeriSoft only stores the initial state and possibly re-explores parts of the state space, while CMC stores every intermediate state, which allows for faster state space traversal. Anecdotally, VeriSoft typically runs out of time, while CMC typically runs out of memory. The evaluation of CMC [105] checked three implementations of the AODV networking protocol [117], finding a total of 40 bugs.

2.2.3 FiSC

In 2004, the same group at Stanford used CMC as the foundation for building the FiSC tool [165] for systematic testing of Linux file systems. The key idea behind FiSC is to enumerate different ways in which concurrent file system operations can modify data and meta-data and simulate system crashes at arbitrary time points, searching for file system inconsistencies. Similar to CMC, FiSC uses a stateful approach, encoding each concrete state as a collection of hashed chunks of the concrete state in order to decrease memory footprint of each state. The evaluation of FiSC [165] checked three file systems – ext3, JFS, and ReiserFS – and found serious bugs in all of them, 32 in total.

2.2.4 eXplode

In 2006, the same group at Stanford created the eXplode tool [163] for systematic testing of arbitrary storage systems. Unlike FiSC, which inherited the stateful nature of its exploration from CMC, eXplode embraces stateless exploration, which "lead to orders of magnitude reduction in complexity and effort in using eXplode as opposed to FiSC". Notably, eXplode is able to check storage systems without requiring their source code. The extensive evaluation of eXplode [163] checked three version control systems, Berkeley DB [108], an NFS implementation [132], ten file systems, a RAID system [116], and the VMware GSX virtual machine, finding bugs in all of them, 36 in total.

2.2.5 MaceMC

In 2007, a group at UCSD created the programming language Mace [81], geared towards designing distributed programs, and the systematic testing tool MaceMC [82], for verifying Mace designs. The Mace programming language groups events into Mace transitions and MaceMC then uses this abstraction to systematically explore different serializations of concurrent transitions. A unique feature of MaceMC enabled by the co-design of a programming language and a testing tool is that it can check for both safety and bounded liveness properties. A number of distributed programs including peer-to-peer systems Chord [145] and Pastry [126] were re-implemented in Mace. The evaluation of MaceMC [82] checked Mace implementations of four different programs and found a total of 51 bugs.

2.2.6 ISP

In 2007, a collaboration between the University of Utah and Argonne National Laboratories was started that eventually resulted in the ISP tool [118, 147] for systematic testing of MPI programs [103]. The tool replaces key MPI communication primitives with wrappers that are used to control the order in which MPI primitives execute. The systematic testing is deployed via a link-time interposition. The evaluation of ISP [118, 147] checked four MPI programs and found bugs in all of them.

2.2.7 CHESS

In 2008, researchers from Microsoft Research Redmond designed the CHESS tool [107], which explores different serializations of memory accesses in a multithreaded program. CHESS uses run-time interposition and thus does not require any source code modifications. This feature greatly simplifies the integration of CHESS into the testing process. The evaluation of CHESS [107] checked eight different programs ranging from process management libraries to a distributed execution engine to a research operating system, finding bugs in all of them, 27 in total.

2.2.8 MoDist

In 2009, a combined effort of researchers from research labs and universities in both the United States and China lead to the creation of the systematic testing tool MoDist [160]. The MoDist project combined ideas from previous work to deliver a tool capable of systematic exploration of different serializations of concurrent system calls in multithreaded and distributed Windows programs. Similar to CHESS, MoDist does not require source code modifications. However, instead of using run-time interposition, MoDist relies on binary instrumentation of Windows API provided by Box [68]. MoDist evaluation [160] checked three distributed programs – the replication extension of Berkeley DB [108], a production implementation of the Paxos protocol [89], and a prototype of a primary-backup replication protocol – and found a total of 35 bugs.

This list of success stories demonstrates that there is momentum in advancing systematic testing of concurrent programs and provides evidence that systematic testing tools are effective at finding errors. The success of the above tools stems from sophisticated techniques for mitigating the combinatorial explosion of the number of possible program states of a concurrent program.

Interestingly, none of the previous work actually attempts to measure how bad the state space explosion problem actually is. In cases where systematic testing fails to exhaust the state space, the absence of a state space size estimate poses a challenge to quantifying the benefits of the test results. To address this problem, Chapter 4 describes and evaluates techniques for estimating the size of the stateless exploration space. Our evaluation on a wide range of concurrent programs provides evidence of the extent of state space explosion.

The rest of the thesis then presents research that aims to push the practical limits of systematic testing. Chapter 5 presents a novel algorithm [139] for state space exploration that scales extremely well, enabling stateless exploration for large computer clusters. Chapters 6 and 7 show how restricted runtime scheduling [39] and abstraction [38] help to alleviate the state space explosion problem.

Chapter 3

Systematic Testing Infrastructure

This chapter describes two tools built in support of the research carried out by this thesis. The ETA tool (§3.1) targets systematic testing of scheduling nondeterminism in multithreaded components of the Omega cluster management system [129], while the dBug tool (§3.2) targets systematic testing of scheduling nondeterminism in multithreaded and distributed programs of POSIX-compliant operating systems [119].

To enable systematic testing of scheduling nondeterminism, both dBug and ETA need to address common challenges. Consequently, the presentation of ETA and dBug follows a common pattern. First, the presentation introduces a formal model used for representing executions of target programs. The reason for defining these models is to provide a formal framework for describing the happens-before [88] and dependence [60] relations between program state transitions. These relations are computed by both tools in order to enable efficient state space exploration based on dynamic partial order reduction [51]. Second, the presentation discusses the mechanisms the tools use to control input and scheduling nondeterminism and implement state space exploration. Finally, the presentation highlights notable aspect of the respective implementations.

3.1 ETA

This section describes ETA [138], a tool for systematic testing of multithreaded components of the Omega cluster management system [129]. In particular, §3.1.1 defines a model for representing multithreaded components of Omega, which embrace the actors programming paradigm [2], and §3.1.2 describes ETA's implementation and its integration with pre-existing testing infrastructure.

3.1.1 Program Model

Conceptually, an *actor program* consists of a set of actors, each of which has its own private state, a public queue for receiving messages, and a set of handlers for processing of queued messages. The only way two actors can communicate is by sending messages

to one another. Further, each individual actor is sequential, repeatedly invoking handlers to process queued messages. Processing of a message is assumed never to block indefinitely and can both modify the private state of the actor processing the message and push new messages on queues of other actors. An actor program realizes parallelism by concurrently executing handlers of different actors. The following definition formalizes the notion of an actor program.

Definition 3.1.1. An *n-actor program* $\mathbb{M}$ is a tuple of actors $(A_1, \ldots, A_n)$, where for $i \in \{1, \ldots, n\}$ an *actor* A_i is in turn a tuple (Q_i, L_i, Δ_i) consisting of a set Q_i of *queue states*, a set L_i of *local states*, and a *transition function* $\Delta_i : \mathit{Messages} \times L_i \to (Q_1 \times \ldots \times Q_n) \times L_i$. A queue state $q_i \in Q_i$ is a finite sequence $(m_1, \ldots, m_k) \in \mathit{Messages}^k$ of *messages*.

Note that the above definition intentionally leaves several terms undefined. The set *Messages* is assumed to be an abstract representation of the set of messages actors use to communicate. In particular, the inputs and outputs of the actor program are encoded as messages. The sets $L_1, \ldots, L_n$ are assumed to be an abstract representation of the local state of each actor and the transition functions $\Delta_1, \ldots, \Delta_n$ are assumed to be an abstract representation of the handlers invoked to process queued messages.

Next, a state of an actor program is defined as a vector of pairs of local and queue states belonging to the actors of the actor program.

Definition 3.1.2. Let $\mathbb{M}$ be an n-actor program. A *state* of $\mathbb{M}$ is defined as a tuple $((q_1, l_1), \ldots, (q_n, l_n)) \in (Q_1 \times L_1) \times \ldots \times (Q_n \times L_n)$. Further, a state of $\mathbb{M}$ can be inspected using the following projections:

- $\mathit{actor} : ((Q_1 \times L_1) \times \ldots \times (Q_n \times L_n)) \times \{1, \ldots, n\} \to \bigcup_{i=1}^{n} (Q_i \times L_i)$, a function such that $\mathit{actor}(((q_1, l_1), \ldots, (q_n, l_n)), i) = (q_i, l_i)$
- $\mathit{queue} : \bigcup_{i=1}^{n} (Q_i \times L_i) \to \bigcup_{i=1}^{n} Q_i$, a function such that $\mathit{queue}(q_i, l_i) = q_i$
- $\mathit{local} : \bigcup_{i=1}^{n} (Q_i \times L_i) \to \bigcup_{i=1}^{n} L_i$, a function such that $\mathit{local}(q_i, l_i) = l_i$

Next, the transition function of an actor program is defined as a combination of the transition functions of the actor program actors. In particular, the formal model of an actor program transitions between states by choosing an actor with a non-empty queue, removing a message from the queue of that actor, and invoking the corresponding message handler. By the nature of the transition function, the invoked message handler can append messages to a message queue of any actor, but it can only change the local state of the actor invoking the handler.

Definition 3.1.3. Let $\mathbb{M}$ be an n-actor program. The *transition function* of $\mathbb{M}$ is a function $\Delta_{\mathbb{M}} : (Q_1 \times L_1) \times \ldots \times (Q_n \times L_n) \times \{1, \ldots, n\} \to \{\bot\} \cup ((Q_1 \times L_1) \times \ldots \times (Q_n \times L_n))$ such that $\Delta_{\mathbb{M}}((q_1, l_1), \ldots, (q_n, l_n), i)$ equals $\bot$ if q_i is empty. Otherwise, $\Delta_{\mathbb{M}}((q_1, l_1), \ldots, (q_n, l_n), i)$ equals $((q_1 \circ q_1', l_1), \ldots, (\mathit{tail}(q_i) \circ q_i', l_i'), \ldots, (q_n \circ q_n', l_n))$ where $q_1', \ldots, q_n'$ and l_i' are such that $\Delta_i(\mathit{head}(q_i), l_i) = ((q_1', \ldots, q_n'), l_i')$.

Next, an execution of an actor program is defined as a sequence of states. Further, this sequence is identified with 1) an index sequence that encodes the order in which actors process messages along the execution and 2) the sequence of messages processed along the execution.

Definition 3.1.4. Let $\mathbb{M}$ be an n-actor program, $\Delta_{\mathbb{M}}$ its transition function, and s its state. An *execution* of $\mathbb{M}$ from s is a finite sequence $\alpha = (s_0, \ldots, s_k)$ of states of $\mathbb{M}$ such that $s = s_0$ and there exists an *index sequence* $I(\alpha) = (i_0, \ldots, i_{k-1}) \in \{1, \ldots, n\}^k$ such that $s_{j+1} = \Delta_{\mathbb{M}}(s_j, i_j)$ for all $j \in \{0, \ldots, k-1\}$. Further, the *message sequence* $M(\alpha) = (m_0, \ldots, m_{k-1}) \in \mathit{Messages}^k$ is a sequence of messages such that $m_j = \mathit{head}(\mathit{queue}(\mathit{actor}(s_j, i_j)))$ for $j \in \{0, \ldots, k-1\}$ and $M(\alpha)$ is used to denote the set $\{m_0, \ldots, m_{k-1}\}$.

Next, the happens-before [88] and dependence relations [60] are defined to track causality and interactions between messages. These relations are necessary in order to adapt the dynamic partial order reduction algorithm (DPOR) [51] to actor programs. In other words, the happens-before relation is a transitive closure of the causal order in which 1) an actor processes messages, 2) a message handler creates new messages.

Definition 3.1.5. Let $\alpha = (s_0, \ldots, s_k)$ be an execution, $I(\alpha) = (i_0, \ldots, i_{k-1})$ its index sequence, and $M(\alpha) = (m_0, \ldots, m_{k-1})$ its message sequence. The *happens-before* relation $\rightsquigarrow_\alpha \subseteq M(\alpha) \times M(\alpha)$ is the smallest relation that satisfies the following conditions:

- for all $j, l \in \{0, \ldots, k-1\}$: if $j < l$ and $i_j = i_l$, then $m_j \rightsquigarrow_\alpha m_l$
- for all $j \in \{0, \ldots, k-1\}$: if $\Delta_{i_j}(m_j, \mathit{local}(\mathit{actor}(s_j, i_j))) = ((q'_1, \ldots, q'_n), l')$, then $m_j \rightsquigarrow_\alpha m'_j$ for all $m'_j \in \bigcup_{i=1}^{n} q'_i$
- for all $j_1, j_2, j_3 \in \{0, \ldots, k-1\}$: if $m_{j_1} \rightsquigarrow_\alpha m_{j_2}$ and $m_{j_2} \rightsquigarrow_\alpha m_{j_3}$, then $m_{j_1} \rightsquigarrow_\alpha m_{j_3}$

Definition 3.1.6. Let $\alpha = (s_0, \ldots, s_k)$ be an execution, $I(\alpha) = (i_0, \ldots, i_{k-1})$ its index sequence, and $M(\alpha) = (m_0, \ldots, m_{k-1})$ its message sequence. A *dependence* relation is a relation $D_\alpha \subseteq M(\alpha) \times M(\alpha)$ such that for all $j_1, j_2 \in \{0, \ldots, k-1\}$: if there exists $i \in \{1, \ldots, n\}$ such that $\Delta_{i_{j_1}}(m_{j_1}, \mathit{local}(\mathit{actor}(s_{j_1}, i_{j_1}))) = ((q_1, \ldots, q_n), l)$, $\Delta_{i_{j_2}}(m_{j_2}, \mathit{local}(\mathit{actor}(s_{j_2}, i_{j_2}))) = ((q'_1, \ldots, q'_n), l')$, and q_i and q'_i are a non-empty sequence, then $D_\alpha(m_{j_1}, m_{j_2})$.

The motivation behind the definition of dependence is to identify pairs of messages with possibly non-commutative effects. In particular, if $D_\alpha(m_1, m_2)$, then in the course of the execution α, handling of messages m_1 and m_2 adds messages to overlapping sets of message queues and the order in which messages m_1 and m_2 are handled may affect the outcome of the actor program.

Note that the above definition of dependence produces a valid dependence relation [60] but not necessarily a minimal one. Thus, the dependence relation computed based on Definition 3.1.6 may be an over-approximation of the true dependence. In contrast to that, the happens-before relation computed based on Definition 3.1.5 may be an under-approximation of the true causality.

The nature of the DPOR algorithm justifies such approximations. The smaller the dependence relation is and the larger the happens-before relation is, the more states the DPOR algorithm can reduce. As long as the dependence relation does not fail to relate any messages that are in fact dependent, and the happens-before relation does not relate any messages that are in fact concurrent, the DPOR algorithm will work correctly.

In practice, one aims to strike a balance between accurate identification of dependence and causality and the overhead of doing so. The above definitions of the happens-before and dependence relations are crafted so that these relations can be easily computed at runtime, avoiding the need for static analysis or program annotations.

3.1.2 Implementation

The runtime of Omega actor programs controls the order in which messages are handled through an *actor manager*. In production, the actor manager is multithreaded, concurrently processing messages from different message queues. In contrast to that, when testing Omega's actor programs, a sequential actor manager that enables fine-grained control of message ordering is used instead. This approach to testing relies on the assumption that the actor managers used in production and in testing are functionally equivalent.

To enable systematic testing of Omega actor programs, ETA implements a new *exploratory* actor manager. Similar to the default actor manager for testing, the *exploratory* actor manager serializes the handling of concurrent messages. Unlike the default actor manager for testing, which is deterministic, different executions of the same actor program test under the exploratory actor manager can explore different message orders and program states.

Exploration

To explore different program states, ETA repeatedly executes the same test and uses stateless exploration to navigate the state space. In other words, ETA runs the actual actor program but, with the exception of the current program state, it does not store the program states explicitly. Instead, program states are represented implicitly using index sequences of executions that lead to them.

The program states revealed by test executions are recorded and stored in an *execution tree*, with nodes implicitly representing program states and edges representing index sequence elements. Initially, the exploration holds no knowledge about the structure of the execution tree and as new message sequences are explored, the execution tree is gradually unfolded.

An execution of an actor program test in ETA uses the execution tree to generate an index sequence which identifies a node of the execution tree exploration frontier to explore next. The exploratory actor manager then uses this index sequence to steer the execution towards this node. Once this node is reached, the exploratory actor manager switches to round-robin scheduling, recording its decisions. Once the test execution completes (or times out), ETA processes the explored message sequence, computing the happens-before relation (cf. Definition 3.1.5) and the dependence relation (cf. Definition 3.1.6), and uses the DPOR algorithm (cf. Section 2.1.4) to update the exploration frontier.

Nondeterminism

In order for ETA to navigate the execution tree, the message queue identifiers of the Omega actors runtime must be identical across different test executions (property of the actors library), and identical index sequences must produce identical message queues contents across different test executions (property of the actor program and its test).

To accomplish the latter, ETA needs to make sure that message ordering is indeed the only source of nondeterminism. Otherwise, the exploratory actor manager might not be able to deterministically replay previously explored message orders, which is necessary for stateless exploration. To meet this requirement, ETA uses deterministic seeds for pseudo-random number generators and deterministic mock implementations of nondeterministic components of the environment such as RPC servers.

The broader notion is to treat all nondeterminism as messages but only explore the ones the user considers most pertinent. This notion could be implemented as a more general exploratory manager that controls the different sources of nondeterminism. The main advantage of a more general exploratory manager is its ability to test interactions between the different sources of nondeterminism. The main disadvantage is the additional combinatorial explosion of the number of possible interactions, impeding the ability of the more general exploratory manager to investigate all interactions. Given the size of the state spaces of actor program tests in the Omega test suite (see Chapter 4), ETA focuses on message ordering only.

Deployment

The ultimate goal of systematic testing is to help software engineers to test their programs better. Keeping this goal in mind, ETA is designed to smoothly integrate with the infrastructure for testing Omega actor programs.

Typically, an Omega actor program test sets up some initial state and then repeatedly uses the Omega actors library API to either trigger message handling or to inspect the current program state for errors. The message handling triggers can instruct the actor manager to handle one message or to continue handling messages until a temporal property [99] becomes true (or there are no messages left).

To automate the testing process, Omega uses the Google Test framework [66]. This framework provides a number of test macros through which a software engineer describes a unit test and the framework then executes these tests and reports the results. To add support for systematic testing into the existing infrastructure for testing Omega, the Google Test framework was extended by this thesis author with new test macros, one for each original test macro, that can be used as drop-in replacements for the original Google Test macros. By default the new test macros behave identically to their Google Test counterparts, executing the body of the test macro once. However, if the test is executed with a special command-line flag, systematic testing is used, repeatedly executing the body of the test until the execution tree representing the state space of the test body is fully explored or a timeout is reached.

Runtime Estimation

For traditional tests, runtime can be estimated using a back of the envelope analysis based on the program design and hardware architecture. In contrast, the runtime of systematic tests depends largely on the interaction between 1) the combinatorial explosion of the number of ways in which concurrent events of the test can execute and 2) the realized state space reduction. This quality of systematic tests makes estimation of their runtime challenging. In fact, prior to ETA, no systematic testing tool had the capability to estimate the runtime of its tests.

The absence of runtime estimates leads to ad-hoc and possibly inefficient allocations of scarce testing resources. Recognizing this shortcoming of existing systematic testing tools, ETA implements a novel technique for runtime estimation (cf. Chapter 4). This technique predicts runtimes for systematic Omega tests and enables efficient allocation of testing resources.

Parallel State Space Exploration

In addition to the traditional sequential implementation of the DPOR algorithm, ETA implements a novel scalable version of the DPOR algorithm (cf. Chapter 5), which explores the state space of possible program states concurrently using a number of parallel test executions. Although concurrent exploration of the execution tree seems straightforward at first sight, a naive implementation may result in redundant exploration [167] and working out the details in the context of DPOR requires some care.

ETA's scalable implementation of the DPOR algorithm enables concurrent exploration of systematic Omega tests using Google data centers [65]. The technique achieves strong scaling and pushes the practical limits of systematic testing by orders of magnitude.

3.2 dBug

This section describes dBug [135, 137], a tool for systematic testing of multithreaded and distributed programs of POSIX-compliant operating systems [119]. Unlike ETA, dBug is publicly available [135] and is not restricted to a particular programming language. The motivation behind creating dBug is not only to provide a platform for carrying out systematic testing research, but also to provide a practical tool that can be used for systematic testing of a wide range of existing programs.

Similar to the presentation of ETA, the presentation of dBug first describes a formal framework used by dBug to model concurrent programs (§ 3.2.1) and then discusses notable aspects of dBug's design (§ 3.2.2) and implementation (§ 3.2.3).

3.2.1 Program Model

This subsection defines a formal framework for modeling real-world concurrent programs. Similar to the framework used by ETA, the purpose of this framework is to capture causality and dependence between program state transitions so that dBug can implement efficient state space exploration based on dynamic partial order reduction (DPOR) [51]. Unlike the framework used by ETA, the framework used by dBug is not tailored to a specific programming paradigm and is capable of modeling concurrent programs at different levels of abstraction.

Definition 3.2.1. An *abstract state* s is an element of the $S = G_S \times \mathcal{P}(\mathit{Identifiers} \times L_S)$ universe, where $G_S : G_O \to \mathit{Values}$ is the *global state* function that assigns values to *global objects* G_O, *Identifiers* is a set of *thread identifiers*, $L_S : L_O \to \mathit{Values}$ is the *local state* function that assigns values to *local objects* L_O, and $\mathcal{P}(X)$ denotes the power set of X. Given an abstract state s, it can be inspected using the following projections:

- $\mathit{global} : S \to G_S$, a function such that $\mathit{global}((g, \{(id_1, l_1), \ldots, (id_n, l_n)\})) = g$
- $\mathit{local} : S \times \mathit{Identifiers} \to \{\bot\} \cup L_S$, a function such that:

$$\mathit{local}((g, \{(id_1, l_1), \ldots, (id_n, l_n)\}), id) = \begin{cases} l_i & \text{if } id = id_i \\ \bot & \text{otherwise} \end{cases}$$

Definition 3.2.2. An *abstract program* is a tuple $\mathbb{P} = (S, \Delta, \Phi)$, where S is a set of abstract states, $\Delta \subseteq S \times S$ is a *transition* relation, with elements referred to as *transitions*, and $\Phi : \Delta \to \mathcal{P}(G_O) \times \mathcal{P}(G_O) \times \mathcal{P}(\mathit{Identifiers} \times \mathcal{P}(L_O))$ is an *abstract footprint* function. Further, given an abstract program $\mathbb{P}$, it can be inspected using the following projections:

- $\mathit{threads} : \Delta \to \mathcal{P}(\mathit{Identifiers})$, a function such that $\mathit{threads}((s, s')) = \{id \mid$ there exist $g_r, g_w \in \mathcal{P}(G_O)$, $id_1, \ldots, id_n \in \mathit{Identifiers}$, $l_1, \ldots, l_n \in \mathcal{P}(L_O)$, and $i \in \{1, \ldots, n\}$ such that $\Phi((s, s')) = (g_r, g_w, \{(id_1, l_1), \ldots, (id_n, l_n)\})$ and $id = id_i\}$
- $\mathit{readset} : \Delta \to \mathcal{P}(G_O)$, a function such that $\mathit{readset}((s, s')) = g_r$ such that $\Phi((s, s')) = (g_r, g_w, \{(id_1, l_1), \ldots, (id_n, l_n)\})$ for some $g_w \in \mathcal{P}(G_O)$, $id_1, \ldots, id_n \in \mathit{Identifiers}$, and $l_1, \ldots, l_n \in \mathcal{P}(L_O)$
- $\mathit{writeset} : \Delta \to \mathcal{P}(G_O)$, a function such that $\mathit{writeset}((s, s')) = g_w$ such that $\Phi((s, s')) = (g_r, g_w, \{(id_1, l_1), \ldots, (id_n, l_n)\})$ for some $g_r \in \mathcal{P}(G_O)$, $id_1, \ldots, id_n \in \mathit{Identifiers}$, and $l_1, \ldots, l_n \in \mathcal{P}(L_O)$

Note that the above definitions intentionally leave several terms undefined. The set G_O is assumed to be an abstract representation of the objects shared among the threads, the L_O set is assumed to be an abstract representation of the local objects of each thread, the *Values* set is assumed to be a set of values the global and local objects can take on, and the *Identifiers* set is assumed to be a set of unique thread identifiers. The transition relation Δ abstracts actual program transitions and the abstract footprint function Φ abstracts how global and local objects are accessed.

The motivation behind the above definitions is to formalize a mechanism that makes it possible to model concurrent programs at different levels of abstraction captured by

```c
#include <pthread.h>
#include <stdio.h>

int x = 0;

void *foo(void *args) {
  x++;
  return NULL;
}

int main(int argc, char **argv) {
  pthread_t tid;
  pthread_create(&tid, NULL, foo, NULL);
  x++;
  pthread_join(tid, NULL);
  assert(x == 2);
  return 0;
}
```

Figure 3.1: Concurrent Memory Access Example - Source Code

the choice of global and local objects. The following paragraphs illustrate the above abstract definitions on concrete examples.

Example 1. Consider the C program depicted in Figure 3.1. This program spawns a thread and then concurrently increments a global variable x from both threads. To abstractly model the state space of possible program states, one can set G_S to consist of assignments to the variable x, L_S to consist of control flow locations identified by source code line numbers, and *Identifiers* to consist of identifiers $\{parent, child\}$. The state space generated by this abstraction is depicted as a graph in Figure 3.2. The nodes represent abstract program states and the edges represent abstract program transitions, labeled by values of the *threads*, *writeset*, and *readset* functions respectively. This example illustrates 1) how to abstractly model program states of a concrete program, 2) how to model scheduling nondeterminism using a transition relation, and 3) how to model the effects of program transitions using the abstract footprint function. Note that choosing line numbers, as opposed to instruction register values, as the abstraction for control flow locations produces a model that does not contain a program state that witnesses the data race feasible when executing the program on hardware that does not increment x atomically.

Example 2. Consider the MPI [103] program depicted in Figure 3.3. When MPI creates two instances of this program, the instances exchange a message. To abstractly model the state space of possible program states, one can set G_S to consist of possible contents of the buffer MPI uses to transfer a message, L_S to consist of MPI function invocations identified by source code line numbers, and *Identifiers* to consist of identifiers $\{p0, p1\}$. The state space generated by this abstraction is depicted as a graph in Figure 3.4. The

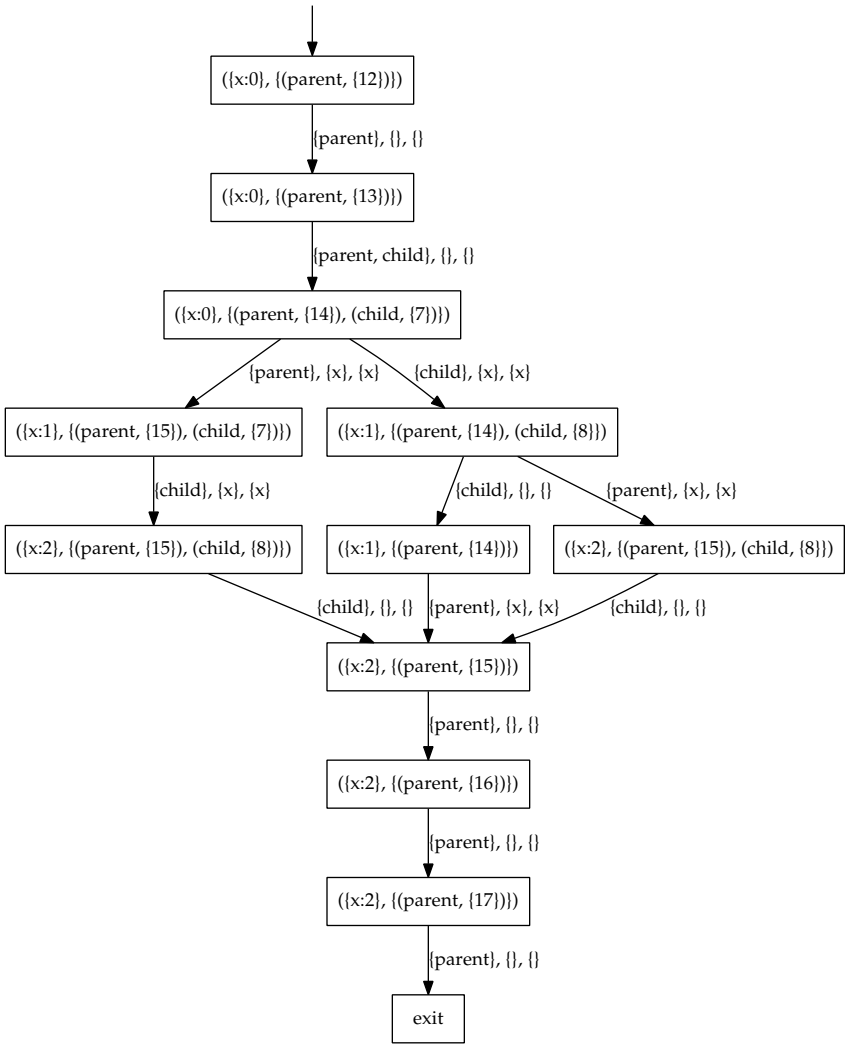


Figure 3.2: Concurrent Memory Access Example - State Space

nodes represent abstract program states and the edges represent abstract transitions, labeled by values of the *threads*, *writeset*, and *readset* functions respectively. This example illustrates how to model nondeterminism in the MPI specification – messages can be exchanged synchronously or asynchronously – using a transition relation.

Examples 1 and 2 together illustrate how the formalism introduced by Definitions 3.2.1 and 3.2.2 enables modeling concurrent programs at different level of abstraction. Note that the level of abstraction affects the complexity and precision of program analysis. The more details are abstracted away, the smaller the number of program states to analyze but the lower the precision with which the program is analyzed. The key to effective program analysis is to strike a balance between these two opposing goals: avoiding omission of important program behaviors or creation of overly detailed models that are too large to be analyzed [24]. dBug approaches this problem by offering different levels of abstraction at which it can model concurrent programs (cf. Chapter 7).

```c
#include <mpi.h>
#include <stdio.h>
#include <string.h>

#define SIZE 32

int main(int argc, char **argv) {
  char message[SIZE];
  int myrank;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  if (myrank == 0) {
    snprintf(message, SIZE, "Hello!");
    MPI_Send(message, strlen(message) + 1, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
  }
  if (myrank == 1) {
    MPI_Status status;
    MPI_Recv(message, SIZE, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &status);
    printf("%s\n", message);
  }
  MPI_Finalize();
  return 0;
}
```

Figure 3.3: MPI Communication Example - Source Code

Given a level of abstraction, dBug uses sequences of abstract states to model concrete executions of a concurrent program and state space exploration based on the DPOR algorithm to explore different scenarios allowed by scheduling nondeterminism. Similar to the presentation of actor programs, definitions of the happens-before and dependence relations are necessary to adapt the DPOR algorithm for abstract programs.

Definition 3.2.3. Let $\mathbb{P} = (S, \Delta, \Phi)$ be an abstract program and $s \in S$ an abstract state. An *execution* of $\mathbb{P}$ from s is defined as a finite sequence $\alpha = (s_0, \ldots, s_k)$ of abstract states of S such that $s = s_0$ and for all $i \in \{0, \ldots, k-1\} : (s_i, s_{i+1}) \in \Delta$. Δ_α is used to denote the set $\{(s_i, s_{i+1}) \mid i \in \{0, \ldots, k-1\}\}$.

To compute the happens-before relation for executions of abstract programs, dBug uses a *clock* function that for each thread and global object maintains a vector of the most recently observed values of *logical* time of each thread. In particular, when a thread or a global object is involved in an abstract program transition, the *clock* function updates the vector of the logical time values maintained by this thread or global object as follows. First, the most recently observed values of all threads and global objects involved in the same abstract program transition are joined and then each thread involved in an abstract program transitions advances its own logical time by one.

Definition 3.2.4. Let $\mathbb{P} = (S, \Delta, \Phi)$ be an abstract program, G_O its set of global objects, and $\alpha = (s_0, \ldots, s_k)$ its execution. The *clock* function $clock : S \times (Identifiers \cup G_O) \times Identifiers \to \mathbb{N}$ is a function that meets the following conditions:

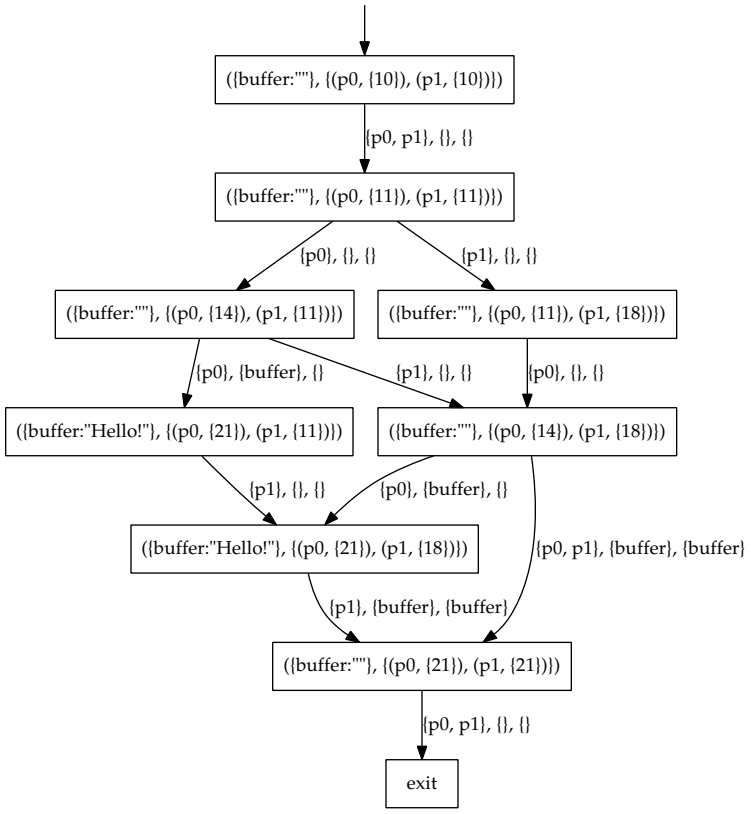


Figure 3.4: MPI Communication Example - State Space

1. for all $g \in G_O$ and $id, id' \in \mathit{Identifiers}$: $\mathit{clock}(s_0, id, id') = \mathit{clock}(s_0, g, id') = 0$
2. for all $id_1, id_2 \in \mathit{Identifiers}$ and $i \in \{1, \ldots, k\}$:

$$\mathit{clock}(s_i, id_1, id_2) = \begin{cases} \mathit{join}(id_2, s_{i-1}, s_i) + 1 & \text{if } id_1 \in \mathit{threads}(s_{i-1}, s_i) \text{ and } id_1 = id_2 \\ \mathit{join}(id_2, s_{i-1}, s_i) & \text{if } id_1 \in \mathit{threads}(s_{i-1}, s_i) \text{ and } id_1 \neq id_2 \\ \mathit{clock}(s_{i-1}, id_1, id_2) & \text{otherwise} \end{cases}$$

3. for all $g \in G_O$, $id \in \mathit{Identifiers}$, and $i \in \{1, \ldots, k\}$:

$$\mathit{clock}(s_i, g, id) = \begin{cases} \mathit{join}(id, s_{i-1}, s_i) & g \in \mathit{writeset}(s_{i-1}, s_i) \\ \mathit{clock}(s_{i-1}, g, id) & \text{otherwise} \end{cases}$$

where $\mathit{join}(id, s, s') = max\{\{\mathit{clock}(s, id', id) \mid id' \in \mathit{threads}(s, s')\} \cup \{\mathit{clock}(s, g, id) \mid g \in (\mathit{readset}(s, s') \cup \mathit{writeset}(s, s'))\}\}$.

Definition 3.2.5. Let $\mathbb{P} = (S, \Delta, \Phi)$ be an abstract program and $\alpha = (s_0, \ldots, s_k)$ its execution. The *happens-before* relation $\rightsquigarrow_\alpha \subseteq \Delta_\alpha \times \Delta_\alpha$ is a binary relation that meets the following condition: for all $(s_i, s_i'), (s_j, s_j') \in \Delta_\alpha$: $(s_i, s_i') \rightsquigarrow_\alpha (s_j, s_j')$ if and only if for all $id_i \in \mathit{threads}(s_i, s_i')$ and $id_j \in \mathit{threads}(s_j, s_j')$:

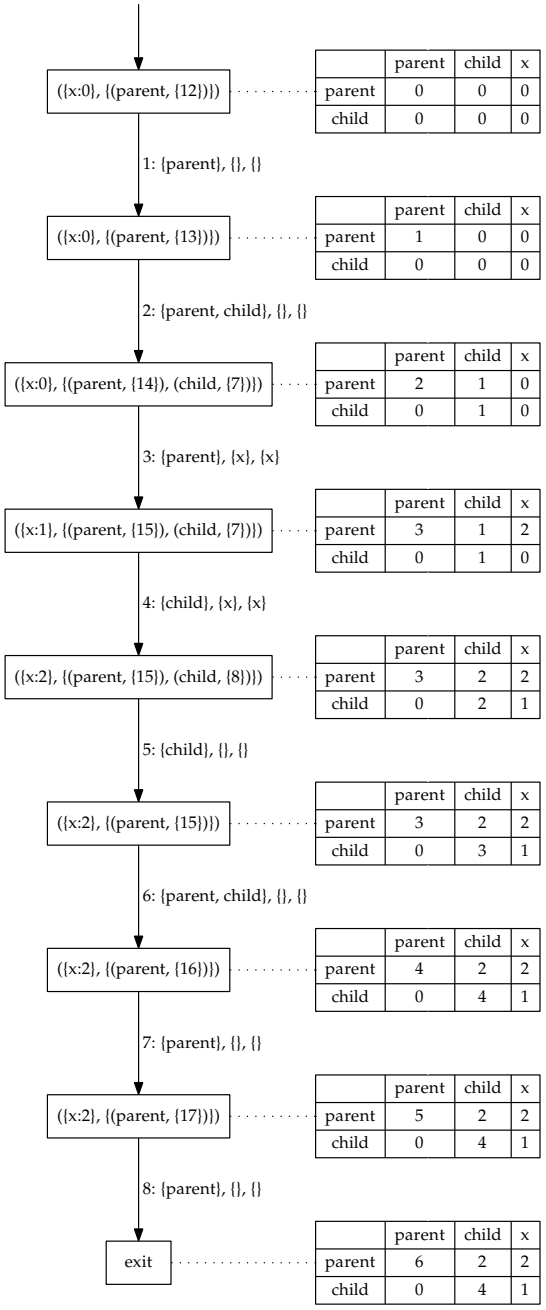


Figure 3.5: Concurrent Memory Address Example - Execution

1. for all $id \in \mathit{Identifiers} : \mathit{clock}(s_i, id_i, id) \leq \mathit{clock}(s_j, id_j, id)$ and
2. there exists $id \in \mathit{Identifiers}$ such that $\mathit{clock}(s_i, id_i, id) < \mathit{clock}(s_j, id_j, id)$.

Further, when a pair of transitions is not related through the happens-before relation, the transitions are called *concurrent*.

Example 3. Consider again the C program depicted in Figure 3.1 and its execution depicted in Figure 3.5, which annotates each abstract state with appropriate values of the *clock* function. The first argument of the *clock* function is identified by the abstract state each annotation table is attached to and the second and the third argument are identified by the column and the row of each annotation table respectively. Notably, this example illustrates how logical time is propagated through transitions that modify local state of multiple threads or the global state. The transitions of the depicted execution α are numbered and the values of the *clock* function can be used to compute the happens-before relation $\rightsquigarrow_\alpha = \{(1,2),(2,3),(3,5),(3,6),(4,5),(6,7),(7,8)\}^+$, where X^+ denotes the transitive closure of X. Note that the transitions 3 and 4 both increment the global variable x and are not causally ordered by $\rightsquigarrow_\alpha$. Consequently, although the abstract state space does not contain an abstract state that witnesses the data race feasible in practice, the happens-before relation $\rightsquigarrow_\alpha$ along with the abstract footprint function Φ can be used to detect the data race. Further, note that since the abstract program does not keep track of the status of the child thread as part of the global state, the happens-before relation $\rightsquigarrow_\alpha$ fails to capture the causality implied by joining the child thread from the parent thread. In other words, the happens-before relation of the abstract program is in general an under-approximation of the true causality between program transitions.

Next, a dependence relation [60] between transitions of an abstract program execution is defined to track transitions with non-commutative effects.

Definition 3.2.6. Let $\mathbb{P} = (S, \Delta, \Phi)$ be an abstract program and $\alpha = (s_0, \ldots, s_k)$ its execution. The *dependence* relation $D_\alpha \subseteq \Delta_\alpha \times \Delta_\alpha$ is a binary relation such that for all $(s_i, s_i'), (s_j, s_j') \in \Delta_\alpha : ((s_i, s_i'), (s_j, s_j')) \in D_\alpha$ if any of the following conditions is true:

1. $\mathit{writeset}(s_i, s_i')$ and $\mathit{writeset}(s_j, s_j')$ are not disjoint
2. $\mathit{readset}(s_i, s_i')$ and $\mathit{writeset}(s_j, s_j')$ are not disjoint
3. $\mathit{writeset}(s_i, s_i')$ and $\mathit{readset}(s_j, s_j')$ are not disjoint
4. $\mathit{threads}(s_i, s_i')$ and $\mathit{threads}(s_j, s_j')$ are not disjoint

Further, when a pair of transitions is related through the dependence relation, the transitions are called *dependent*.

Unlike the dependence relation for actor programs (Definition 3.1.6), the dependence relation for abstract programs (Definition 3.2.6) can fail to relate transitions with non-commutative effects. This can happen when the state modified by both transitions has been abstracted away. Note that this is not a shortcoming of the above definition. If the abstract program modeled every bit of the concrete program state, the dependence relation would likely relate every pair of program transitions. After all, every program transition modifies the contents of the instruction register. Consequently, the DPOR algorithm would fail to achieve any reduction, rendering the systematic testing approach impractical for any real-world program. Instead, the abstraction of the concrete program state identifies state pertinent to the inspection of the desired program behavior. The dependence relation computed based on Definition 3.2.6 simply represents a mechanism that passes this information onto the DPOR algorithm.

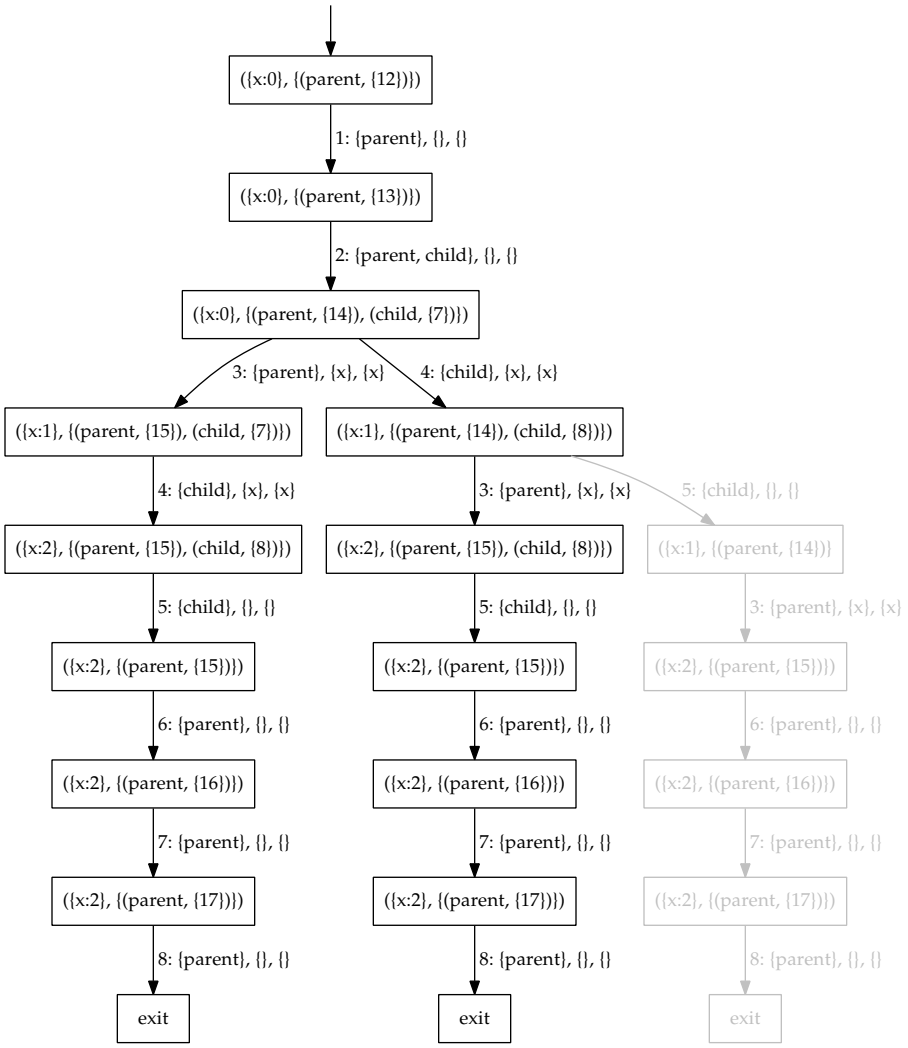


Figure 3.6: Concurrent Memory Address Example - Execution Tree

In practice, dBug executes the concrete program and the DPOR algorithm implemented in dBug inspects an abstraction of this execution, searching for pairs of transitions that are both concurrent and dependent. A pair of such transitions suggests that the transitions could execute in either order and the effects of the transitions may not be commutative, possibly affecting the outcome of the test. When the DPOR algorithm encounters a pair of such transitions, the exploration frontier of the DPOR algorithm is updated to make sure that all permutations of concurrent and dependent transitions are eventually explored.

Example 4. Consider again the C program depicted in Figure 3.1. The execution tree depicted in Figure 3.6 illustrates the use of the DPOR algorithm to explore the abstract state space depicted in Figure 3.2. Let us assume that the DPOR algorithm first

explores the execution corresponding to the leftmost branch of the execution tree. This happens to be the same execution as depicted in Figure 3.5. In Example 3, we found out that the transitions 3 and 4 are concurrent. Comparing their abstract footprints reveals that they are also dependent. Consequently, the DPOR algorithm infers that it is necessary to explore an execution that permutates these two transitions. Let us assume that to that end, the DPOR algorithm explores the execution corresponding to the middle branch of the execution tree depicted in Figure 3.6. The DPOR algorithm then examines this execution and finds out that transitions 4 and 3 are concurrent and dependent. Consequently, the DPOR algorithm infers that it is necessary to explore an execution that permutates these two transitions. As this has already happened, the DPOR algorithm finishes the exploration. Notably, the DPOR algorithm avoids exploration of the rightmost branch of the execution tree depicted in Figure 3.6.

3.2.2 Design

The goal of dBug is to enable systematic testing of multithreaded and distributed programs. To this end, dBug needs to be able to monitor the program state, to control the order of concurrent program transitions, and to deterministically replay parts of program executions. These three requirements map to the design of the following dBug components. The *interposition layer* is responsible for monitoring the program state and controlling the execution of program transitions. The *arbiter* is responsible for maintaining the abstract program state, identifying what transitions can be executed next, and scheduling these transitions. The *explorer* is responsible for exploring the state space of abstract program states.

Interposition Layer

The interposition layer monitors a concrete program execution for events that affect the abstract state maintained by the arbiter. When a thread is about to execute such an event, the interposition layer intercepts this event, suspends the execution of the thread, and informs the arbiter about this event. At some later point, the arbiter instructs the interposition layer to resume execution of the suspended thread.

The interposition layer receives its name from the mechanism intended for its deployment. As Figure 3.7 suggests, an interposition layer is assumed to exist between the program and the environment in which the program runs, providing a mechanism to monitor and control the interactions of the program and the environment.

Arbiter

The arbiter uses a client-server architecture (Figure 3.8) to collect information about the abstract state of the program through the events intercepted by the interposition layer. To control scheduling nondeterminism, the arbiter waits until all threads cease execution either by trapping into the interposition layer or simply terminating. The

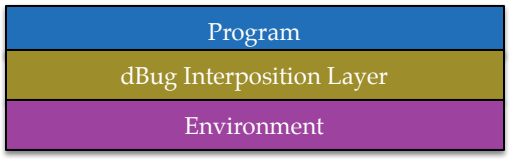


Figure 3.7: Monitoring Program Behavior with dBug Interposition Layer

arbiter then enumerates all transitions that are possible from the current abstract state and selects one of them. The selected transitions identifies one or more threads whose execution is then resumed. This process is repeated until all threads terminate or a stopping condition, such as reaching a deadlock, is encountered.

Explorer

To explore the space of possible abstract states of a program test, the explorer repeatedly starts an instance of the arbiter and the program test with the interposition layer in place and waits for the program test to finish. Once the program test finishes, the explorer collects information about the sequence of abstract states explored by the arbiter. This information is used to represent the state space as an execution tree and to compute the happens-before relation and the dependence relation in order to explore the execution tree using the DPOR algorithm. Once an execution is processed by the explorer, an unexplored node on the exploration frontier is identified and used to create a schedule for the arbiter which will steer the next execution towards unexplored parts of the state space. Figure 3.9 illustrates how, over time, the explorer communicates with different instances of the arbiter, while exploring the execution tree.

3.2.3 Implementation

Interposition Mechanism

In the course of dBug's lifetime, several mechanisms for the implementation of the interposition layer have been experimented with. For example, early dBug prototypes [136] relied on manual source code annotations. This process was time consuming, error-prone, and required access to and understanding of the program source code.

To avoid these problems, the mechanism has since evolved to its current form [135, 137] that relies on runtime interposition on dynamically linked symbols. This mechanism is well aligned with dBug's goal to offer systematic testing at different levels of abstraction, which are provided by the function prototypes of the interposed functions. Further, runtime interposition avoids the need for having access to the program source code and works irrespective of the choice of programming language.

The main drawback of using runtime interposition is its inability to track memory accesses, limiting the precision of program abstraction and analysis. Although tracking

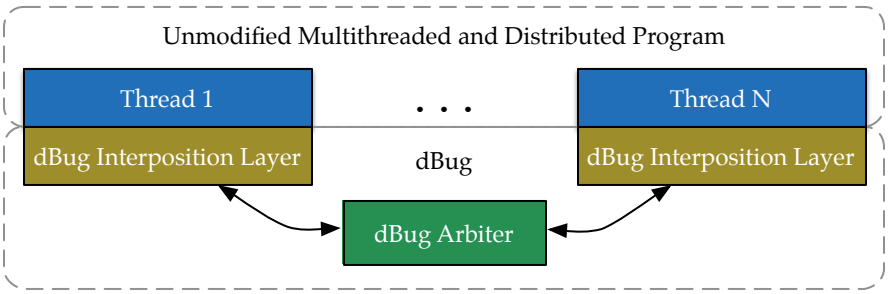


Figure 3.8: Controlling Scheduling Nondeterminism with dBug Arbiter

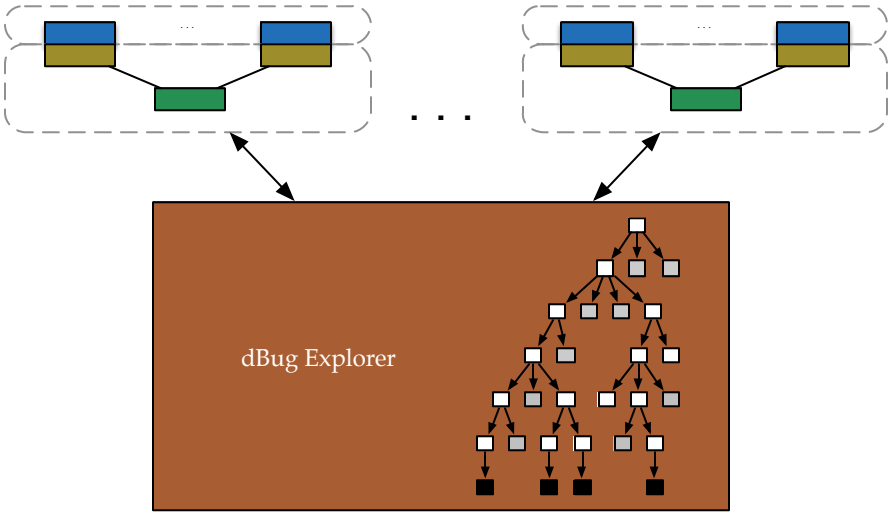


Figure 3.9: Exploring Execution Tree with dBug Explorer

of memory accesses is possible with alternative approaches to runtime interposition such as dynamic instrumentation [98] and virtualization [11, 28], tracking memory accesses has its own drawbacks. Namely, modeling concurrent program transitions at the granularity of (shared) memory accesses exacerbates the combinatorial explosion of the number of scenarios to explore. In addition to that, using technology for tracking memory accesses imposes a runtime overhead on program execution and this overhead reduces the rate at which systematic testing explores the state space. Given the extent of the combinatorial explosion without considering memory accesses (see Chapters 6 and 7), dBug chooses to avoid tracking of memory accesses, leaving the job of searching for low-level data races to tools such as Eraser [128], RaceTrack [169], and PACER [25]. The analysis carried out by these tools is orthogonal to systematic testing and thus can be used in combination with systematic testing to increase its precision.

Default Abstraction

The default abstraction dBug uses to model program execution is the POSIX interface [119]. Using this abstraction has several advantages. It is shared by all programs of POSIX-compliant operating systems and the libraries that implement the POSIX interface typically use dynamic linking, enabling runtime interposition. Consequently, using the POSIX interface as the default abstraction allows dBug to target a wide range of programs without the need to access or modify their source code. In practice, applications of dBug reported in this thesis are done in the context of the operating system Linux.

In the POSIX abstraction, the global state keeps track of the following objects: threads, processes, barriers, condition variables, mutexes, read-write locks, semaphores, spinlocks, epoll descriptors, pipe descriptors, file descriptors, and socket descriptors. The local state of each thread consists of its call stack and dBug keeps track of the values of the *clock* function using a vector clock [123]. A detailed list of POSIX interface functions that dBug interposes on can be found in Appendix A.

Exploration

To explore the space of possible abstract program states, dBug controls the order in which program threads execute concurrent program transitions. To this end, dBug keeps track of all program threads, maintaining this information as part of the global state and updating the global state when relevant POSIX interface invocations, such as `pthread_create()` or `fork()`, are encountered. Being knowledgeable of the set of all program threads allows dBug to recognize when the program reaches a *quiescent* state when all program threads have either terminated or trapped into the dBug interposition layer. When a quiescent state is reached, the program waits for the dBug arbiter to make a scheduling decision.

The dBug arbiter repeatedly waits for the program to reach a quiescent state, enumerates all program transitions possible from that state, selects one of the program transitions, and resumes execution of the threads that participate in that transition. This process is repeated until the program terminates or a stopping condition, such as a deadlock, is encountered. Note that the algorithm that the dBug arbiter uses to schedule program transitions relies on the absence of busy waiting. If a thread can run indefinitely without trapping into the interposition layer, the program might never reach a quiescent state. To identify busy waiting, dBug uses a timeout and reports the backtrace of suspect threads, effectively enforcing a programming discipline where threads yield the CPU through functions such as `sleep()`, `sched_yield()`, or `poll()`.

Once the dBug arbiter reaches a terminal state, it sends a description of the sequence of program transitions it explored and all the alternative scheduling choices it has encountered to the dBug explorer. The dBug explorer then maps the explored sequence of program transitions to a branch of the execution tree and marks it as explored. The description of program transitions contains logical clock timestamps and abstract

footprints that the dBug explorer uses to compute the happens-before and dependence relations over the set of program transitions. This information is in turn used by dBug's implementation of the DPOR algorithm to update the exploration frontier of the execution tree, identifying nodes of the exploration frontier that need to be explored to guarantee that all permutations of concurrent and dependent program state transitions are explored.

Next, the dBug explorer identifies a node of the exploration frontier to explore next. This node represents a sequence of scheduling choices that has not been previously explored. The initial program state is then recreated, either by simply starting the program anew or through a user-provided initialization function, and the dBug arbiter is told what scheduling choices to make to steer the execution towards the previously identified node of the exploration frontier.

An important prerequisite of systematic testing is that the identification of scheduling choices needs to be consistent across different program executions. To this end, dBug uses deterministic numbering of threads, which is possible because dBug controls the order in which threads are created. Further, dBug's implementation follows a strict programming discipline to make sure that identical scheduling choices result in identical abstract program states across different program executions.

The programming discipline dictates that the dBug arbiter models each event it interposes on using three functions – *preschedule*, *test*, and *execute* – that cannot block and execute atomically with respect to one another. The following paragraphs detail these functions and Figure 3.10 illustrates the typical operation of the dBug arbiter.

The *preschedule* function is used to update the global state when the corresponding event is originally intercepted. For example, the *preschedule* function of the `pthread_barrier_wait(b)` event updates the number of waiters of the barrier *b*, while the *preschedule* function of the `pthread_cond_wait(c,m)` event updates the state of the mutex *m*. The programming discipline requires that the effects different *preschedule* functions have on the global state are commutative.

The *test* function is used to inspect the global state and to determine what program transitions containing the corresponding event are possible. The *test* function computes a set of program transitions, each containing its abstract footprint and a *seed* that can be used to reproduce the nondeterministic choices made in the course of creating this program transition. For example, the *test* function of the `pthread_barrier_wait(b)` event checks if the actual number and target number of waiters of the barrier *b* match and if so, generates *n* program transitions, one per waiter, returning `PTHREAD_BARRIER_SERIAL_THREAD` from different waiters (see `pthread_barrier_wait()` man pages for details). The programming discipline requires that no *test* function modifies the global state.

The *execute* function is used to model the effect that executing the corresponding event has on the global state. It inputs a program transition seed and uses it to reproduce any nondeterministic choices in accordance with the choices made by the *test* function that generated this program transition. When a program transition involves multiple

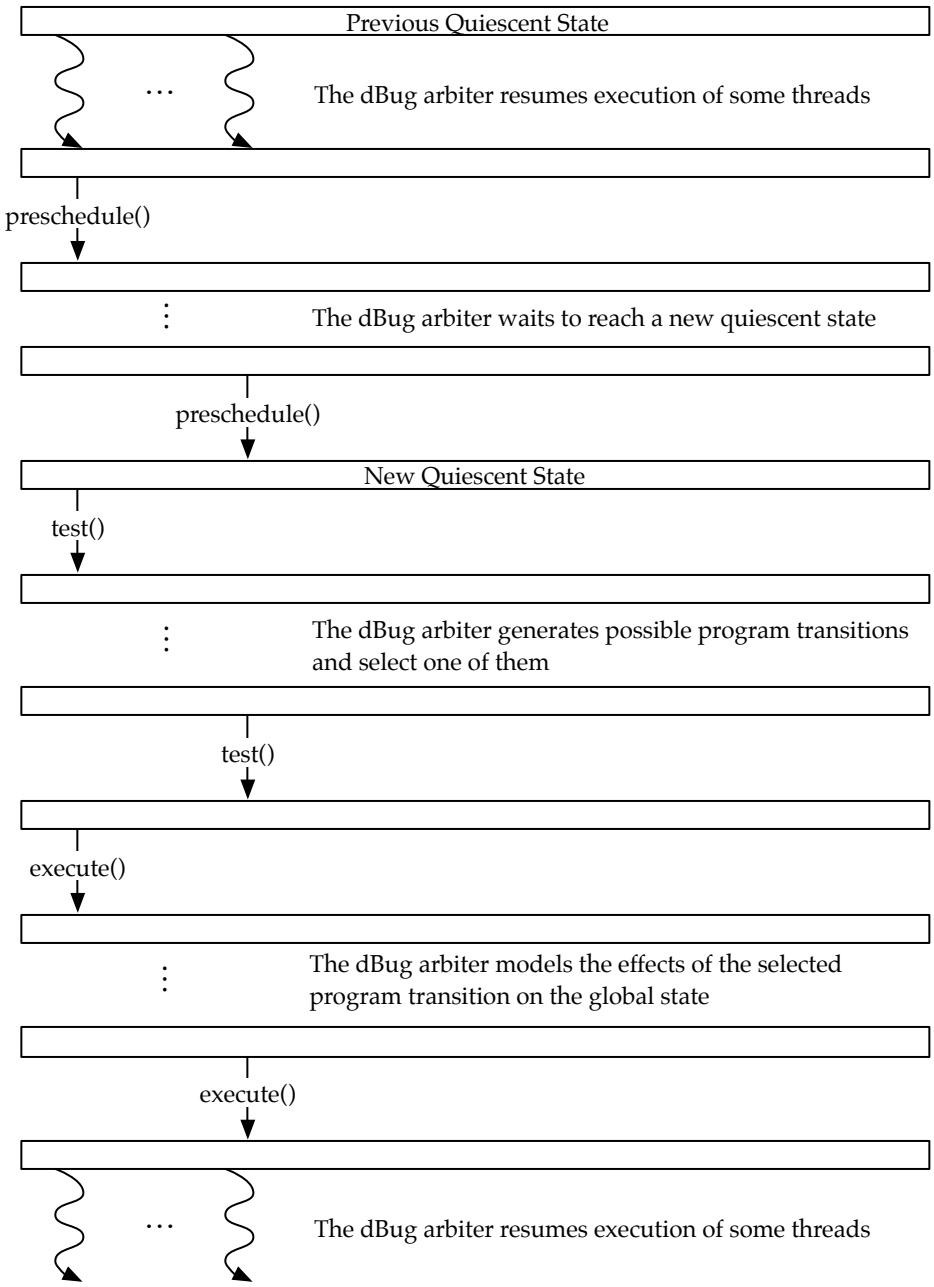


Figure 3.10: Arbiter Execution Example

events, the programming discipline requires that their respective *execute* functions are executed in a deterministic order.

In summary, the programming discipline of dBug arbiter makes sure that dBug does not introduce any nondeterminism into the process of enumerating and executing program transitions. Its key properties are that 1) different *preschedule* functions are commutative, 2) no *test* function is allowed to modify the global state, and 3) *execute* functions are executed in a deterministic order.

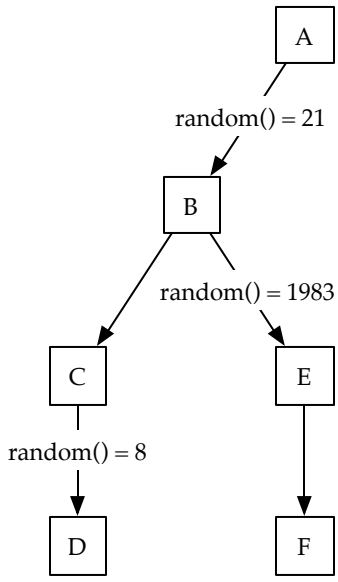


Figure 3.11: Direct Record-Replay Mechanism Example

Nondeterminism

A separate concern that dBug needs to address in order to enable deterministic replay is the input nondeterminism coming from the operating system. In general, a program can input values through the functions it interposes on and these values can depend on factors external to the program. For example, the value of the `getpid()` function, which returns an integer identifier of a process, depends on the context in which the program runs, while the value of the `time()` function, which returns the number of seconds since January 1st 1970, depends on the time of the function invocation. If the program logic depends on these values, then different program executions that use the same scheduling choices could eventually diverge. To address this problem, dBug uses three different mechanisms: direct record-replay, indirect record-replay, and time travel.

The *direct record-replay* mechanism builds on previous work [55, 127] that records outcomes of system calls to enable deterministic replay of program execution. The difference in the context of dBug is that a program transitions from a replay phase to a record phase in the course of its execution. This transition occurs when the program execution finishes replaying scheduling choices identified by the dBug explorer, reaching the execution tree exploration frontier. This mechanism is used for functions, such as the `random()` function or the `read()` function when reading from `/dev/random`, that generate transient variable values.

Example 5. Figure 3.11 depicts an example of direct record-replay execution. Initially, the execution tree exploration frontier maintained by dBug consists of the initial program state represented by the node A. The underlying program is then executed and the explored sequence of scheduling choices is modeled as the branch A-B-C-D. In the

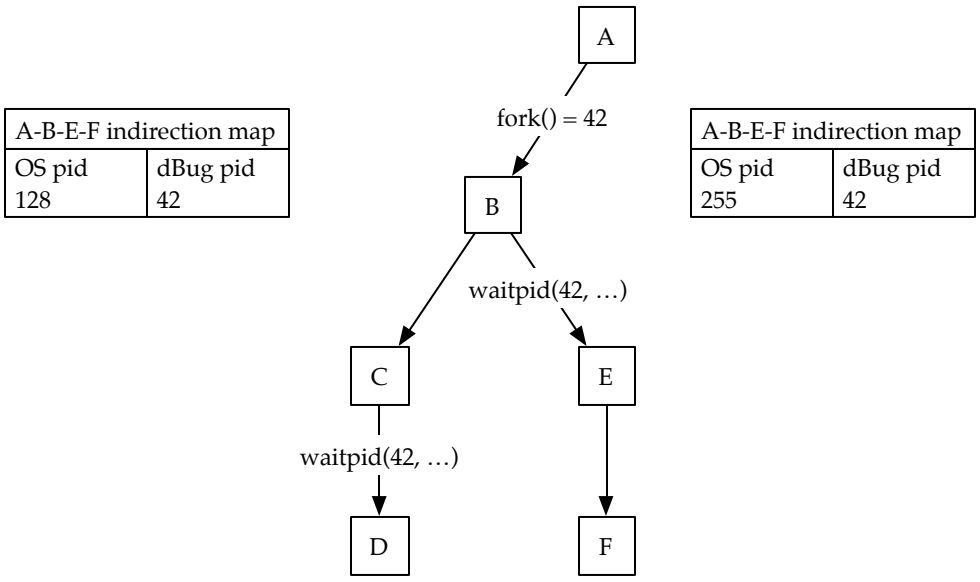


Figure 3.12: Indirect Record-Replay Mechanism Example

course of the execution, two calls to the `random()` function are encountered and their outcome is recorded in the execution tree. The DPOR algorithm is then used to add important alternative scheduling choices, such as the node E, to the execution tree exploration frontier. A new program execution is then started, steering the execution towards the node E and the explored sequence of scheduling choices is modeled as the branch A-B-E-F. In the course of this execution, two calls to the `random()` function are encountered again. In contrast to the first execution, the outcome of the first call is replayed using the outcome recorded in the first execution. The execution then continues replaying scheduling choices identified by the dBug explorer until it reaches the program state represented by the node B and then transitions from the replay phase to the record phase. From that point the execution continues to explore an arbitrary sequence of scheduling choices. When the second call to the `random()` function is encountered, its outcome is recorded.

The direct record-replay mechanism may fail when the outcome of one interposed function persists in the program environment and can be later used as an input of another interposed function. The problematic scenario occurs when the outcome of the first function call is replayed and then the second function call uses this outcome to look up information in the environment. Since dBug does not control the state of the environment, the look up may fail. To address this problem, dBug uses the *indirect record-replay* mechanism. This mechanism maintains an indirection map that translates environment values to deterministically generated unique values, which are used for the purpose of record-replay. When these deterministic values are used as an input to an interposed function, dBug uses the indirection map to translate these values back to their environmental counterpart. This mechanism is used for values that serve as

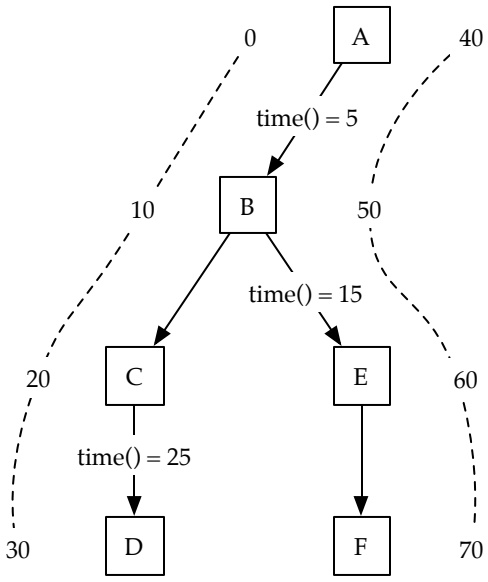


Figure 3.13: Time Travel Mechanism Example

environmental handles such as process identifiers of functions `fork()` and `waitpid()` or socket descriptors of functions `socket()` and `send()`.

Example 6. Figure 3.12 depicts an example of indirect record-replay. Similar to Example 5, this example explores two executions, replaying some scheduling choices in the course of the second execution. The first execution follows a call to the `fork()` function with a call to the `waitpid()` function. The true outcome of the `fork()` function, process identifier 128, is assigned the deterministic value of 42, which is recorded and returned to the child process. When the `waitpid()` function later uses 42 as an input argument, the dBug interposition layer replaces this value with its environmental counterpart 128. The second execution also follows a call to the `fork()` function with a call to the `waitpid()` function. In contrast to the first execution, when the second execution encounters the `fork()` function call, its true outcome, process identifier 255, is assigned the deterministic value of 42, replayed from the first execution. When the `waitpid()` function uses 42 as an input argument, the dBug interposition layers replaces this value with its environmental counterpart 255.

Although input nondeterminism stemming from functions that return absolute time values can be handled by the direct record-replay mechanism, the use of the mechanism can result in unlikely timing scenarios. For example, consider a program that uses a pair of calls to the `gettimeofday()` function, one at the beginning of its execution and one at the end of its execution, to measure its runtime. The use of direct record-replay will result in the outcome of the first call in the first execution being replayed in all of the subsequent executions, while the outcome of the second call will be generated anew in every execution. Consequently, the runtime measurements will grow more and

more skewed with each explored execution. To address this problem, dBug uses the *time travel* mechanism, which for each node of the execution tree keeps track of the time the node was first encountered, referred to as the *creation* time. Similar to the previous mechanisms, the time travel mechanism uses a replay phase and a record phase. The replay phase is identical to the replay phase of direct record-replay and the transition to the record phase occurs when the program execution finishes replaying scheduling choices identified by the dBug explorer. When the transitions happens, the creation time of the execution tree node corresponding to the current program state is compared to the current time and their difference is subtracted from outcomes of all subsequent calls to absolute time functions. This mechanism is used for values of functions that return absolute time values such the `gettimeofday()` function or the `time()` function.

Example 7. Figure 3.13 depicts an example of time travel. Similar to Examples 5 and 6, this example explores two executions, replaying some scheduling choices in the course of the second execution. In addition to depicting the execution tree, the figure also depicts two dashed lines that record the progression of absolute time in the course of the two executions. The first execution makes two calls to the `time()` function and their outcome is simply recorded and returned to the caller. The second execution also makes two calls to the `time()` function. In contrast to the first execution, the outcome of the first call is replayed using the recorded value, returning the value 5. When the node B is encountered, the time travel mechanism recognizes that it is about to cross the exploration frontier and computes the difference between 50, the current time, and 10, the first time the node B was encountered. This difference is then subtracted from the second call to the `time()` function, producing the value 15, instead of the value 55. By doing so, the time travel mechanism tricks every execution into thinking they started at the same time.

Interface Modeling

To extend dBug to support interposition of new events, for example when adding support for a new level of abstraction (cf. Chapter 7), several steps are necessary. First, one needs to implement wrappers that allow the dBug interposition layer to intercept these events when they are about to occur. Second, one needs to extend the RPC interface of the dBug arbiter with methods that control scheduling of the intercepted events. When an event is intercepted, the dBug interposition layer issues a blocking RPC to the dBug arbiter containing information about the intercepted event. The RPC returns when the dBug arbiter decides to schedule the intercepted event for execution. Third, one needs to extend the dBug arbiter with the *preschedule*, *test*, and *execute* functions that model the semantics of the intercepted events.

By and large, extending dBug with support for one event requires over one hundred lines of new code spread across different source files. Unsurprisingly, this process is time consuming, limiting the rate at which the support for new events and existing events can be added and modified respectively. To address this problem, dBug uses a custom code generator to generate the dBug interposition layer wrappers source code

```
<type>      ::= list<type> | <primitive>
<primitive> ::= boolean | integer | string | event
```

Figure 3.14: Type Grammar of dBug Object Models

and the dBug arbiter RPC interface automatically from the prototypes of the interposed events. The RPC interface implementation is in turn generated from the dBug arbiter RPC interface using Apache Thrift [5].

In addition, dBug offers a modeling language that can be used to describe the objects maintained as part of the global state and the effect interposed events have on the global state. This modeling language comes with a compiler that uses the object and event models written in this language to automatically generate parts of the dBug arbiter source code, including the *preschedule*, *test*, and *execute* functions.

The objects are modeled as records consisting of fields that have a name and a type. The types recognized by dBug are summarized by the grammar presented in Figure 3.14. The grammar defines four primitive types: `boolean`, `integer`, `string`, and `event`. The first three represent the standard data types, while the `event` type represents event identifiers. In addition to the primitive types, the grammar defines the `list<type>` type that represents lists of typed elements.

The events are modeled as state machines that input typed values and always execute two transitions on their way from their initial state to their final state. The first transition models the effect of the *preschedule* function, while the second transition models the effect of the *execute* function. Each transition is labeled by a *guard* that identifies the global states in which this transition is enabled and an *action* that describes the effect this transition has on the global state. Further, the second transition is also labeled with the return value to use for the event.

The guards and actions recognized by dBug are summarized by the grammars presented in Figures 3.15 and 3.16 respectively. The `object` terminal represents different types of objects defined by the object models. The `id` terminal represents identifiers of the state machine input values. The `field` terminal represents the different object fields defined by the object models. The `integer` terminal represents an integer. The `event` terminal represents an identifier of this event. The `tid` terminal represents an identifier of the thread that intercepted this event. The `exists(object(id))` guard checks if the object identified by the given object type and identifier exists. The `empty(object(id).field)` guard checks if the list identified by the given object type, identifier, and field is empty. The `contains(object(id).field, <value>)` guard checks if the list identified by the given object type, identifier, and field contains the given value. The `length(object(id).field)` value equals the length of the list identified by the given object type, identifier, and field. The `create(object(<ids>))` action creates a new object given the object type and a list of constructor arguments.

```
<guard>       ::= <guard> and <proposition> | <proposition>
<proposition> ::= not(<proposition>)
                | exists(object(id))
                | empty(object(id).field)
                | contains(object(id).field, <value>)
                | <expression> == <expression>
                | <expression> > <expression>
                | <expression> < <expression>
                | <expression>
<expression>  ::= <expression> + <value>
                | <expression> - <value>
                | <value>
<value>       ::= length(object(id).field)
                | object(id).field
                | id
                | integer
                | event
                | tid
```

Figure 3.15: Guard Grammar of dBug Event Models

The `delete(object(id))` action deletes an existing object identified by the given object type and identifier. The `enqueue(object(id).field, <value>)` action adds the given value to the front of the list identified by the given object type, identifier, and field. The `remove(object(id).field, <value>)` action removes all elements whose values matches the given value from the list identified by the given object type, identifier, and field. The `dequeue((object(id).field)` action removes the last element of the list identified by the given object type, identifier, and field. The `empty((object(id).field)` action removes all elements of the list identified by the given object type, identifier, and field. The `warning(string)` action generates a warning message using the given string.

To illustrate the modeling language in action, Figures 3.17, 3.18, 3.19, 3.20, and 3.21 depict the event models for the events of the `pthreads` spinlock interface. Note that these figures are generated automatically from the event models using our compiler.

In addition to the `pthreads` spinlock interface, the modeling language was used to model the `pthreads` barrier, condition variable, mutex, and read-write lock interfaces. The object and events models resulting from this effort along with the prototypes of the events dBug intercepts amount to 3,229 lines of data. This data is used to automatically generate 48,761 out of 73,359 lines (over 66%) of dBug's source code.

```
<action>     ::= <statements>
<statements> ::= <statements>; <statement> | <statement>
<statement>  ::= create(object(<ids>))
                 | delete(object(id))
                 | enqueue(object(id).field, <value>)
                 | remove(object(id).field, <value>)
                 | dequeue(object(id).field)
                 | empty(object(id).field)
                 | warning(string)
                 | object(id).field = <expression>
<expression> ::= <expression> + <value>
                 | <expression> - <value>
                 | <value>
<value>      ::= object(id).field
                 | id
                 | integer
                 | event
                 | tid
<ids>        ::= <ids>, id | id
```

Figure 3.16: Action Grammar of dBug Event Models

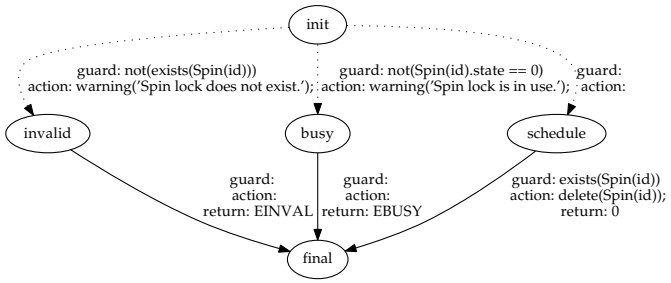


Figure 3.17: Event Model of `pthread_spin_destroy(id)`

State Space Estimation

Similar to ETA, dBug recognizes the importance of estimating test complexity and implements a technique that estimates the number thread interleavings that dBug examines in the course of a systematic test. In particular, to estimate the size of a partially explored execution tree, dBug uses a technique based on the *weighted backtrack estimator* [80] (cf. Chapter 4). The technique treats the set of explored branches as a sample of the entire execution tree assuming uniform distribution over edges and computes the estimate as the number of explored branches divided by the aggregated probability the branches are explored by a random exploration.

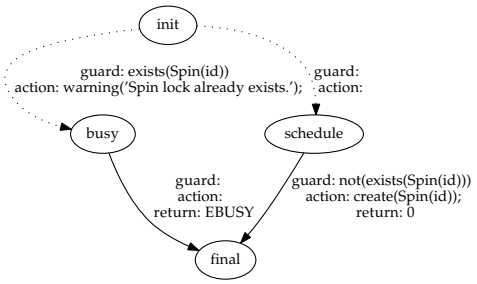


Figure 3.18: Event Model of `pthread_spin_init(id)`

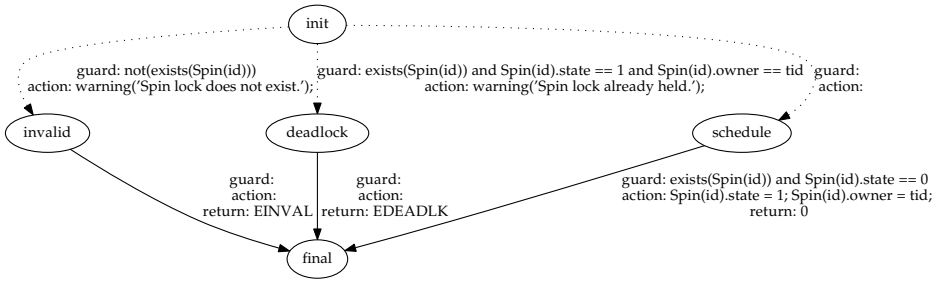


Figure 3.19: Event Model of `pthread_spin_lock(id)`

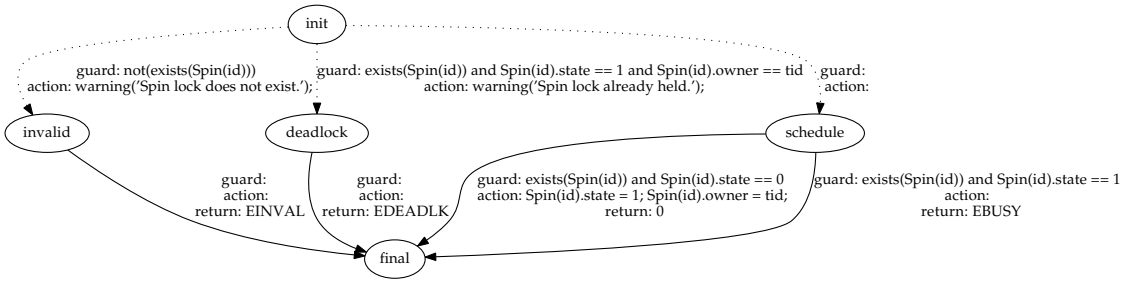


Figure 3.20: Event Model of `pthread_spin_trylock(id)`

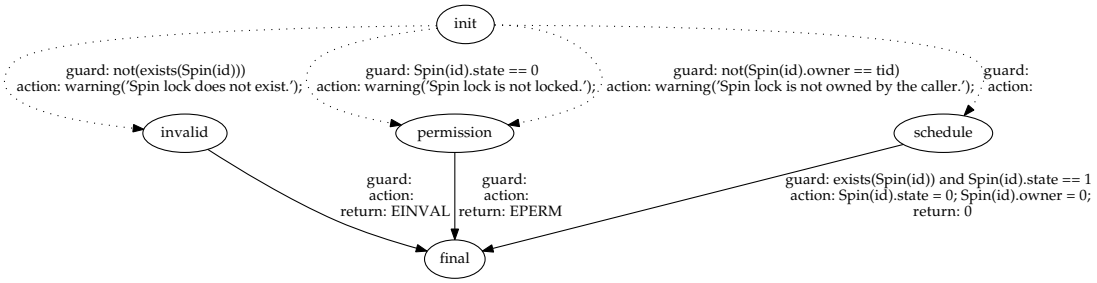


Figure 3.21: Event Model of `pthread_spin_unlock(id)`

Chapter 4

State Space Estimation

Over the course of the past 15 years, research advances in systematic testing of concurrent programs [61, 82, 107, 160, 163, 165] have made the approach practical and easy to adopt. As systematic testing moves into wider practice, becoming a part of testing infrastructures of large-scale system developers [107, 138], new practical challenges are emerging. For example, when resources available for systematic testing are scarce, one needs to solve the problem of efficient allocation of resources to a collection of tests.

Real-world test suites, such as the one targeted by ETA [138], consist of hundreds of tests of varied complexity. In the context of systematic testing of concurrent programs, it is reasonable to assume that the resources available for running these tests are not always sufficient to complete all tests in the time allotted for testing. In such cases, high-level testing objectives, such as *maximizing the number of completed tests* or *achieving even coverage across tests* can be used to drive allocation of testing resources. Mapping these high-level testing objectives into working allocation mechanisms is an important, practical, and yet unaddressed problem.

This chapter proposes a solution to this problem based on estimation of the length of the state space exploration carried out by systematic tests. As discussed in Chapter 2, most systematic testing implementations [61, 82, 137, 160] use *stateless* exploration, recording the different executions encountered during a test in an *execution tree*. Under this abstraction, the problem of test length estimation can be formulated as the problem of estimating the length of execution tree exploration. Besides offering a measure of test complexity, the estimates can be also used to implement resource allocation policies.

The estimation techniques presented in this chapter can be characterized as *online* – updating the estimate as the exploration makes progress through the state space – and *passive* – not mandating a particular order in which the exploration proceeds. The benefit of online estimation is that the estimate can be refined as new information about the state space is gathered, while the benefit of passive estimation is that it can be combined with any exploration strategy. Furthermore, passive estimation techniques can be evaluated using exploration traces. To enable verification of the results presented in this chapter, our evaluation uses a publicly available collection of exploration traces from a real-world deployment at Google [140].

This chapter makes the following contributions. First, building on research on search tree size estimation [80], this chapter presents techniques for estimating the length of a systematic test. Second, this chapter demonstrates the practicality of test length estimation by using it to implement several resource allocation policies. Third, this chapter uses a collection of exploration traces from a real-world deployment to evaluate 1) the accuracy of the presented estimation techniques and 2) the efficiency of the presented resource allocation policies.

The rest of this chapter is organized as follows. Section 4.1 describes the syntax and semantics of exploration traces. Section 4.2 presents 1) techniques for estimating the length of systematic tests and 2) policies for resource allocation based on test length estimates. Section 4.3 describes a collection of exploration traces [140] and uses these traces to evaluate the accuracy of the presented estimation techniques and the efficiency of the presented resource allocation policies. Section 4.4 discusses related work and Section 4.5 draws conclusions.

4.1 Background

The estimation techniques presented in this chapter are passive, not mandating a particular order in which the exploration proceeds. Consequently, the problem of estimation of the length of a systematic test can be described using the abstraction of an *exploration trace*, which identifies events pertinent to the estimation.

An exploration trace is a sequence of events, where an event is one of the following:

1. `AddNode x y` – A node `x` with parent `y` is added to the execution tree.
2. `Explore x` – Node `x` is scheduled for exploration.
3. `Transition x` – Exploration transitions to node `x`.
4. `Start` – Exploration of a new execution is started from the root node.
5. `End t` – Exploration of the current execution has finished after `t` time units.

Example 8. Figure 4.1 depicts an execution tree and Figure 4.2 depicts a trace of its exploration. Initially, the root node 0 is added and scheduled for exploration. Next, an execution is started from the root node. The children 1, 2, and 3 of the root node are added and the child 1 is scheduled for exploration. The execution then transitions to node 1. The children 4 and 5 of node 1 are added and the child 4 is scheduled for exploration. The execution transitions to node 4. The child 6 of node 4 is added and scheduled for exploration. The execution transitions to node 6. For the sake of this example, let us assume that the exploration then infers that node 3 needs to be explored and schedules it for exploration. This concludes exploration of the current execution, requiring a total of, for example, 0.42 time units. Next, a new execution is started from the root node. The second execution steers towards node 3 and explores a branch in its subtree, requiring a total of, for example, 0.29 time units, concluding the exploration. Note that nodes 2, 5, and 8 are never explored because the exploration inferred they will not produce new interleavings of concurrent and dependent events.

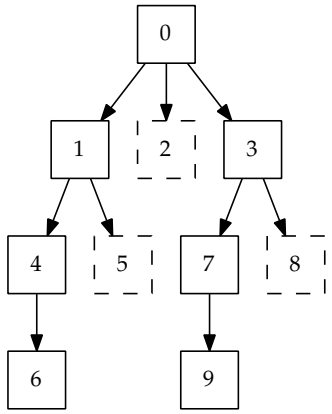


Figure 4.1: Execution Tree Example

```
 0 AddNode 0 -    14 Transition 6
 1 Explore 0      15 Explore 3
 2 Start          16 End 0.42
 3 AddNode 1 0    17 Start
 4 AddNode 2 0    18 Transition 3
 5 AddNode 3 0    19 AddNode 7 3
 6 Explore 1      20 AddNode 8 3
 7 Transition 1   21 Explore 7
 8 AddNode 4 1    22 Transition 7
 9 AddNode 5 1    23 AddNode 9 7
10 Explore 4      24 Explore 9
11 Transition 4   25 Transition 9
12 AddNode 6 4    26 End 0.29
13 Explore 6
```

Figure 4.2: Exploration Trace Example

4.2 Methods

This section presents a number of techniques for estimating the length of an exploration carried out by the dynamic partial order reduction (DPOR) [51] algorithm. Notably, each execution explored by the DPOR algorithm starts from the initial state of the program (root node of the execution tree) and ends in some final state of the program (a leaf node of the execution tree). An estimation technique can thus be described as an algorithm that operates over an exploration trace described in Section 4.1. In particular, a sequential scan of an exploration trace can be used to simulate gradual exploration of the execution tree, maintaining the exploration status of each node.

The presentation of the estimation techniques is divided into three logical components. First, the *strategy* component is used to determine how to treat nodes of the execution tree that have not been scheduled for exploration yet. Second, the *estimator* component is used to determine how to combine a strategy and the exploration information gathered so far to compute an intermediate estimate. Third, the *fit* component is used to aggregate the sequence of intermediate estimates computed so far to produce the final estimate.

4.2.1 Strategies

In general, the DPOR algorithm does not explore all subtrees of an execution tree because it may identify exploration of some parts of the execution tree as redundant. Further, the information about which subtrees need to be explored is revealed only as the exploration makes its way through the state space. The first step towards estimating the length of an exploration, it thus to determine what portions of a partially explored execution tree will be eventually explored.

To this end, this chapter considers three *strategies* for computing the function $F : V \to \mathcal{P}(V)$, which for a node v of a partially explored execution tree, estimates the set of children of node v that the strategy expects to schedule for exploration in the future. Consequently, at any point of the exploration, the function F can be used to identify the subset of nodes of a partially explored execution tree that the strategy believes will be eventually explored.

1. *Oracle* – Assumes perfect knowledge about which nodes will be explored. The function F is computed by pre-processing the exploration trace. This strategy is infeasible in practice but this chapter considers it for comparison purposes.
2. *Lazy* – Assumes that a node will not be explored unless it has been already scheduled for exploration. In other words, $F(v) = \emptyset$ for all nodes v.
3. *Eager* – Assumes that a node will be explored unless the exploration has finished exploring its parent without scheduling the node for exploration. In other words, $F(v)$ is equal to the set of children of node v that are yet to be scheduled for exploration if the exploration of the subtree of v has not finished and $\emptyset$ otherwise.

Space and Time Complexity

The oracle strategy, which is used for benchmarking purposes only, requires linear pre-processing time and has $\mathcal{O}(n)$ space overhead, where n is the size of the execution tree. The lazy and the eager strategies have no overhead.

4.2.2 Estimators

After a strategy is used to determine which parts of a partially explored execution tree will be explored, this information is passed onto an *estimator*, responsible for producing an intermediate estimate. This chapter considers two different estimators based on previous work [80]: the *weighted backtrack* estimator and the *recursive* estimator.

Weighted Backtrack Estimator

The weighted backtrack estimator (WBE) is an online variant of Knuth's offline technique [85] for tree size estimation. WBE uses the length of each explored branch weighted by the probability it is explored by a random exploration (assuming uniform distribution over edges) to predict the size of the tree. To adapt WBE to exploration length estimation, the length of each branch is replaced with the time used to explore it. Formally, WBE updates its estimate every time the DPOR algorithm explores a branch, setting the estimate to:

$$estimate = \frac{\sum_{b \in B} t(b)}{\sum_{b \in B} p(b)}$$

where B is the set of explored branches, $t(b)$ is the time used to explore a branch, and $p(b)$ is the probability of exploring the branch, presented next.

For a branch $b = (v_1, \ldots, v_n)$, where v_i is the i-th node along the branch b, the probability $p(b)$ is defined as:

$$\prod_{i=1}^{n-1} \frac{1}{|E(v_i)| + |S(v_i)| + |F(v_i)|}$$

where $E(v_i)$ is the set of explored children of node v_i, $S(v_i)$ is the set of children of node v_i scheduled for exploration, and $F(v_i)$ is determined by the strategy (cf. Section 4.2.1).

Recursive Estimator

The recursive estimator (RE) is an online technique that estimates the size of an unexplored subtree using the arithmetic mean of the estimated sizes of its (partially) explored siblings. To adapt RE to exploration length estimation, the size of each subtree is replaced with the time used to explore it. Formally, when the DPOR algorithm explores a branch $b = (v_1, \ldots, v_n)$, RE updates the exploration length estimate for every node along the branch b in a bottom-up manner, setting the estimate to:

$$estimate(v_i) = \begin{cases} \left(1 + \frac{|S(v_i)|+|F(v_i)|}{|E(v_i)|}\right) \times \left(\sum_{v \in E(v_i)} estimate(v)\right) & \text{if } i \neq n \\ t(b) & \text{if } i = n \end{cases}$$

where $E(v_i)$, $S(v_i)$, and $F(v_i)$ have the same meaning as above and $t(b)$ is the time used to explore the branch b.

Space and Time Complexity

The WBE estimate needs to be updated upon two events: 1) when a new branch is explored and 2) when a node is scheduled for exploration. To avoid recomputation of all of the values $p(b)$ and $t(b)$ for each update of the estimate, one can store, for each node v of the execution tree, the sums:

$$\sum_{b \in B(v)} p(b) \text{ and } \sum_{b \in B(v)} t(b)$$

where $B(v)$ is the set of explored branches that contain the node v.

When a new branch is explored, the aggregate probability and time values make it possible to update the WBE estimate by updating only the values for the nodes along the current branch. Note that although the time complexity of updating the WBE estimate for a new branch is linear in the depth of the execution tree, this time complexity is amortized over the time used to explore the branch to a constant.

When a new node is scheduled for exploration, the aggregate probability and time values of the nodes along the exploration frontier need to be updated. The time complexity of this operation is $\mathcal{O}(d)$, where d is the depth of the execution tree. Unlike exploring a new branch, the cost associated with updating the estimate when a node is scheduled for exploration does not have constant amortized complexity. In practice, the worst case scenario is rare and our evaluation did not experience significant overhead.

In contrast to the WBE estimate, the RE estimate needs to be updated only when a new branch is explored. The time complexity of this operation is linear in the depth of the execution tree but amortizes to $\mathcal{O}(1)$ over the time used to explore the branch.

4.2.3 Fits

As the exploration progresses, new and presumably more accurate intermediate estimates are computed. Previous work [80] considers the intermediate estimates produced at distinct time points in isolation, using only the latest estimate for decision making. In comparison, this chapter treats the intermediate estimates as a progression, fitting it with a function $f(t)$ and using the solution of the equation $f(t) = t$ as the final estimate.

Both intuition and experience suggests that the exploration length estimates can be initially inaccurate but over the course of an exploration converge to the correct value. Consequently, this chapter uses a method to interpolate the progression of estimates over time. The interpolation method is based on the Marquardt-Levenberg algorithm [90, 100] for weighted non-linear least-square fitting. The algorithm is used to find the values for coefficients of the function that best fits the sequence of intermediate estimates. To reflect the increasing confidence in estimates over time, the least-square fitting is weighted, using t as the weight for an estimate at time t. This chapter considers four different fitting functions:

1. *Empty* function: This scheme does no fitting. Instead, it emulates previous work [80], using the latest intermediate estimate as the final estimate.
2. *Constant* function: $f(t) = c$. The advantage of using a constant function is that the final estimate computed by solving the equation $f(t) = t$ is guaranteed to be a positive number. The disadvantage of using a constant function is that it does not detect trends.
3. *Linear* function: $f(t) = a * t + b$. The advantage of using a linear function is its ability to detect linear trends in the sequence of intermediate estimates. However, in pathological cases, the final estimate computed by solving the equation $f(t) = t$ might be a negative number.
4. *Logarithmic* function: $f(t) = a * ln(t) + b$. The advantage of using a logarithmic function is its ability to detect non-linear trends in the sequence of intermediate estimates. However, in pathological cases, the equation $f(t) = t$ might have no solution, preventing computation of the final estimate.

Space and Time Complexity

The space and time complexity of fitting the empty function is $\mathcal{O}(1)$. The space complexity of fitting the other functions is linear in the length of the sequence being fitted. As for the time complexity, the Marquardt-Levenberg algorithm uses a hill climbing technique. A single iteration of the algorithm is linear in the length of the sequence being fitted. The number of iterations is potentially unbounded and depends

on the desired precision. In other words, fitting a function every time a new intermediate estimate results in time complexity that is quadratic in the length of the sequence being fitted. For long sequences of intermediate estimates, this overhead can be reduced by employing reservoir sampling [154], which maintains a constant-size set of random samples of the sequence of intermediate estimates.

4.2.4 Resource Allocation Policies

A test suite of a large-scale system under development is expected to consist of many systematic tests of varied complexity. It is not unreasonable to expect that the resources available for running these tests are not always sufficient to complete all tests in the time allotted for testing. This subsection describes how to use test length estimation to map testing objectives to effective policies for allocation of scarce testing resources.

All policies maintain a priority queue of systematic tests used to identify which systematic test to advance next. After a systematic test is identified, a new branch of its execution tree is explored, and its estimate and priority queue position are updated. This process is repeated for as long as there are unfinished systematic tests and the time allotted for testing has not expired. Different testing objectives are distinguished by the function used for ordering the elements of the priority queue. In particular, this chapter considers two different testing objectives:

1. *Maximize the number of completed tests*: This objective is motivated by the guarantee realized upon completion of a systematic test. For this objective, elements of the priority queue are ordered by their estimated time to completion, which is computed by subtracting the elapsed time from the test length estimate. In other words, the resource allocation follows the "shortest remaining time first" policy.
2. *Achieve even coverage across tests*: This objective is motivated by allocating testing resources proportionally to the length of each systematic test. For this objective, elements of the priority queue are ordered by their estimated progress, which is computed by dividing the elapsed time by the test length estimate. In other words, the resource allocation follows the "smallest coverage first" policy.

Space and Time Complexity

The space complexity of maintaining a priority queue is $\mathcal{O}(k)$, where k is the number of systematic tests. The time complexity of identifying the top element of a priority queue is $\mathcal{O}(1)$ and the time complexity of updating the value of the top element is $\mathcal{O}(\log k)$.

4.3 Evaluation

The goal of this section is to evaluate 1) the accuracy of the estimation techniques and 2) the efficiency of resource allocation mechanisms described in Section 4.2.

To this end, this section uses a set of 10 exploration traces recently released by Google [140]. The evaluation uses a trace simulator that reads these traces, simulates the exploration of the execution tree, and computes an intermediate estimate every time a branch of the execution tree is explored. Additionally, the evaluation uses an implementation of the Marquardt-Levenberg algorithm that inputs a progression of intermediate estimates and a function template and computes what function coefficients to use to fit the function template with the progression.

4.3.1 Exploration Traces

The set of exploration traces used for our evaluation is summarized in Table 4.1. The table identifies the name of a test, the number of nodes of its execution tree, the number of branches of its execution tree, and the total time used at Google for the exploration. The unit of time is abstract as the timing of the exploration traces has been scaled by a magic constant during the trace anonymization process [140].

Test Name	# Nodes	# Branches	Time
Resource(2)	110	8	2.42
Resource(3)	4,914	279	86.15
Resource(4)	248,408	12,054	4,438.54
Scheduling(6)	29,578	720	250.80
Scheduling(7)	237,528	5,040	1,956.32
Scheduling(8)	2,142,164	40,320	19,868.90
Store(3,3,7)	20,577	924	392.78
Store(3,3,8)	88,386	3,790	1,715.49
Store(3,3,9)	230,747	9,230	2,613.85
TLP	4,201,044	27,200	24,197.60

Table 4.1: Test Statistics

The Resource(x) tests are representative of a class of tests that evaluate interactions of x different users that acquire and release resources from a pool of x resources. The Scheduling(x) tests are representative of a class of tests that evaluate handling of x concurrent scheduling requests. The Store(x, y, z) tests are representative of a class of tests that evaluate interactions of x users of a distributed key-value store with y front-end nodes and z back-end nodes. Finally, the TLP test is representative of a class of tests that perform scheduling work.

4.3.2 Accuracy Evaluation

To evaluate estimation accuracy, an exploration is advanced for some time, generating intermediate estimates using different combinations of strategies and estimators. The intermediate estimate progression is then fitted by the various fit functions and the solutions of the $f(t) = t$ equations are compared to the known correct value.

The experiment collected progressions of intermediate estimates computed during simulation of the exploration traces RESOURCE(4), SCHEDULING(8), STORE(3,3,9), and TLP, which are representative of the full set. For each test, the complete progression of intermediate estimates was collected for all six possible combinations of the oracle, lazy, and eager strategies and the weighted backtrack and recursive estimators. For each progression, the empty, constant, linear, and logarithmic fits were then computed using the initial 1%, 5%, and 25% of the progression.

Figure 4.3 depicts the results of the experiment. Given the correct value *correct*, the accuracy of the estimate *estimate* is computed as follows:

$$accuracy(correct, estimate) = \begin{cases} 100 * (correct/estimate)\% & \text{if } estimate > correct \\ 100 * (estimate/correct)\% & \text{otherwise} \end{cases}$$

For example, if the correct value is 100, then the accuracy of the estimate 50 is 50%, while the accuracy of the estimate 400 is 25%. In other words, the above measure of accuracy does not distinguish between under-estimation and over-estimation, a provision used to simplify the presentation.

Figure 4.3 consists of three bar graphs, one for each percentage at which the fit was computed. Each of the bar graphs depicts the accuracy achieved for the 96 different combinations of a test, a strategy, an estimator, and a fit, with the results clustered by fit and test. Note that the vertical axis depicting the accuracy is in logarithmic scale. In some cases, the application of a fit did not generate a positive solution for the estimate and in such cases the bar is missing.

Strategy Comparison

The evaluation indicates that, unlike the eager strategy, the lazy strategy is consistent with the oracle strategy. In other words, the lazy and the oracle strategies tend to agree on which children will be explored in the future; in fact, in the case of the SCHEDULING tests and the TLP test, the lazy strategy and the oracle strategy are indistinguishable. At the same time the evaluation indicates that the the oracle strategy does not always produce the most accurate results. Surprisingly, the eager strategy occasionally produces the most accurate results. The poor performance of estimation techniques based on the oracle strategy is attributed to under-estimation introduced by the estimators and fits, which the eager strategy compensates for with its over-estimation.

Estimator Comparison

The evaluation does not indicate that either of the two estimators consistently outperforms the other one. The weighted backtrack estimator, however, produces estimates that are, except for two cases, within an order of magnitude of the correct value. This cannot be said about the recursive estimator. This makes the weighted backtrack estimator more a robust choice.

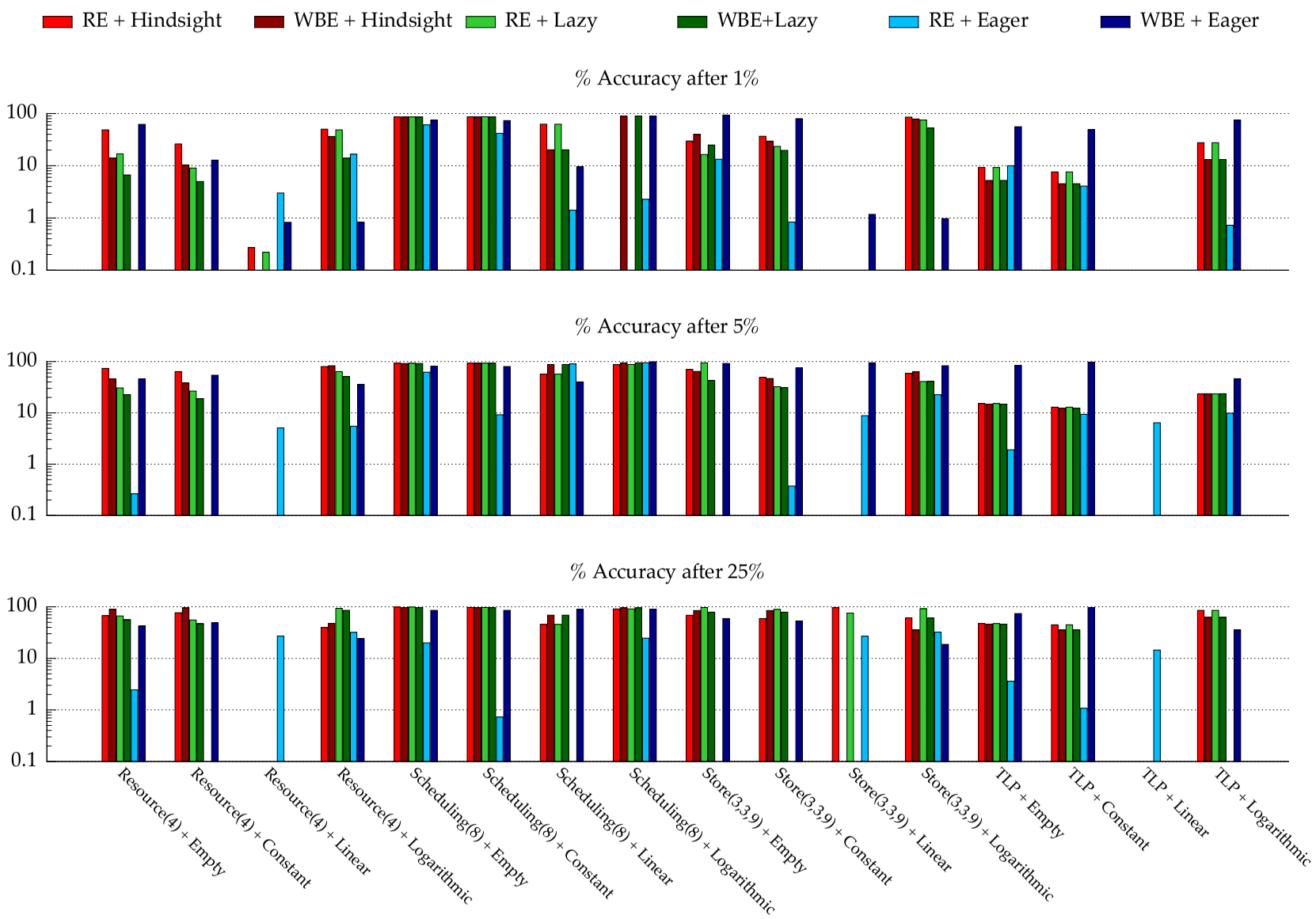


Figure 4.3: Accuracy of Estimation Techniques

Fit Comparison

The linear fit often fails to generate a positive solution, which makes it unreliable. Comparing the empty fit to the constant fit, the empty fit produces equivalent or better results in most of the cases, which confirms our intuition that the intermediate estimates grow more accurate with time. The logarithmic fit generates a solution in all but two cases, and compared to the empty fit, produces equivalent or better results.

Technique	Resource(4)	Scheduling(8)	Store(3,3,9)	TLP	Mean
e+w+e	62.11%	75.19%	93.46%	55.87%	68.97%
l+r+e	16.95%	86.21%	16.45%	9.24%	16.69%
l+r+l	48.78%	N/A	75.19%	27.70%	42.92%
l+w+l	14.03%	89.29%	52.91%	13.18%	22.57%

Table 4.2: Accuracy of Best Techniques after 1%

e+w+e	45.66%	80.65%	91.74%	84.03%	69.93%
l+r+e	30.68%	92.59%	94.34%	15.22%	33.44%
l+r+l	63.69%	87.72%	40.82%	23.42%	42.37%
l+w+l	51.02%	94.34%	41.15%	23.58%	41.32%

Table 4.3: Accuracy of Best Techniques after 5%

e+w+e	42.19%	85.47%	59.17%	74.07%	60.61%
l+r+e	66.67%	99.01%	97.09%	47.39%	70.92%
l+r+l	93.46%	90.91%	91.74%	85.47%	90.09%
l+w+l	85.47%	95.24%	60.61%	62.50%	72.99%

Table 4.4: Accuracy of Best Techniques after 25%

Overall Comparison

To analyze the overall accuracy, Tables 4.2, 4.3, and 4.4 take a closer look at the best-performing techniques, reporting the accuracy after 1%, 5%, and 25% of the exploration respectively. The acronyms in the Technique column have the following meaning: e+w+e stands for the eager strategy, the weighted backtrack estimator, and the empty fit, l+r+e stands for the lazy strategy, the recursive estimator, and the empty fit, l+r+l stands for the lazy strategy, the recursive estimator, and the logarithmic fit, and l+w+l stands for the lazy strategy, the weighted backtrack estimator, and the logarithmic fit.

Interestingly, the accuracy of the e+w+e technique does not improve over time, while the accuracy of the other techniques do. To understand this phenomenon better, Figure 4.4 depicts the progression of intermediate estimates over time for selected

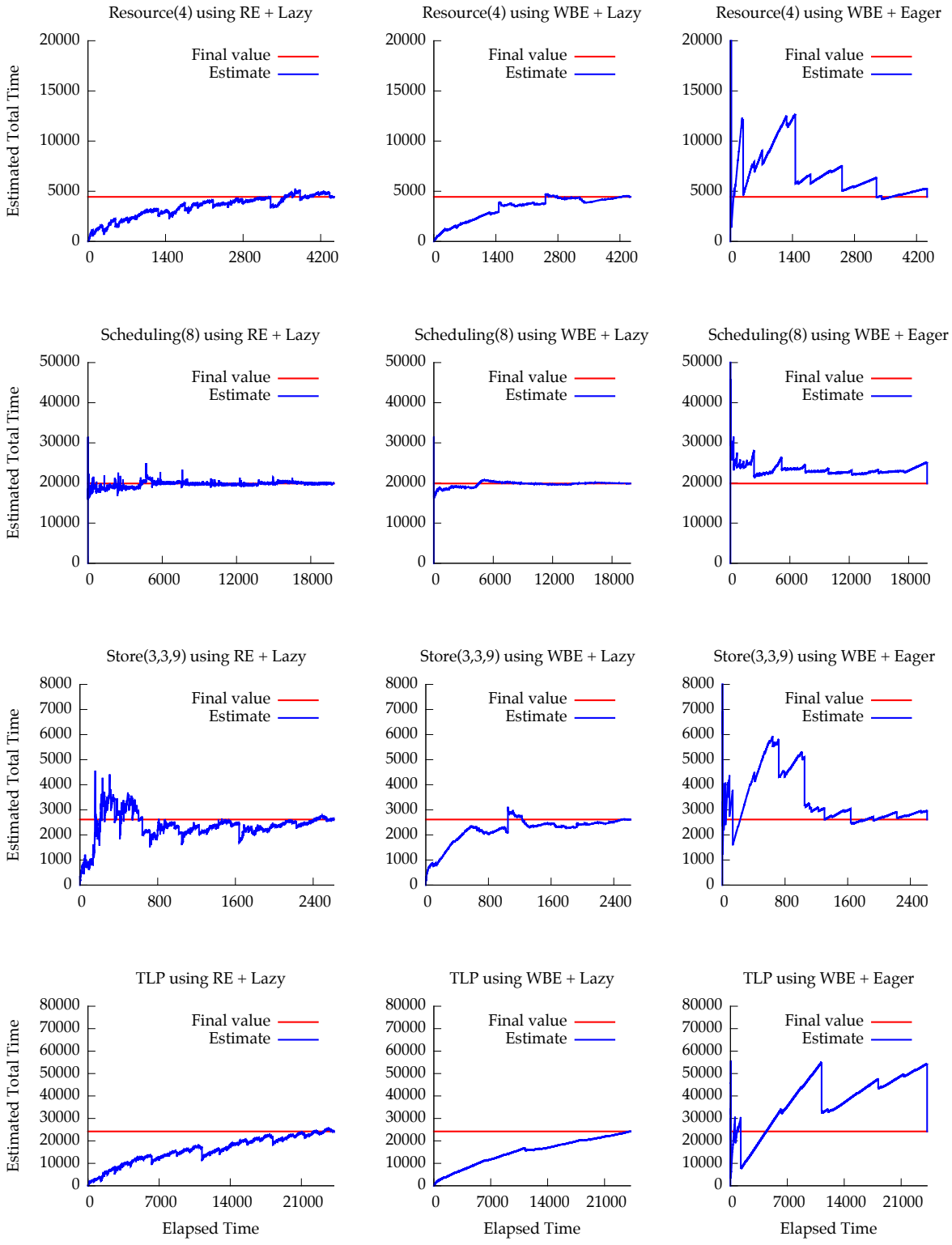


Figure 4.4: Evolution of Intermediate Estimates Over Time

tests and best-performing techniques. Each graph also includes a horizontal line that identifies the correct value.

Figure 4.4 reveals that the intermediate estimates based on the eager estimator can change quickly over time, which suggests that the accuracy of estimation techniques based on the combination of the eager strategy and the empty fit can be sensitive to the time at which their estimates are computed. Further, note that eager strategy typically results in over-estimation, while the lazy strategy typically results in under-estimation.

In summary, many techniques examined in this chapter consistently achieve average accuracy above 60% after exploring as little as 1% of the state space. At first glance the E+W+E technique seems to be the best but further investigation suggests the L+W+L technique is a more robust choice. This conclusion is based on two facts: 1) the accuracy of fitting a logarithm to a progression of intermediate estimates generated by the combination of the lazy strategy and weighted backtrack estimator and 2) the resilience of the logarithmic fit to spikes and drops in the progression of intermediate estimates.

4.3.3 Efficiency Evaluation

This subsection evaluates the potential of test length estimation to help implement allocation policies that target the two testing objectives described in Section 4.2: maximizing the number of completed tests and achieving even coverage.

Maximizing # of Completed Tests

To evaluate how well test length estimation techniques help in maximizing the number of completed tests, a resource allocation simulator was created. This simulator maintains a priority queue that tracks the remaining time estimated for each exploration trace and a scheduler that uses the previously described trace simulator to advance exploration in the order dictated by the priority queue.

The evaluation examines all combinations of the weighted backtrack and recursive estimators and the oracle, eager and lazy strategies. However, given the frequency of estimate computation in this experiment, only the empty fit is considered in order to limit the simulation duration.

For the sake of comparison, two additional allocation policies are examined: 1) a *round-robin* policy, which selects the next test to advance using round-robin, representing a baseline approach commonly used in practice, and 2) an *optimal* policy, which knows the true length of each tests and executes tests from the shortest to the longest.

Lastly, the allocation simulator was provided with unlimited time and, for each estimation technique, recorded the time at which the 10 different exploration traces are completed. In other words, instead of using a fixed time budget, the recorded data can be used to derive the results for an arbitrary fixed time budget.

Figure 4.5 presents a bar graph, which for each allocation policy plots the time needed to complete a certain number of tests. The horizontal axis shows the number of completed tests, while the vertical axis shows time units on a logarithmic scale.

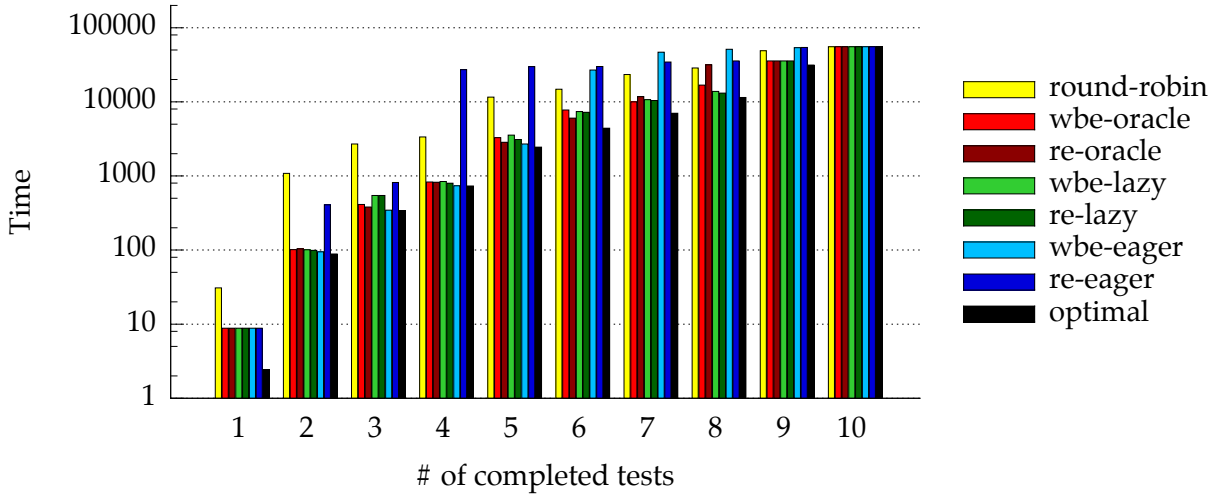


Figure 4.5: Maximizing # of Completed Tests

Completed	re-eager	re-lazy	wbe-eager	wbe-lazy	optimal
1	0.29	0.29	0.29	0.29	0.08
2	0.38	0.09	0.09	0.09	0.08
3	0.30	0.20	0.13	0.20	0.13
4	8.08	0.24	0.22	0.25	0.22
5	2.58	0.27	0.23	0.31	0.21
6	2.03	0.49	1.82	0.50	0.30
7	1.47	0.45	2.00	0.46	0.30
8	1.24	0.46	1.79	0.48	0.40
9	1.11	0.73	1.10	0.73	0.64
10	1.00	1.00	1.00	1.00	1.00
Mean	1.85	0.42	0.87	0.43	0.34

Table 4.5: Performance of Allocation Algorithms

Note that the measurement is oblivious to the order in which the tests finish and only compares the times required by different algorithms to complete a certain number tests.

The experiment results indicate that the policies that incorporate the eager strategy tend to perform poorly and, in some cases, need more time to complete a certain number of tests than the round-robin policy. In contrast to that, the policies that incorporate the oracle and the lazy strategies match the performance of the optimal policy, requiring a fraction of the time required by the round-robin policy to complete first few tests. Further, the experiment results indicate than in the context of this experiment, the choice of the estimator is not significant.

Table 4.5 reports the fractions of the time required by the best performing algorithms with respect to the baseline round-robin policy. The rows report these fractions for each possible number of completed tests and the last row reports their arithmetic mean.

These numbers indicate that the policies based on a combination of the lazy strategy, either of the two estimators, and the empty fit, reduce the time needed by the baseline round-robin policy to complete a certain number of tests by 2.38× on average, coming close to the optimal value of 2.94×.

Achieving Even Coverage

To evaluate how well test length estimation techniques help in achieving even coverage across the test suite, the resource allocation simulator from the previous subsection was modified to order elements of its priority queue by the estimated coverage instead of the estimated remaining time.

Similarly to the previous experiment, all combinations of the strategies, the estimators, and the empty fit were examined and a baseline was modeled using a policy based on the round-robin order. In contrast to the previous experiment, a scarcity of testing resources was simulated by setting the time budget to $5,000$ time units, representing 10% of the time needed to fully explore all tests.

For each test, the experiment measured the *coverage error* computed as the relative difference between the realized and even coverage. Formally, let $time_{sum}$ be sum of the actual runtimes of all tests in a test suite and $time_{budget} \leq time_{sum}$ be the time allotted for testing, then *even* coverage equals:

$$even = \frac{time_{budget}}{time_{sum}}$$

Further, let $time_{full}(t)$ be the time needed to fully explore the test t and $time_{explored}(t)$ be the time spent exploring the test t, then the *realized* coverage of the test t equals:

$$realized(t) = \frac{time_{explored}(t)}{time_{full}(t)}$$

and the *coverage error* of the realized coverage $realized(t)$ with respect to the even coverage $even$, denoted $error(realized(t), even)$ is defined as:

$$error(realized(t), even) = \begin{cases} realized(t)/even & \text{if } even > realized(t) \\ even/realized(t) & \text{otherwise} \end{cases}$$

For example, if the target even coverage is 10%, then the coverage error of the realized coverage 5% is 2, while the coverage error of the realized coverage 40% is 4. In other words, the above measure of coverage error does not distinguish between falling short of and exceeding the target even coverage, a provision used to simplify the presentation.

Figure 4.6 depicts a bar graph, which for each test contains a cluster of the coverage errors achieved by each allocation policy. The horizontal axis identifies the test cluster, while the vertical axis plots the coverage error.

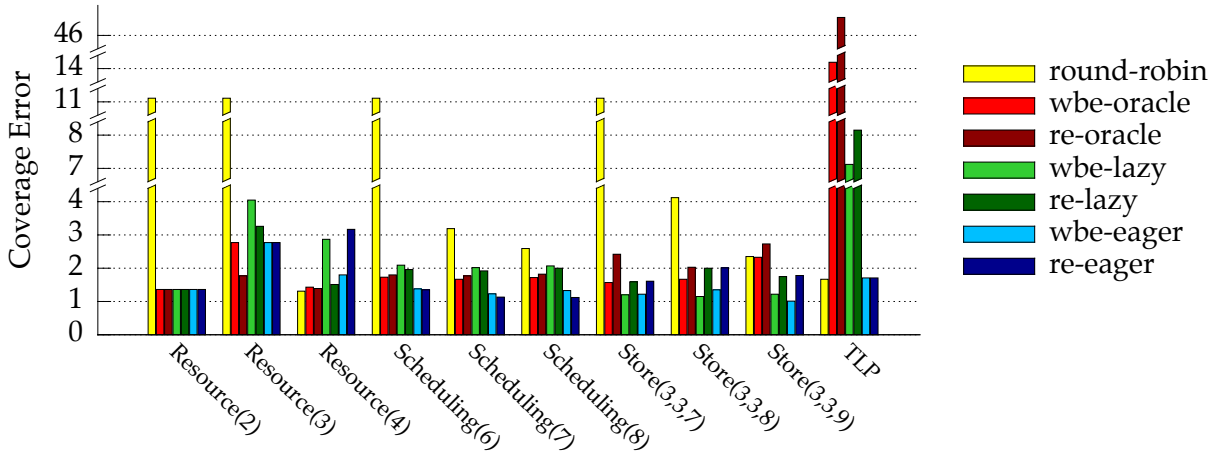


Figure 4.6: Achieving Even Coverage

Test	round-robin	re-eager	re-lazy	wbe-eager	wbe-lazy
Resource(2)	11.11	1.36	1.36	1.36	1.36
Resource(3)	11.11	2.77	3.26	2.77	4.05
Resource(4)	1.31	3.17	1.51	1.80	2.87
Scheduling(6)	11.11	1.35	1.96	1.38	2.09
Scheduling(7)	3.19	1.13	1.92	1.23	2.02
Scheduling(8)	2.59	1.12	2.00	1.33	2.07
Store(3,3,7)	11.11	1.61	1.59	1.22	1.20
Store(3,3,8)	4.12	2.01	2.00	1.35	1.15
Store(3,3,9)	2.35	1.78	1.75	1.01	1.22
TLP	1.67	1.71	8.15	1.71	7.12
Mean	5.97	1.80	2.55	1.52	2.51

Table 4.6: Coverage Error of Allocation Algorithms

The experiment results indicate that the policies that incorporate the oracle and lazy strategies tend to perform poorly and in some cases achieve even higher coverage error than the baseline policy. In contrast to that, the policies based on the eager strategy often produce the best result. Further, the policies that incorporate the weighted backtrack estimator tend to perform better than those that incorporate the recursive estimator.

Table 4.6 compares the coverage errors of the best-performing policies to the coverage error achieved by the baseline round-robin policy. The rows report the coverage errors for individual tests and the last row reports their arithmetic mean. The experiment indicates that the best allocation policy is based on a combination of the eager strategy, the weighted backtrack estimator, and the empty fit, and this policy reduces the average coverage error of the baseline round-robin policy from 5.97 down to 1.5.

4.4 Related Work

State Space Estimation

This chapter adapts work on estimating search tree size by Kilby et al. [80] to the problem of estimating tree exploration length. In their evaluation, Kilby et al. present the accuracy of their estimation techniques achieved during exploration of search trees corresponding to both decision and optimization problems.

Similar to our work, the techniques of Kilby et al. can be described as online and passive. However, in contrast to search tree size estimation, our techniques have to address the dynamic nature of the DPOR algorithm. Nonetheless, comparing the accuracy of our estimation techniques to Kilby et al. estimation techniques on search trees, our techniques perform equally well on a harder problem.

Taleghani and Atlee [146] studied the problem of state space coverage estimation for explicit-state model checking. Their solution is based on Monte Carlo techniques and complements state space exploration with random walks to estimate the ratio between visited and unvisited states. They implemented their technique for the Java PathFinder (JPF) [153] model checker and used a collection of Java programs for evaluation.

In contrast to our work, the technique of Taleghani and Atlee is limited to stateful approaches and can be described as offline and active. More precisely, the estimate is computed at the end of the exploration and the computation relies on a particular exploration strategy. Although Taleghani and Atlee do not explicitly mention whether their experiments were carried out in the context of state space reduction, since JPF supports it, we assume they were. If that is the case, our technique performs equally well on a similar problem but does not rely a specific exploration algorithm.

Resource Allocation

Dynamic allocation of resources to a collection of independent tasks is both a well studied theoretical problem [150] and a practical problem addressed by a range of systems ranging from batch schedulers such as the Maui scheduler [77] to platforms for sharing resources in a data center [69, 129].

While in practice [69, 77, 129] tasks are usually running concurrently on a cluster, this paper uses a simple model that schedules tasks sequentially. This simplification is justified by the unique nature of the execution tree exploration, which typically consists of many executions of the same test. Recording progress of each exploration using an execution tree enables fine-grained interleaving of concurrent explorations. In addition, different executions of the same test can be explored in parallel, enabling linear speed-up [139] (cf. Chapter 5). Representing a cluster of machines as a sequence of machine cycles is a reasonable abstraction of a large set of small independent tasks.

Another unique aspect of resource allocation among systematic tests is how value is measured. In general, value of a task accrues when it finishes, which is reflected in scheduling objectives that minimize average task latency [77] or maximize task

throughput [150]. In contrast, for systematic tests, value could accrue as state space is being covered, which is reflected in a scheduling policies that allocate resources proportionally to the estimated test length.

Interestingly, the problem of deciding which systematic test to advance next is similar to the problem of multi-armed bandit [151]. In short, a multi-armed bandit problem for a gambler is to decide which arm of an n-slot machine to pull to maximize his total reward in a series of trials. To bridge the research on multi-armed bandits with our work, one needs to define a reward function for systematic tests. The search for such a function is an interesting avenue for future work.

4.5 Conclusions

This chapter presents a solution to the problem of allocating scarce resources to a collection of systematic tests. The solution comes in the form of different allocation policies based on techniques for estimating the length of a systematic test.

In the context of this chapter, an estimation technique consists of three logical components a strategy, an estimator, and a fit. This chapter has considered three strategies: eager, oracle (infeasible in practice), and lazy; two estimators: weighted backtrack and recursive; and four fits: empty, constant, linear, and logarithmic.

The evaluation of the different combinations of these components reveals that the overall average accuracy of our best estimation techniques is upwards of 60% after exploring as little as 1% of the state space. To further improve the estimation accuracy, future work could take advantage of common structural properties, such as the power-law of random networks graphs [10], of execution trees.

Besides evaluating the accuracy of estimation techniques, this chapter also investigated the ability of these techniques to map a testing objective to a policy for allocating scarce resources to a collection of systematic tests. Two testing objectives were considered: 1) maximizing the number of completed tests and 2) achieving even coverage across different tests. For each of these objectives an allocation policy parametrized by an estimation technique was designed and its performance evaluated against the performance of a baseline policy.

The experimental evaluation of these allocation policies revealed that while the lazy strategy outperformed the eager strategy at meeting the first testing objective, the eager strategy outperformed the lazy strategy at meeting the second testing objective. This indicates that the lazy strategy achieves better accuracy when estimating the absolute length of an individual test, while the eager strategy achieves better accuracy when estimating the ratio between lengths of different tests.

Further, the evaluation demonstrated that, for the testing objectives considered in this chapter, the allocation policies based on test length estimation improve on the round-robin policy. Future work along these lines could experiment with other testing objectives, such as maximizing the number of bugs found, which is not consider in this chapter since the exploration traces from Google do not contain such information.

Chapter 5

Parallel State Space Exploration

This chapter presents a new method for distributed systematic testing of concurrent programs, which pushes the limits of systematic testing to an unprecedented scale. The approach presented here is based on a novel exploration algorithm that 1) enables trading space complexity for parallelism, 2) achieves load-balancing through time-slicing, 3) provides fault tolerance, a mandatory aspect of scalability, 4) scales to more than a thousand parallel workers, and 5) is guaranteed to avoid redundant exploration.

The rest of the chapter is organized as follows. Section 5.1 discusses previous work on distributed systematic testing. Section 5.2 presents a novel exploration algorithm and details its use for distributed systematic testing at scale. Section 5.3 presents an experimental evaluation. Section 5.4 discusses related work and Section 5.5 draws conclusions.

5.1 Background

This section summarizes distributed dynamic partial order reduction (distributed DPOR) [51, 167], a state of the art algorithm for distributed systematic testing of concurrent programs.

Distributed DPOR targets concurrent exploration of branches of the execution tree. The goal of distributed DPOR is to offset the combinatorial explosion of possible permutations of concurrent events through parallel processing.

Parallelization of execution tree exploration seems straightforward: assign different parts of the execution tree to different *workers* and explore the execution tree concurrently. However, as pointed out by Yang et al. [167], such a parallelization suffers from two problems. First, due to the non-local nature in which the DPOR algorithm updates the exploration frontier (cf. Section 2.1.4), different workers may end up exploring identical parts of the state space. Second, since the sizes of the different parts of the execution tree are not known in advance, load-balancing is needed to enable linear speedup.

To address these two problems, Yang et al. [167] proposed two heuristics. Their first heuristic modifies DPOR's lazy addition of nodes to the exploration frontier [51]

so that nodes are added to the exploration frontier eagerly instead. As evidenced by their experiments, replacing lazy addition with eager addition mitigates the problem of redundant exploration of identical parts of the execution tree by different workers. Their second heuristic assumes the existence of a centralized load-balancer that workers can contact in case they believe they have too much work on their hands and would like to offload some of the work. The centralized load-balancer keeps track of which workers are idle and which workers are active and facilitates offloading of work from active to idle workers.

5.2 Methods

While scaling DPOR to a large cluster at Google [139], several shortcomings of the previous work [167] were identified. First, at large scale, distributed exploration must be able to cope with failures of worker processes or machines. Although Yang et al. [167] suggest how fault tolerance could be implemented, they do not quantify how their envisioned support for fault tolerance would affect scalability. Second, although the out-of-band centralized load-balancer of Yang et al. renders the communication overhead negligible, it is not clear whether the centralized approach can be used to support additional features, such as fault tolerance and state space size estimation, without becoming a bottleneck. Third, the load-balancing of Yang et al. uses a heuristic based on a threshold to offload work from active to idle workers. It is likely that for different programs and different numbers of workers, different threshold values should be used. However, Yang et al. provide no insight into the problem of selecting a good threshold. Fourth, their DPOR modification for avoiding redundant exploration is a heuristic, not a guarantee.

This section presents an alternative design for distributed DPOR. The design is centralized and uses a single master and n workers to explore the execution tree. Despite its centralized nature, our experiments show that the design scales to more than a thousand workers. Unlike previous work [167], the design can tolerate worker faults, is guaranteed to avoid redundant exploration, and is based on a novel exploration algorithm that allows 1) trading off space complexity for parallelism and 2) efficient load-balancing through time-slicing.

5.2.1 Partitioned Depth-First Search

The key advantage of using depth-first search for the exploration carried out by the DPOR algorithm (cf. Section 2.1.4) is its favorable space complexity [61]. This fact is the main reason why the bottleneck of state space exploration in existing tools for systematic testing of concurrent programs [51, 107, 137, 160] is the CPU speed and not the memory size. This is hardly surprising given that the time complexity of the DPOR algorithm based on depth-first search exploration is linear in the size of the execution tree and quadratic in its depth, while its space complexity is linear in its depth.

To enable parallel processing, Yang et al. [167] depart from the strict depth-first search nature of state space exploration. Instead, the execution tree is explored using a collection of possibly overlapping depth-first searches and the exploration order is determined by a load-balancing heuristic, which uses an ad-hoc threshold to evenly distribute unexplored subtrees of the execution tree across the worker fleet.

The design presented here uses a novel exploration algorithm, called *n-partitioned depth-first search*, which relaxes the strict depth-first search nature of traditional state space exploration in a controlled manner and, unlike traditional depth-first search, is amenable to parallelization.

The main difference between depth-first search and n-partitioned depth-first search is that the exploration frontier of the new algorithm is partitioned into up to n frontier *fragments* and the new algorithm explores each fragment using the traditional depth-first search, interleaving exploration of different fragments.

For the sake of the presentation, a sequential version of the DPOR algorithm based on the n-partitioned depth-first search is first presented as Algorithm 5. The algorithm maintains an exploration *frontier*, represented as a set of up to n stacks of sets of nodes. The elements of the exploration frontier are referred to as *fragments* and together they form a partitioning of the exploration frontier. The execution tree is explored by interleaving depth-first search exploration of frontier fragments. The algorithm implements this idea by repeating two steps – Partition and Explore – until the execution tree is explored.

During the Partition step, the current frontier is inspected to see whether existing frontier fragments should be and can be further partitioned. A new frontier fragment *should* be created in case there is less than n frontier fragments. A new frontier fragment *can* be created if there exists a frontier fragment with at least two nodes.

The Explore step is given one of the frontier fragments and uses depth-first search to explore the next edge of the subtree induced by the selected frontier fragment (the subtree that contains all ancestors and descendants of the nodes contained in the selected frontier fragment). The UpdateFrontier$(frontier, fragment, node)$ function operates in a similar fashion to the UpdateFrontier$(frontier, node)$ function described in Chapter 2. The main distinction is that after the new version of the function identifies which nodes are to be added to the exploration frontier using the DPOR algorithm, these nodes are added to the current frontier fragment only if they are not already present in some other fragment. This way, the set of sets of nodes contained in each fragment remains a partitioning of the exploration frontier – an invariant maintained throughout our exploration that helps our design to avoid redundant exploration.

5.2.2 Parallelization

This subsection describes how to efficiently parallelize Algorithm 5. First, observe that the presence or absence of the Partition step in the body of the main loop of the algorithm has no effect on the correctness of the algorithm. This allows us to sequence several Explore steps together, which hints at possible distribution of the exploration.

Algorithm 5 SEQUENTIALSCALABLEDYNAMICPARTIALORDERREDUCTION(n, *root*)

Require: n is a positive integer and *root* is the initial program state.
Ensure: All program states reachable from *root* are explored.

```
 1: procedure PARTITION(frontier, n)
 2:     if SIZE(frontier) = n then
 3:         return
 4:     end if
 5:     for all fragment ∈ frontier do
 6:         while SIZE(frontier) < n and SIZE(fragment) > 1 do
 7:             node ← an arbitrary element of a set contained in fragment
 8:             remove node from fragment
 9:             new-fragment ← a new frontier fragment for node
10:             INSERT(new-fragment, frontier)
11:         end while
12:     end for
13: end procedure
14: procedure EXPLORE(node, fragment, frontier)
15:     remove node from TOP(fragment)
16:     UPDATEFRONTIER(frontier, fragment, node)
17:     if CHILDREN(node) not empty then
18:         child ← arbitrary element of CHILDREN(node)
19:         PUSH({child}, fragment)
20:         navigate execution to child
21:     end if
22:     pop empty sets from the fragment stack
23: end procedure
24: frontier ← NEWSET
25: INSERT(PUSH({root}, NEWSTACK), frontier)
26: while SIZE(frontier) > 0 do
27:     PARTITION(frontier, n)
28:     fragment ← an arbitrary element of frontier
29:     node ← an arbitrary element of TOP(fragment)
30:     EXPLORE(node, fragment, frontier)
31:     if SIZE(fragment) = 0 then
32:         REMOVE(fragment, frontier)
33:     end if
34: end while
```

Namely, one could spawn concurrent workers and use them to carry out sequences of EXPLORE steps over different frontier fragments. However, a straightforward implementation of this idea would require synchronization when concurrent workers access and update the exploration frontier, which is shared by all workers. The trick to

Algorithm 6 DistributedScalableDynamicPartialOrderReduction(n, *budget*, *root*)

Require: n is a positive integer, *budget* is a time budget for worker exploration, and *root* is the initial program state.
Ensure: All program states reachable from *root* are explored.

```
 1: procedure ExploreCallback(old-fragment, new-fragment, frontier)
 2:     replace old-fragment of frontier with new-fragment
 3:     mark new-fragment as unassigned
 4:     signal main exploration loop
 5: end procedure
 6: procedure ExploreLoop(fragment, budget, frontier)
 7:     start-time ← GetTime
 8:     repeat
 9:         node ← an arbitrary element of Top(fragment)
10:         Explore(node, fragment, frontier)
11:     until (GetTime − start-time > budget) or (Size(fragment) = 0)
12: end procedure
13: frontier ← NewSet
14: Insert(Push(root, NewStack), frontier)
15: while Size(frontier) > 0 do
16:     Partition(frontier, n)
17:     while exists an idle worker and an unassigned frontier fragment do
18:         fragment ← an arbitrary unassigned element of frontier
19:         Spawn(ExploreLoop, fragment, budget, ExploreCallback)
20:     end while
21:     wait until signaled by ExploreCallback
22: end while
```

overcome this obstacle to efficient parallelization is to give each worker a private copy of the execution tree. As pointed out by Yang et al. [167], such a copy can be concisely represented using the depth-first search stack of the frontier fragment to be explored.

A worker can then repeatedly invoke the Explore function over (a copy of) the assigned frontier fragment. Once the worker either completes the exploration of the assigned frontier fragment or it exceeds the time allotted for its exploration, it reports back with the results of the exploration. The exploration progress can be concisely represented using the original and the final state of the depth-first search stack of the assigned frontier fragment.

Algorithm 6 presents a high-level approximation of the actual implementation of our design for scalable DPOR. The implementation operates with the concept of *fragment assignment*. When a frontier fragment is created, it is unassigned. Later, a fragment becomes assigned to a particular worker through the invocation of the Spawn function. When the worker finishes its exploration, or exhausts the time budget assigned for exploration, it reports back the results, and the fragment assigned to this worker

becomes unassigned again. The results of worker exploration are mapped back to the "master" copy of the execution tree using the ExploreCallback callback function. The time budget for worker exploration is used to achieve load-balancing through time-slicing, which is explained in Section 5.2.4. Further, Section 5.2.5 describes a mechanism to resolve conflicting concurrent updates. The Partition function behaves identically to the original one, but it partitions unassigned fragments only.

Algorithm 6 presents the pseudo-code of the ExploreLoop function, which is executed by a worker. The Explore function is identical to the one used in the sequential version of the algorithm but the *frontier* argument uses a copy of the frontier that contains only the nodes needed to further the exploration of the assigned frontier fragment. The workers are started through the Spawn function which creates a private copy of a part of the execution tree. Structuring the concurrent exploration in this fashion enables both multithreaded and multiprocess implementations of our design.

Since our goal has been to scale the DPOR algorithm to thousands of workers, each worker is implemented as an RPC server running as a separate process. The Spawn function issues an asynchronous RPC request that triggers invocation of the ExploreLoop function with the appropriate arguments at the RPC server of the worker. The response to the RPC request is then handled asynchronously by the ExploreCallback function, which maps the result of the worker exploration into the master copy of the execution tree and prompts another iteration of the main loop of Algorithm 6.

5.2.3 Fault Tolerance

As is commonly done in large distributed applications [32, 57], failure of one out of thousands of nodes will be common [52] and must be handled gracefully, but failure of just one particular node is infrequent enough to be dealt with using re-execution. In accordance with this practice, our design assumes that the master, which is running the main loop of Algorithm 6, will not fail. The workers on the other hand are expected to fail and the exploration can tolerate such events.

In particular, an RPC request issued by the master to a worker RPC server uses a deadline to decide whether the worker has failed. The value of the deadline is set proportionally larger than the value of the worker time budget. When the deadline expires without an RPC response arriving, the master assumes that the worker has failed and unassigns the frontier fragment originally assigned to the failed worker. Other workers are then able to be assigned its exploration.

5.2.4 Load-balancing

The key to high utilization of a worker fleet is effective load-balancing. To achieve load-balancing, our design time-slices frontier fragments among available workers. The availability of frontier fragments is impacted by two factors.

The first factor is the upper bound n on the number of frontier fragments that the distributed exploration creates. This parameter determines the size of the pool of

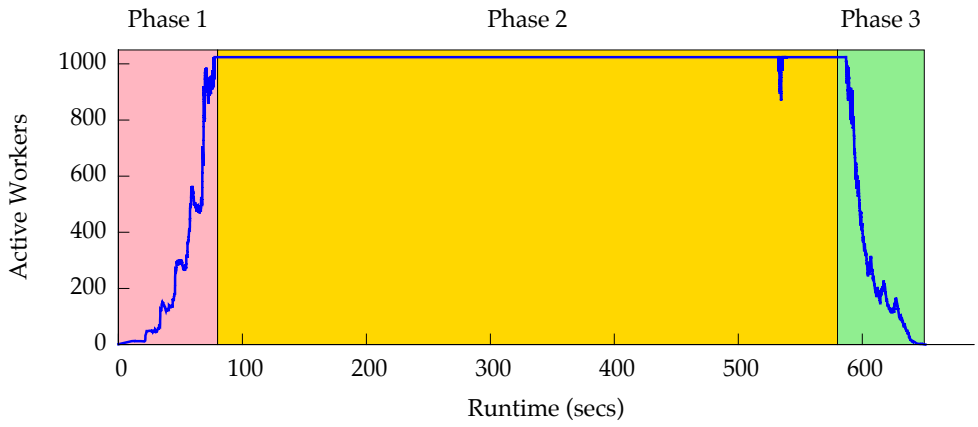


Figure 5.1: Fixed Time Budget Exploration

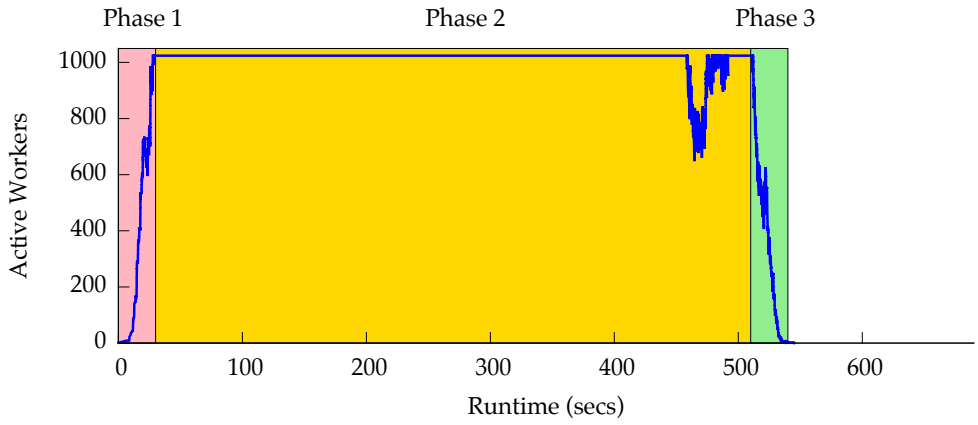


Figure 5.2: Variable Time Budget Exploration

available work units. The higher this number, the higher the memory requirements of the master but the higher the opportunity for parallelism. In our experience, setting n to twice the number of workers works well. Studying dynamic scaling of the number of frontier fragments is an interesting avenue for future work.

The second factor is the size of the time slice used for worker exploration. Smaller time slices lead to more frequent generation of new fragments but this elasticity comes at the cost of higher communication overhead. Our initial design used a fixed time budget, choosing the value of 10 seconds to balance elasticity and communication overhead. However, the initial evaluation of our prototype made us realize that a variable time budget is more appropriate for large worker fleets.

In particular, as the number of workers in a fixed budget scheme increases, a gap between the realized and the ideal speed up opens up. Our intuition led us to believe this could be caused by periods of time during which the exploration has insufficient number of frontier fragments to keep all workers busy. To study this problem, the

master was modified to keep track of the number of active workers over the course of an exploration. Figure 5.1 plots this information for one of our test programs on a configuration with $1,024$ workers and an upper bound of $2,048$ frontier fragments. The figure is representative of other measurements at such a scale.

One can identify three phases of the exploration. In the first phase, the number of active workers gradually increases over 100 seconds until there is enough frontier fragments to keep all workers busy. In the second phase, all workers are constantly kept busy. In the third phase, the number of active workers gradually decreases to zero over 100 seconds. Ideally, the first and the third phase should be as short as possible in order to minimize the inefficiency resulting from not fully utilizing the available worker fleet.

To this aim, one can switch from a fixed time budget to a variable time budget. In particular, if the exploration is configured to use a time budget b, the master actually uses fractions of b proportional to the number of active workers. For example, the first worker will receive a budget of $\frac{b}{n}$, where n is the number of workers. When half of the workers are active, the next worker to be assigned work will receive a budget of $\frac{b}{2}$. The scaling of the time budget is intended to reduce the time before the master has the opportunity to re-partition and load-balance and thus to reduce the duration of the first and the third phase.

Figure 5.2 plots the number of active workers over time for the optimized implementation for the same test as Figure 5.1. For this test, switching to a variable time budget reduced the exploration length from 655 seconds down to 527 seconds. Similar runtime improvements have been achieved for other tests.

5.2.5 Avoiding Redundant Exploration

For clarity of presentation, Algorithm 6 omits a provision that prevents concurrent workers from exploring overlapping portions of the execution tree. This could happen when two workers make concurrent UpdateFrontier calls and add identical nodes to their frontier fragment copies.

To avoid this problem, our implementation introduces the concept of node ownership. A worker exclusively owns a node if it is contained in the original frontier fragment assigned to the worker, or if the node is a descendant of a node that the worker owns. All other nodes are assumed to be shared with other workers and the node ownership restricts which nodes a worker may explore.

In particular, the depth-first search exploration of a worker is allowed to operate only over nodes that the worker owns. When it encounters a shared node during its exploration, the worker terminates its exploration and sends an RPC response to the master indicating which nodes of the frontier fragment are shared. The ExploreCallback function checks the status of the newly discovered shared nodes. If a newly discovered shared node is not part of some other frontier fragment, the node is added to the master copy of the currently processed frontier fragment (ownership is claimed). Otherwise, the ownership of the node has been already claimed and the node is not added to the master copy of the currently processed frontier fragment.

Although this provision could in theory lead to increased communication overhead and decreased worker fleet utilization, our experiments indicate that in practice the provision does not affect performance.

5.3 Evaluation

To evaluate the design presented is Section 5.2, a prototype of the design was implemented as part ETA [139], a tool developed for systematic testing of multithreaded components of the Omega cluster management system [129]. These components are written using a library based on the actors paradigm [2]. To exercise different concurrency scenarios, ETA systematically enumerates different orders in which messages between actors can be delivered.

5.3.1 Experimental Setup

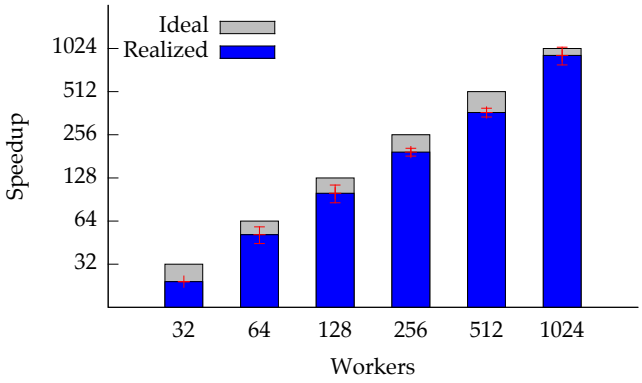
The evaluation used instances of tests from the Omega test suite that exercise fundamental functionality of core components of the cluster management system. The RESOURCE(X) test is representative of a class of actor program tests that evaluate interactions of x different users that acquire and release resources from a pool of x resources. The SCHEDULING(X) test is representative of a class of actor program tests that evaluate interactions of x users issuing concurrent scheduling requests. The STORE(X,Y,Z) test is representative of a class of actor program tests that evaluate interactions of x users of a distributed key-value store with y front-end nodes and z back-end nodes.

Unless stated otherwise, each measurement presented in the remainder of this section presents a complete exploration of the given test and the results report the mean and the standard deviation of three repetitions of the exploration. Lastly, all experiments were carried out inside of a Google data center [65] using stock hardware and running each process on a separate virtual machine.

5.3.2 Faults

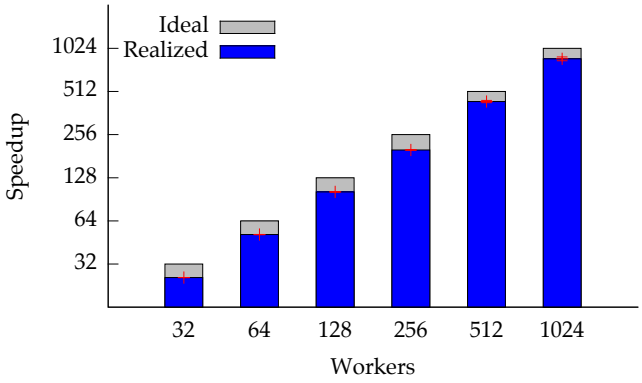
First, the evaluation focused on the ability of the implementation to handle worker failures. To that end, the implementation was extended with an option to inject an RPC fault with a certain probability. When an RPC fault is injected, the master fails to receive the RPC response from a worker and waits for the RPC deadline to expire instead.

Our experiments demonstrated that the runtime increases proportionally to the geometric progression of repeated RPC failures. For example, when RPCs have a 50% chance of failing, the runtime doubles. Since in actual deployments of ETA, RPCs fail with probability well under 1% [52], our support for fault tolerance is practical.



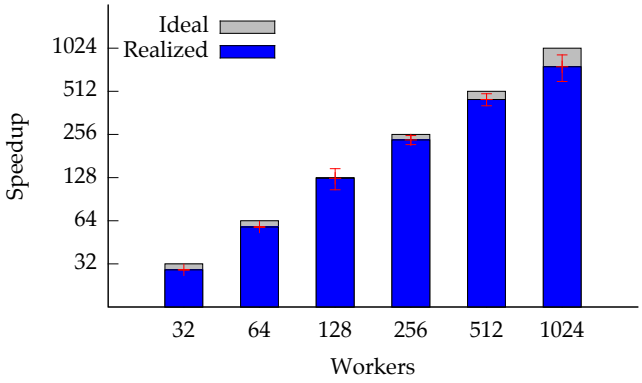
Configuration	Runtime	Speedup
32 Workers	31,105s	24×
64 Workers	14,681s	51×
128 Workers	7,557s	100×
256 Workers	3,874s	194×
512 Workers	2,051s	366×
1024 Workers	820s	916×

Figure 5.3: Scalability Results for Resource(6)



Configuration	Runtime	Speedup
32 Workers	26,646s	29×
64 Workers	13,396s	58×
128 Workers	6,252s	126×
256 Workers	3,332s	233×
512 Workers	1,784s	434×
1024 Workers	1,022s	758×

Figure 5.4: Scalability Results for Scheduling(10)



Configuration	Runtime	Speedup
32 Workers	17,707s	26×
64 Workers	8,870s	51×
128 Workers	4,468s	102×
256 Workers	2,278s	200×
512 Workers	1,046s	436×
1024 Workers	527s	865×

Figure 5.5: Scalability Results for Store(12,3,3)

5.3.3 Scalability

To measure the scalability of the implementation, the time needed to complete an exploration by a sequential implementation of the DPOR algorithm was compared against the time needed to complete the same exploration by our distributed implementation.

Configurations with 32, 64, 128, 256, 512, and 1,024 workers were considered and the algorithm was used to explore the RESOURCE(6), STORE(12,3,3), and SCHEDULING(10) instances of actor program tests; their parameters were chosen to stimulate interesting state space sizes. The time budget b of each worker exploration was set to 10 seconds and the target number of frontier fragments was set to twice the number of workers.

The results of RESOURCE(6), STORE(12,3,3), and SCHEDULING(10) experiments are presented in Figures 5.3, 5.4, and 5.5. Due to the magnitude of the state spaces being explored – 18.5 million, 21 million, and 3.6 million branches respectively – the runtime of the sequential algorithm was extrapolated using a partial run to 209, 215, and 126 hours respectively. The graphs visualize the speedup over the extrapolated runtime of the sequential algorithm and compare it to the ideal speedup. These results evidence strong scaling of our implementation of the DPOR algorithm at a large scale. The largest configuration uses 1,024 workers and our implementation achieves speedup that ranges between 760× and 920×.

5.3.4 Theoretical Limits

Finally, our evaluation focused on projecting the theoretical scalability limits of our implementation. To this end, the memory and CPU requirements of the master – the obvious bottleneck in our centralized design – were measured and projected.

Memory Requirements: The memory overhead is dominated by the cost to store the master copy of the exploration frontier. To estimate the overhead, the amount of memory allocated for the nodes of the execution tree and the exploration frontier data structures was measured over the course of an exploration. For the SCHEDULING(10) test on a configuration with 1,024 workers and an upper bound of 2,048 frontier fragments, the peak amount of the allocated memory was less than 4 MBs. This number is representative of results for other tests at such a scale. Thus, for the current computer architectures, the typical memory systems support scaling to millions of workers.

CPU requirements: With 1024 workers and a 10-second time budget, the master is expected to issue around 100 RPC requests and to process around 100 RPC responses every second. For such a load, the stock hardware running exclusively the master process experienced peak CPU utilization under 20%. Consequently, for the current computer architectures, the CPU requirements scale to around 5,000 workers. To scale our implementation beyond that, one can proportionally increase the time budget, upgrade to better hardware, or optimize the software stack. For instance, one could replace the master with a hierarchy of masters.

5.4 Related Work

Algorithms

Dwyer et al. [45] presented a parallel algorithm that explores the state space using a number of independent randomized depth-first searches to decrease the time needed to

locate an error. In comparison, our parallelization of systematic testing aims to cover the full state space faster.

The work of Staats and Păsăreanu [143] targets parallelization of symbolic execution using randomness and static partitioning. The parallelization presented in this chapter is systematic and uses dynamic partitioning.

Tools

Inspect [166] is a tool for systematic testing of `pthreads` C programs that implements the distributed DPOR [167] discussed in Section 5.1. Unlike our work, the Inspect tool does not support fault tolerance, is not guaranteed to avoid redundant exploration, and has not been demonstrated to scale beyond 64 workers.

DeMeter [67] provides a framework for extending existing sequential model checkers [82, 160] with a parallel and distributed exploration engine. Similar to our work, the framework focuses on efficient state space exploration of concurrent programs. Unlike our work, the design has not been thoroughly described or analyzed and has been only demonstrated to scale up to 32 workers.

Cloud9 [27] is a parallel engine for symbolic execution of sequential programs. In comparison to our work, the state space being explored is the space of possible programs inputs, not schedules. Systematic enumeration of different program inputs is an orthogonal problem to the one addressed by this chapter.

Parallelization of software verification was also investigated in the context of explicit state space model checkers such as **MurPhi** [144], **DiVinE** [12], or **SWARM** [73]. Stateful exploration is less common in implementation-level model checkers as storing a program state explicitly becomes prohibitively expensive.

5.5 Conclusions

This chapter presented a technique that improves the state of the art of scalable techniques for systematic testing of concurrent programs. Our design for distributed DPOR enables the exploitation of a large scale cluster for the purpose of systematic testing. At the core of the design lies a novel exploration algorithm, n-partitioned depth-first search, which has proven to be essential for scaling our design to thousands of workers.

Unlike previous work [167], our design provides support for fault tolerance, a mandatory aspect of scalability, and is guaranteed to avoid redundant exploration of identical parts of the state space by different workers. Further, our implementation and deployment in a real-world system at scale has demonstrated that the design achieves almost linear speed up for up to $1,024$ workers. Lastly, theoretical analysis of our design discussed its scalability limits and proposed solutions for scaling the design for next-generation computer clusters.

Chapter 6

Restricted Runtime Scheduling

Our accelerating computational demand and the rise of multicore hardware have made multithreaded programs increasingly pervasive and critical. Yet, these programs remain extremely difficult to write, test, analyze, debug, and verify. A key reason is that, for decades, the contract between developers and thread runtimes has favored performance over correctness. In this contract, developers use synchronizations to coordinate threads, while thread runtimes can use *any* of the exponentially many thread interleavings, or *schedules*, compliant with the synchronizations. This large number of possible schedules make it more likely that the runtime finds an efficient schedule for a workload. However, ensuring that all schedules are correct is extremely challenging, and a single missed schedule may surface in the least expected moment, triggering concurrency errors responsible for possibly critical failures [91, 97, 122].

To simplify testing, debugging, record-replay, and program behavior replication, a number of recent systems [7, 15, 17, 18, 39, 40, 41, 42, 109] aim to flip this performance-correctness trade-off through *restricted runtime scheduling*, an approach that dramatically limits the number of schedules an execution of a multithreaded program may use.

Restricted runtime scheduling is orthogonal to systematic testing but there is synergy between the two approaches. Restricted runtime scheduling reduces the number of schedules systematic testing needs to check, while systematic testing ascertains that the schedules allowed by restricted runtime scheduling have all been tested.

The goal of this chapter is to demonstrate and quantify the benefits of combining restricted runtime scheduling with systematic testing. To this end, this chapter presents an ecosystem formed by combining dBug with Parrot [39], an implementation of restricted runtime scheduling for POSIX-compliant operating systems. The state space estimation of dBug is then used to contrast the number of schedules dBug needs to check under nondeterministic scheduling to the number of schedules dBug needs to check under Parrot's restricted runtime scheduling.

The rest of this chapter is organized as follows. Section 6.1 provides an overview of restricted runtime scheduling. Section 6.2 describes the integration of dBug and Parrot. Section 6.3 presents the evaluation results, Section 6.4 discusses related work, and Section 6.5 draws conclusions.

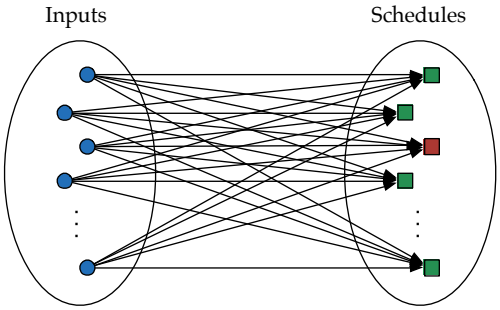


Figure 6.1: Nondeterministic Multithreading

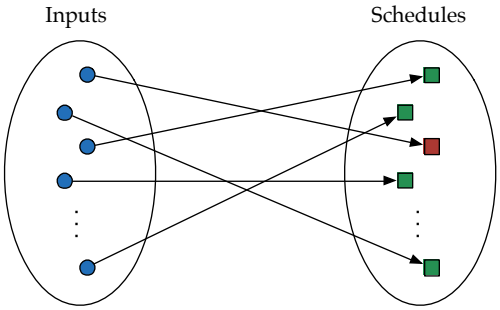


Figure 6.2: Deterministic Multithreading

6.1 Background

This section first presents an evolution of restricted runtime scheduling techniques (§6.1.1) and then describes Parrot [39], an implementation of restricted runtime scheduling for POSIX-compliant operating systems (§6.1.2).

6.1.1 Techniques

To drive performance of multithreaded programs, traditional thread runtimes are allowed to choose any of the exponentially many schedules compliant with thread synchronizations multithreaded programs use. This approach, referred to as *nondeterministic multithreading*, allows many-to-many mapping between program inputs and schedules as depicted in Figure 6.1. The nondeterministic nature of runtime scheduling makes completing the astronomical amount of testing of multithreading infeasible.

In contrast to nondeterministic multithreading, *deterministic multithreading* (DMT) [7, 15, 17, 18, 40, 41, 42, 109] reduces the number of allowed schedules by deterministically mapping each input to a schedule as depicted in Figure 6.2. Thus, with DMT, executions of the same program on the same input and hardware always exhibit the same behavior.

Although DMT enables replication of program behavior, it does not reduce the testing burden as much as one might expect. The reason for this shortcoming of DMT

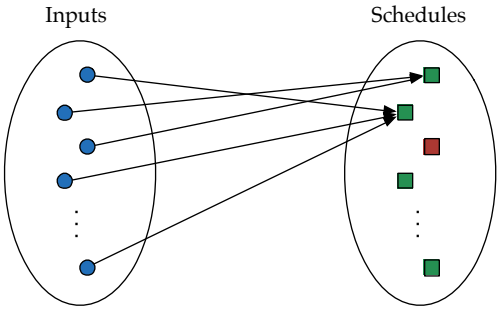


Figure 6.3: Deterministic Stable Multithreading

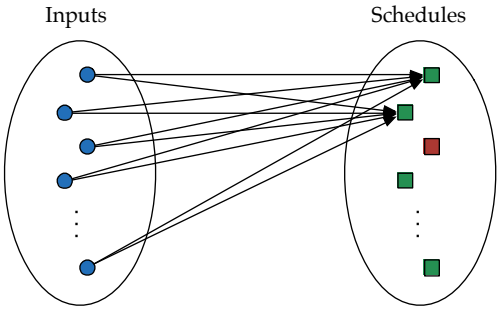


Figure 6.4: Nondeterministic Stable Multithreading

is that different inputs can be mapped to different schedules, disguising scheduling nondeterminism as input nondeterminism. To address this problem, DMT can be complemented with *stable multithreading* (SMT) [7, 17, 18, 40, 41] that reduces the set of schedules for all inputs by mapping similar inputs to the same schedule as depicted in Figure 6.3. This approach considerably reduces the number of allowed schedules and the effort needed to test a program.

While combining DMT with SMT makes testing of multithreaded programs easier, mapping many inputs to the same schedule can lead to artificial serialization of concurrent computation, resulting in performance loss [17, 40, 41]. To address this problem, recent work [39, 161, 162] advocates the use of SMT that may be nondeterministic but keeps the amount of nondeterminism small, resulting in a many-to-few mapping as depicted in Figure 6.4. Nondeterministic SMT balances performance and testability by allowing the runtime to choose an efficient schedule from a set of schedules based on the current timing, while keeping the set of allowed schedules small in order to reduce the effort needed to check all schedules.

6.1.2 Parrot

Parrot [39] is an implementation of nondeterministic SMT for POSIX-compliant operating systems. Similar to dBug, it uses runtime interposition to intercept and order

```
void soft_barrier_init(int size, void *key, int timeout);
void soft_barrier_wait(void *key);
void pcs_enter();
void pcs_exit();
```

Figure 6.5: Performance Hints API

invocations of the POSIX interface. Unlike dBug, which serializes the program transitions delimited by POSIX interface invocations, Parrot only serializes POSIX interface invocations themselves, overlapping computation that happens in between. Further, Parrot only orders `pthreads` synchronizations [120].

To this end, Parrot maintains a *run queue* that identifies program threads that can make progress and a *wait queue* that identifies program threads that are blocked. By default, Parrot uses deterministic SMT, scheduling `pthreads` synchronizations invoked by threads on the run queue in a round-robin fashion. In addition to that, Parrot exports a simple API, referred to as *performance hints*, through which programs can override the default scheduling policy of Parrot, possibly resulting in nondeterministic SMT. Parrot supports two types of hints – *soft barriers* and *performance-critical sections* – and their API is depicted in Figure 6.5.

A soft barrier can be used to express co-scheduling intent [112]. In that sense, it acts as a traditional barrier. The difference between a soft barrier and a traditional barrier is that a soft barrier can time out. The reason for allowing a soft barrier to time out is to address situations where the programmer cannot accurately predict the number of threads that would join the soft barrier. The timeout argument specifies the maximum number of Parrot's scheduling decisions a soft barrier waits before deterministically releasing all waiting threads. The ability to time out makes a soft barrier more robust to developer mistakes than a traditional barrier because schedulers cannot ignore a traditional barrier [44].

A performance-critical section can be used to identify a code region that the scheduler should execute as quickly as possible. To this end, Parrot schedules `pthreads` synchronizations inside of a performance-critical section as soon as possible, overriding the default round-robin scheduling policy. Thus, unlike a soft barrier, a performance-critical section can result in nondeterministic scheduling. However, in contrast to traditional nondeterministic multithreading that uses nondeterministic scheduling by default, performance-critical sections allow developers to explicitly identify code regions that benefit from nondeterministic scheduling for improved performance.

Figure 6.6 illustrates the mechanism Parrot uses for controlling scheduling of thread synchronizations. Similar to dBug's interposition layer, Parrot intercepts invocations of the `pthreads` synchronizations and schedules them in a round-robin order. This order can be overridden by performance hints. Soft barriers steer Parrot's scheduler towards a more efficient schedule without introducing nondeterminism, while performance-

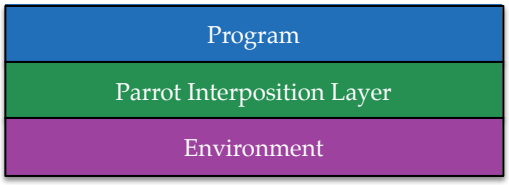


Figure 6.6: Controlling Thread Scheduling with Parrot Interposition Layer

```
1  #include <pthread.h>
2  #include <stdio.h>
3
4  pthread_mutex_t mutex;
5  int x = 0;
6
7  void *foo(void *args) {
8    pthread_mutex_lock(&mutex);
9    x++;
10   pthread_mutex_unlock(&mutex);
11   return NULL;
12 }
13
14 int main(int argc, char **argv) {
15   pthread_t tid;
16   pthread_mutex_create(&mutex, NULL);
17   pthread_create(&tid, NULL, foo, NULL);
18   pthread_mutex_lock(&mutex);
19   x++;
20   pthread_mutex_unlock(&mutex);
21   pthread_join(tid, NULL);
22   pthread_mutex_destroy(&mutex);
23   assert(x == 2);
24   return 0;
25 }
```

Figure 6.7: Concurrent `pthreads` Synchronizations - Source Code

critical sections temporarily exclude some threads from Parrot's scheduling, introducing nondeterminism by allowing multiple schedules.

Example 9. To illustrate Parrot's operation, let us consider the example program depicted in Figure 6.7. In this example, a thread spawns a child thread and both of the threads then use a mutex to protect their concurrent updates to a global variable. When executed with Parrot, Parrot intercepts invocations of POSIX interface functions, internally maintaining a run queue and a wait queue and controlling the order in which `pthreads` synchronizations happen. Figure 6.8 depicts the sequence of abstract program states explored by Parrot. Each state identifies the value of the global variable `x` and the control flow position of both threads. After the child thread is created, the run

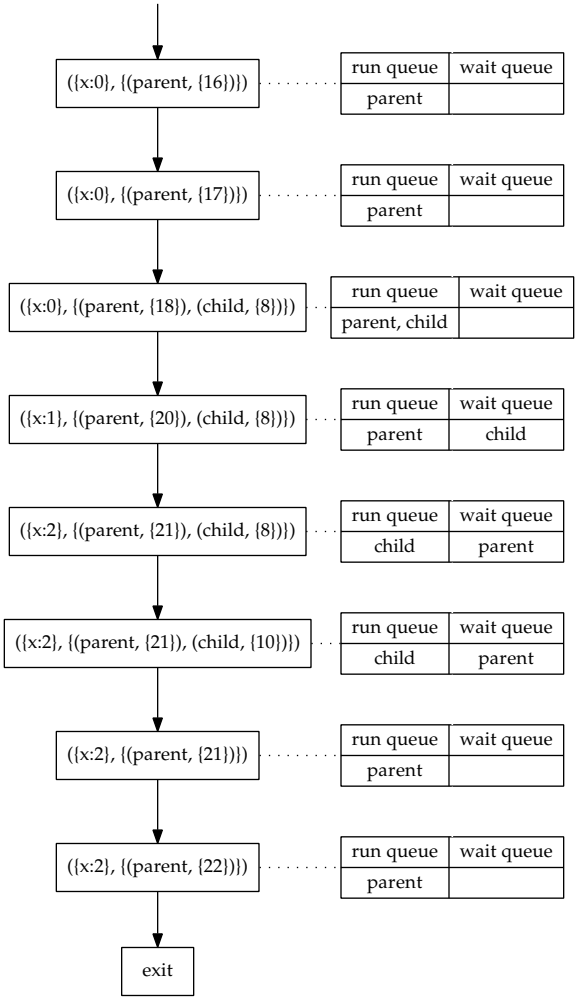


Figure 6.8: Concurrent `pthreads` Synchronizations - Parrot Execution

queue contains both the parent thread and the child thread. Traditionally, the OS scheduler could schedule the `pthreads` synchronization of either of the two threads next. In contrast to that, Parrot suspends execution of both threads and uses a deterministic round-robin policy to determine that the parent thread should be scheduled first. Note that Parrot understand the semantics of `pthreads` synchronizations and updates the run queue and wait queue accordingly. For example, when the parent thread acquires the mutex, the child thread is moved from the run queue to the wait queue.

Note that Parrot assumes that no thread whose synchronizations are ordered by Parrot's scheduler can block outside of Parrot. Obviously, this assumption does not hold for real-world programs that use inter-process synchronizations that may block, such as `poll()`, `select()`, or `epoll_wait()`. To account for inter-process synchronizations, Parrot treats their invocations as if they were included in a performance-critical section, effectively excluding them from Parrot's scheduling algorithm.

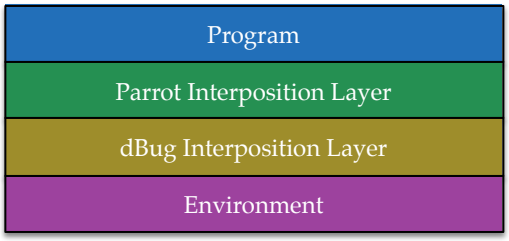


Figure 6.9: Layering of Parrot and dBug Interposition

6.2 Methods

The idea behind nondeterministic SMT is to improve testability of multithreaded programs without hurting performance. To demonstrate that this idea translates to practice, this section presents an integration of Parrot, an implementation of nondeterministic SMT, and dBug, a systematic testing tool. The restricted runtime scheduling of Parrot reduces the number of schedules multithreaded programs can use. However, performance-critical hints or invocations of inter-process synchronizations can still act as a source of scheduling nondeterminism. Consequently, dBug is used to systematically enumerate the reduced set of schedules allowed by Parrot.

6.2.1 Interposition Layering

Parrot and dBug are both implemented using a similar concept: use runtime interposition to intercept and order invocations of the POSIX interface. The key insight to their integration is that for the purpose of systematic testing, Parrot can be viewed as part of the multithreaded program. This insight is captured by the layered design of the integration depicted in Figure 6.9, which combines the interposition of Parrot and dBug depicted in Figures 6.6 and 3.7 respectively.

A naive layering of Parrot and dBug may, however, introduce artificial deadlocks. To understand why this might happen, let us consider the execution depicted in Figure 6.8 but this time with dBug in the picture. When Parrot reaches the state in which both the parent thread and the child thread are on the run queue, it uses its deterministic round-robin policy to schedule the synchronization of the parent thread first, moving the child thread to the wait queue. This synchronization is then intercepted by dBug but, following its scheduling policy, dBug will not make a scheduling decision until it intercepts an event of the child thread. At this point, the parent thread is blocked in dBug waiting for Parrot to make progress and the child thread is blocked in Parrot waiting for dBug to make progress. The crux of this problem is that dBug does not know what threads are blocked in Parrot.

```
void thread_waiting();
void thread_running(pthread_t tid);
```

Figure 6.10: Thread Status API

6.2.2 Scheduling Coordination

To overcome this problem, dBug was extended with an interface that Parrot (or any other program for that matter) can use to inform dBug that 1) the execution of a running thread has been suspended, or 2) the execution of a suspended thread has been resumed. This interface is depicted in Figure 6.10. The `thread_waiting()` function can be used to inform dBug that the calling thread is going to be suspended, while the `thread_running()` function can be used to inform dBug that the thread identified by the `tid` argument is going to be resumed. Further, the scheduling algorithm of the dBug arbiter was modified to treat threads that have invoked the `thread_waiting()` function as quiesced until they are referenced by an invocation of the `thread_running()` function. The implementation of the new interface and the scheduling algorithm modification required less than 50 lines of code. Finally, Parrot was annotated with calls to the new dBug interface to match the movement of threads between Parrot's run queue and wait queue, requiring less than 20 lines of code. With the new interface and annotations in place, the layering interposition of Parrot and dBug works seamlessly.

6.3 Evaluation

To quantify the benefits of integrating restricted runtime scheduling with systematic testing, our integration of Parrot and dBug was evaluated on a diverse set of 108 workloads [39]. This set includes 55 real-world workloads based on the following programs:

- `bdb`, a widely used database library [108]
- `openldap`, a server implementing the Lightweight Directory Access Protocol [110]
- `redis`, a fast key-value data store server [124]
- `mplayer`, a popular media encoder, decoder, and player [104]
- `pbzip2`, a parallel compression utility [59]
- `pfscan`, a parallel `grep`-like utility [113]
- `aget`, a parallel file download utility [1]
- 33 parallel C++ STL algorithm implementations [53, 142]
- 14 parallel image processing utilities in the ImageMagick software suite [76]

Further, this set also includes 53 workloads from four widely used benchmark suites:

- 15 workloads from PARSEC [21]
- 14 workloads from Phoenix [168]
- 14 workloads from SPLASH-2x [20, 141]
- 10 workloads from NPB [8]

No workloads from these benchmark suites were excluded from our consideration to avoid biasing our results. The benchmark suites cover a number of different programming languages, including C, C++, and Fortran, and a plethora of different parallel programming models and idioms such as `pthreads` [120], OpenMP [111], data partition, fork-join, pipeline, map-reduce, and work-pile.

6.3.1 Experimental Setup

The evaluation used a 2.80 GHz dual-socket hex-core Intel Xeon with 24 cores and 64 GB memory running Ubuntu 12.04.

The `bdb` workload is based on a popular benchmark `bench3n` [4], which does fine-grained, highly concurrent transactions. The `openldap` and `redis` workloads are both based on benchmarks included in their distribution. The `mplayer` workload is based on its utility `mencoder`, transcoding a 255 MB video from MP4 to AVI. For `pbzip2`, one workload compresses a 145 MB binary file and another workload decompresses the compressed version of this file. The `pfscan` workload searches for the keyword `return` in 16K files contained in `/usr/include` on our evaluation machine. The `aget` workload downloads a 656 MB file from a local web server. All ImageMagick workloads use a 33 MB JPG file as an input. All 33 parallel STL algorithms use integer vectors with 256M elements. Workloads from the PARSEC, Phoenix, SPLASH-2x, and NPB benchmark suites used the smallest configuration of these workloads. In particular, all workloads in our evaluation used two to four threads.

All multiprocess client-server workloads (`openldap`, `redis`, and `aget`) use a simple binary that drives the workload. This binary first starts the server process, next it starts the client process, next it waits for the client process to terminate, and finally it kills the server process, which concludes the workload.

Note that using relatively small inputs and a small number of threads produces a lower bound on the number of schedules of workloads with more threads and larger inputs. Our measurements (§ 6.3.2) show that even for these relatively small workloads, the extent of state space explosion is considerable and the benefits of combining restricted runtime scheduling with systematic testing far exceed the benefits of any other practical state space reduction technique [51, 67].

All programs were compiled using `gcc -O2`. To support OpenMP programs such as parallel STL algorithms and NPB, the GNU `libgomp` implementation of OpenMP was used. Five programs use ad-hoc synchronization [159], and `sched_yield` was added to their busy-wait loops to make these programs work with Parrot and dBug.

Finally, the evaluation used the performance hints added by authors of Parrot [39]. Of all 108 programs, 18 have reasonable overhead with the default Parrot schedule, requiring no hints. 81 programs need a total of 87 lines of soft barrier hints: 43 need only 4 lines of generic soft barrier hints in `libgomp`, and 38 need program-specific soft barrier hints. These programs enjoy both determinism and reasonable performance. Only 9 programs need a total of 22 lines of performance-critical hints, introducing isolated scheduling nondeterminism to achieve good performance.

6.3.2 Results

The results of our evaluation are divided into two groups. The first group contains 96 workloads for which Parrot uses only one schedule. These are all multithreaded workloads that either use no hints or soft barrier hints only. The second group contains 12 workloads for which Parrot uses multiple schedules. These are all multiprocess workloads (`aget`, `openldap`, and `redis`) and all multithreaded workloads that use performance-critical hints.

Single Schedule Workloads

The first group of workloads does not require dBug to be used for testing as the only schedule allowed by Parrot can be exercised by simply running the workload in Parrot. Nevertheless, to estimate the state space reduction realized by Parrot, all 96 workloads were tested with dBug without using Parrot. In particular, dBug repeatedly executed each workload until it enumerated all schedules or the time out of 24 hours was reached in which case dBug's state space estimation based on the lazy strategy, the weighted-backtrack estimator, and the empty fit (cf. Chapter 4) was used to estimate the time needed to explore all possible schedules.

Figure 6.11 depicts the results of this experiment. The vertical axis plots the runtime of systematic enumeration of all schedules of the workloads identified by the horizontal axis. The workloads whose schedules could be exhaustively enumerated in 24 hours are identified by green color, while the workloads whose schedules could not be exhaustively enumerated in 24 hours are identified by red color. The dotted horizontal line identifies one machine day and the solid horizontal line identifies the one machine year of the sum of Top500 supercomputers [102]. The graph shows results for 95 out of the 96 workloads for which Parrot uses only one schedule. Notably, the graph is missing a result for the SPLASH-2x `volrend` workload. This workload causes dBug to run out of memory trying to represent a single branch of the execution tree with more than 57M nodes.

The results for the first group of workloads offer two insights. First, using Parrot increases the number of workloads whose schedules can be exhaustively enumerated from 46 to 96. Second, the state space reduction realized by combining dBug with Parrot ranges up to $10^{100,000}$ and can be expected to be even more dramatic for programs with

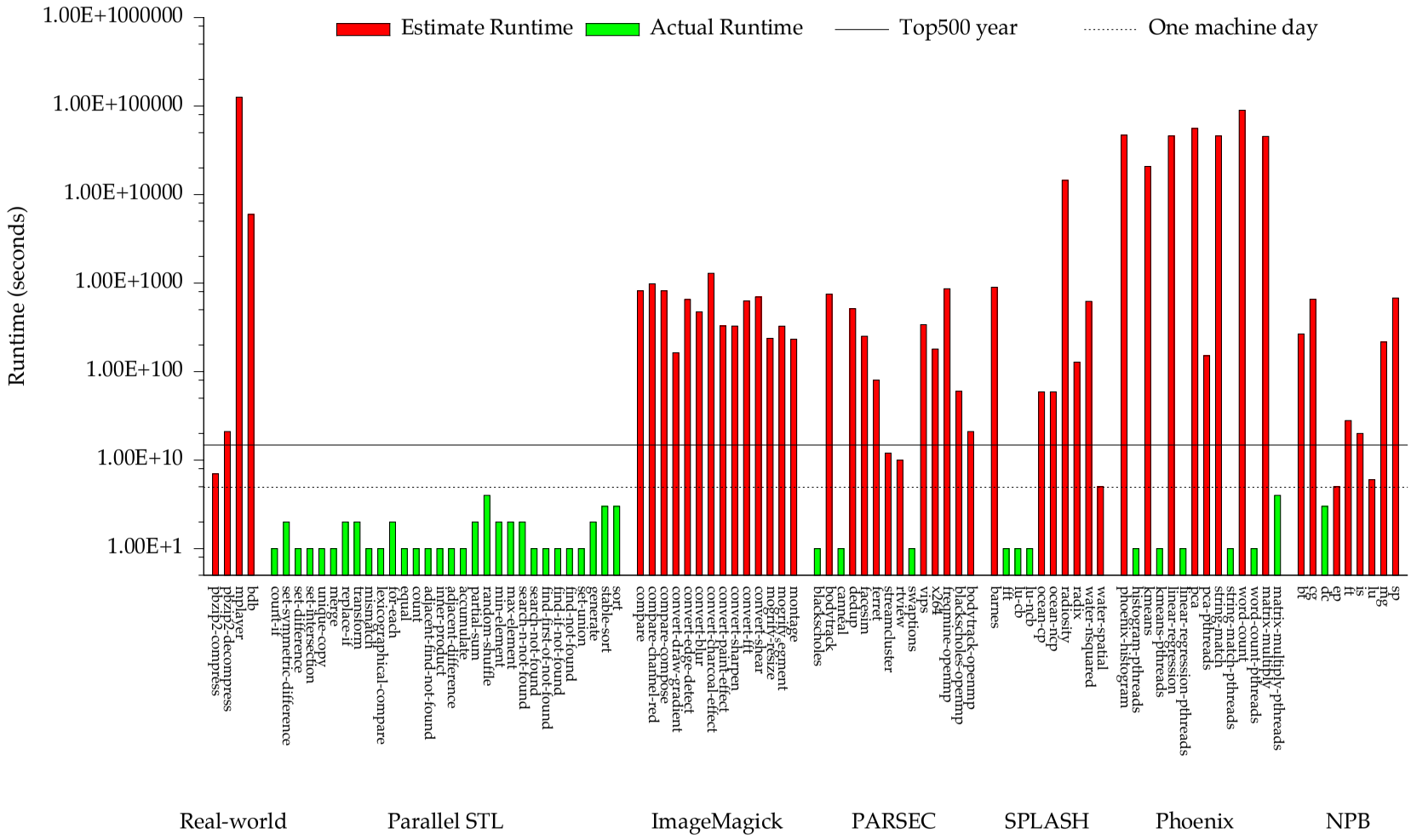


Figure 6.11: Actual and Estimated Runtime of Systematic Tests

Workload	dBug	dBug + Parrot	Time
`partition`	1.37×10^{7}	8,194	307s
`partial_sort`	1.37×10^{7}	8,194	307s
`nth_element`	1.35×10^{7}	8,224	309s
`pfscan`	2.43×10^{2117}	32,268	1,201s
`redis`	1.26×10^{8}	9.11×10^{7}	-
`aget`	2.05×10^{17}	5.11×10^{10}	-
`fmm`	1.25×10^{78}	2.14×10^{54}	-
`cholesky`	1.81×10^{371}	5.99×10^{152}	-
`fluidanimate`	2.72×10^{218}	2.64×10^{218}	-
`openldap`	2.40×10^{2795}	5.70×10^{1048}	-
`raytrace`	1.08×10^{13863}	3.68×10^{13755}	-

Table 6.1: Estimated Runtime of Systematic Tests

larger inputs and more threads. This reduction greatly surpasses all previous methods for state space reduction of systematic testing tools [51, 67].

Multiple Schedules Workloads

The second group of workloads requires dBug to be used for testing as the workloads from this group can exercise any of the multiple schedules allowed by Parrot. Each workload from this group was tested using dBug both with and without Parrot to quantify and contrast the extent of state space explosion. In particular, dBug repeated execution of each workload until it enumerated all of its schedules or the time out of 24 hours was reached in which case dBug's state space estimation based on the lazy strategy, the weighted-backtrack estimator, and the empty fit (cf. Chapter 4) was used to estimate the time needed to explore all possible schedules.

Table 6.1 details the individual results for all 12 workloads from the second group except for the NPB `ua` for which dBug runs out of memory trying to represent a single branch of the execution tree with more than 127M nodes. The first column identifies the workload, the second column identifies the estimated number of schedules when the workload is run without Parrot, the third column identifies the estimated number of schedules when the workload is run with Parrot, and the fourth column reports the time it took dBug to exhaustively enumerate all schedules allowed by Parrot, where the value '-' identifies that dBug timed out after 24 hours.

The results for the second group of workloads show that using Parrot helps dBug exhaustively enumerate schedules for 4 of the 12 workloads. For another 5 of the remaining 8 workloads, using Parrot reduces the estimated number of schedules by many orders of magnitude but the resulting state space is still too large to be fully explored by one machine in 24 hours. In the case of multithreaded workloads, the large number of schedules stems from nondeterminism in scheduling intra-process synchronizations within performance-critical sections. In the case of multiprocess workloads, the large number of schedules stems from nondeterminism in scheduling inter-process communication.

All in all, combining restricted runtime scheduling with systematic testing greatly reduces the number of schedules systematic testing needs to check, increasing the number of workloads dBug can exhaustively check from 46 to 100 (out of 108).

6.4 Related Work

DMT and SMT Systems

Unlike Parrot, several prior systems are not backward-compatible because they require new hardware [42], new language [23], or new programming model and OS [7]. Among backward-compatible systems, some DMT systems, including Kendo [109], CoreDet [15], and CoreDet-related systems [16, 75], improve performance by balancing each thread's load with low-level instruction counts. As mentioned in the introduction of this chapter, DMT systems benefit behavior replication but do not solve the testing problem.

Prior deterministic SMT systems reduce the testing effort in addition to providing determinism. However, they lack a mechanism through which developers could tune their performance and require either heavyweight techniques or unrealistic assumptions. Grace [18] requires fork-join parallelism. Tern [41] and Peregrine [40] record and reuse schedules. Execution recording can slow down the execution by an order of magnitude, and computing schedules from recorded executions relies on sophisticated source code analysis. DThreads [17] stabilizes schedules by ignoring load imbalance among threads, so it is prone to the serialization problem explained in Section 6.1.

State Space Reduction

Combining dBug with Parrot greatly reduces the number of schedules that need to be checked, as such this approach bears similarity to *state space reduction techniques* [51, 61, 67] which soundly reduce the state space to mitigate the state space explosion problem of model checking. Partial order reduction [51, 61] has been the main reduction technique for systematic testing tools [137, 160], including dBug and is discussed in detail in Chapter 2 of this thesis. Recently, researchers have proposed dynamic interface reduction [67] that checks loosely coupled components separately, avoiding expensive global exploration of all components. However, this technique has yet to be shown to work well for tightly coupled components such as threads frequently communicating via synchronizations and shared memory.

Using restricted runtime scheduling offers three advantages over previous reduction techniques: (1) it is conceptually simpler because it does not rely on behavioral equivalence to reduce the state space; (2) in the absence of performance-critical sections and inter-process synchronizations, it remains effective as the checked system scales; and (3) it works orthogonal to existing state space reduction techniques [51, 61, 67] used in systematic testing. Thus, it can be combined with existing reduction techniques to reduce the state space caused by nondeterministic SMT.

Concurrency

Automatic mutual exclusion (AME) assumes all shared memory is implicitly protected and allows advanced developers the flexibility to remove protection. It thus shares a similar high-level philosophy with restricted runtime scheduling. The differences is that, unlike restricted runtime scheduling, AME has only been implemented in simulation.

Restricted runtime scheduling is orthogonal to much prior work on concurrency error detection [46, 96, 128, 169, 170], diagnosis [114, 115, 130], and correction [78, 79, 157, 158]. By reducing the number of schedules, restricted runtime scheduling potentially benefits all of these techniques.

6.5 Conclusions

In conclusion, this chapter presented an integration of restricted runtime scheduling with systematic testing. To this end, dBug was integrated with Parrot, an implementation of restricted runtime scheduling for POSIX-compliant operating systems. Parrot offers a new contract to developers: by default, it schedules synchronizations using round-robin, greatly reducing the number of possible schedules; when the default schedules are slow, it allows developers to use performance hints to improve performance by allowing additional schedules to be used. This contract benefits testing that only needs to focus on schedules allowed by Parrot. This benefit has been demonstrated by using dBug to thoroughly check Parrot's schedules. Results on a diverse set of 108 programs show using Parrot reduces the testing effort required of dBug by many orders of magnitude.

Chapter 7

Abstraction Reduction

Abstraction is an age-old technique used in formal verification [38] to combat state space explosion. This chapter demonstrates that lifting the default abstraction of dBug to match higher-level coordination interfaces can be used to reduce the number of schedules systematic testing needs to check.

The reduction is realized by focusing only on the ordering of the higher-level coordination primitives instead of the ordering of the lower-level POSIX interface primitives that implement the higher-level coordination primitives. In other words, abstraction reduction focuses systematic testing on checking whether a concurrent program uses a higher-level coordination interface correctly, assuming or independently testing that the implementation of the interface matches its specification.

To evaluate the benefits of abstraction reduction, dBug was extended with support for intercepting and modeling events of the message-passing interface (MPI) [103], an interface widely used in the high-performance computing community. Programs from the NAS Parallel Benchmarks (NPB) suite [8] were then used to quantify and contrast the number of schedules dBug needs to check with and without abstraction reduction and how varying the coordination interface used for implementing program functionality between OpenMP [111] and MPI affects its testability.

The rest of this chapter is organized as follows. Section 7.1 provides the background necessary for understanding abstraction reduction. Section 7.2 describes the design and implementation of systematic testing of MPI through dBug. Section 7.3 evaluates our implementation, contrasting the number of schedules of both OpenMP and MPI implementations of programs from the NPB benchmark suite [8]. Section 7.4 discusses related work and Section 7.5 draws conclusions.

7.1 Background

The main advantage of using POSIX interface [119] as the default abstraction for modeling executions of concurrent programs is that it enables systematic testing of a wide range of programs across different programming languages.

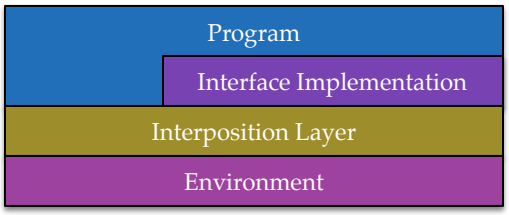


Figure 7.1: Interposition based on Default Abstraction

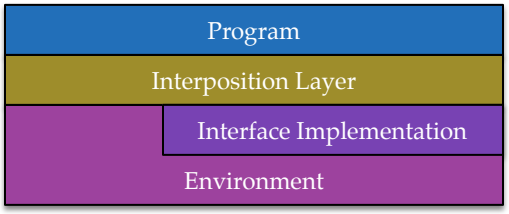


Figure 7.2: Interposition based on Custom Abstraction

The main drawback of the default abstraction is that if a program uses a higher-level interface, such as Apache Thrift [5], Java [6], MPI [103], or Python [149], that is itself implemented using the POSIX interface, then systematic testing based on the default abstraction will control and order scheduling nondeterminism in both the program and the implementation of the higher-level interface, a scenario illustrated in Figure 7.1. For higher-level interfaces with complex implementations, even simple programs can experience considerable state space explosion. For example, using the `mpich` implementation of the MPI specification, dBug equipped with dynamic partial order reduction estimated the number of interleavings of concurrent POSIX interface invocations for a trivial MPI program, which uses two processes and contains no logic besides MPI initialization and finalization of both processes, to be on the order of 10^{107}. This is more than the estimated number of the atoms in the universe.

To avoid the state space explosion stemming from the scheduling nondeterminism in the implementation of a higher-level interface, one can assume that the implementation of the higher-level interface is correct. Equipped with this assumption, systematic testing tools can intercept and model invocations of the higher-level interface primitives, abstracting away the details of the interface implementation, a scenario is illustrated in Figure 7.2. The benefit of using a custom abstraction lies in reducing the number of schedules systematic testing needs to consider, focusing systematic testing on the interactions of a program with the higher-level interface as opposed to interactions an implementation of the higher-level interface with the POSIX interface. The drawback of using a custom abstraction is the effort required to provide support for systematic testing of the higher-level interface [147, 155].

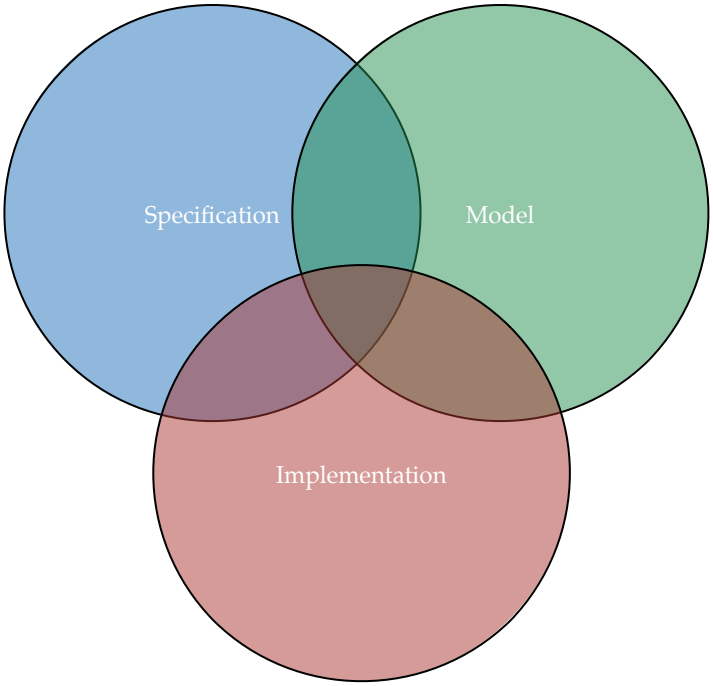


Figure 7.3: Different Representations of Interface Behaviors

Notably, modeling a specification of an interface instead of testing its implementation is prone to human error. As Figure 7.3 suggests, a specification, an implementation, and a model each define a set of behaviors that the interface can exhibit. Ideally, these three definitions match.

In practice, however, for reasons such as performance or backward compatibility, implementations may choose to exclude behaviors included in the specification or include behaviors excluded from the specification. For example, the specification of the `pthread_cond_signal()` function mandates that if any threads are blocked on the specified condition variable, the function unblocks at least one of the threads. To achieve fairness, an implementation of the conditional variables interface may choose to wake up the thread that has been waiting the longest. However, for some programs, this implementation might allow only a subset of the behaviors allowed by the specification.

Systematic testing based on abstraction creates its own model of an interface, defining a set of behaviors that an interface can exhibit, and uses this model to represent concrete program executions that use an implementation of the interface. If the model is inaccurate, the inaccuracies can manifest as behaviors allowed by the model but not by the specification or vice versa. For example, the specification of the `pthread_barrier_wait()` function mandates that when the required number of threads have called the function, a non-zero constant will be returned to one of the threads and zero will be returned to all remaining threads. If a model does not capture this nondeterminism, its application may fail to detect program errors.

7.2 Methods

To explore the idea of abstraction reduction, dBug was extended with support for the message-passing interface (MPI) [103]. In particular, the dBug interposition layer was modified to intercept invocations of MPI primitives and the dBug arbiter was modified to model and schedule execution of these invocations. The remainder of this section first describes the design considerations related to implementing support for systematic testing of MPI in dBug (§7.2.1) and then highlights notable aspects of our implementation of this design (§7.2.2).

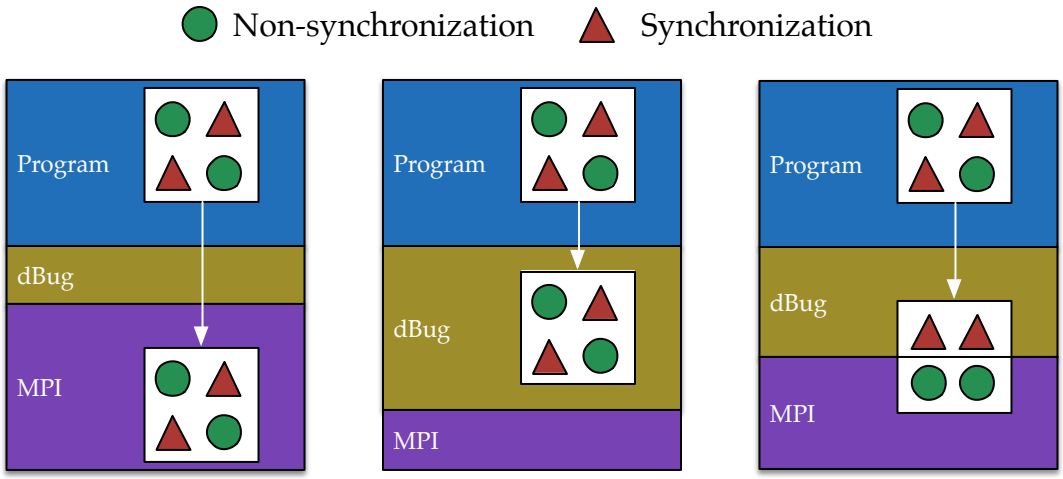
7.2.1 Design

An important design decision to consider when extending dBug with support for MPI, or any other interface, is whether to use dBug only as a scheduler, *delegating* interposed events to an existing implementation of the interface, or whether to use dBug to implement the interface program logic, *replacing* an existing implementation. The advantage of delegation is that it avoids 1) the effort needed to implement the interface program logic inside of dBug and 2) the risk of inaccurately mapping the specification of the interface to an implementation. The disadvantage of delegation is that it may limit the control dBug has over scheduling of program threads and the outcome of the interposed events. The delegation and replacement approaches to implementing MPI program logic are depicted in Figure 7.4 (a) and Figure 7.4 (b) respectively.

Our experience with implementing support for POSIX interface in dBug suggests that delegation is appropriate for program logic that does affect thread scheduling, such as I/O operations, but program logic that affects thread scheduling should be implemented in dBug. For instance, dBug implements the program logic for synchronization events such as `waitpid()` or `epoll_wait()`, while execution of non-synchronization events, such as `fork()` or `send()`, are delegated to the `libc` library [92].

A challenge for dBug's support for MPI is that, as a programming convenience, many MPI primitives mix synchronization and non-synchronization program logic. On one hand, these primitives should be implemented by dBug to retain control over scheduling of program threads. On the other hand, having to implement non-synchronization logic, such as data movement and transformation for the rich set of MPI data types and operations, in order to systematically test concurrency seems counter-intuitive and unnecessarily elaborate.

To resolve this problem, our design chooses to layer dBug on top of an existing implementation of the MPI specification. However, instead of simply delegating complex MPI primitives that mix synchronization and non-synchronization program logic to the existing implementation, our design decomposes these primitives into a set of synchronization and non-synchronization operations. The synchronization operations are then implemented in dBug, while the non-synchronization operations are delegated to the existing implementation. The decomposition approach to implementing MPI program logic is depicted in Figure 7.4 (c).



(a) Delegation (b) Replacement (c) Decomposition

Figure 7.4: Different Approaches to Implementing MPI Program Logic

7.2.2 Implementation

To enable systematic testing of MPI, the dBug interposition layer intercepts invocations of MPI primitives and the dBug arbiter models and schedules execution of these invocations. In particular, the global state maintained by dBug contains objects representing MPI message queues, collective communication operations, and MPI process ranks. The remainder of this section describes notable aspects of our implementation. A detailed list of MPI primitives that dBug interposes on can be found in Appendix B.

Deterministic Process Naming

To facilitate communication between different processes of an MPI program, the MPI specification defines *process ranks* that are used to identify processes in the context of a *communicator*. Communicators provide scope for group communication and prevent communication conflicts. The MPI specification defines a default global communicator that can used to identify and communicate with any process spawned by the same MPI program launch.

When an MPI program starts, each of its processes calls the `MPI_Init()` function that acts as a barrier and, among other things, assigns the calling process a global communicator process rank. This process rank is then typically used to identify what program logic a particular MPI process should execute (cf. Figure 3.3).

To enable deterministic replay, dBug assigns deterministic thread identifiers to threads when they are created and uses these identifiers to identify threads across different executions. Since the program logic of MPI processes is generally determined by their global process rank, it is important for the mapping between dBug thread identifiers and MPI process ranks to stay the same across different executions.

However, the global process ranks are assigned by the `MPI_Init()` function which cannot be decomposed into simpler MPI primitives and our design requires its invocations are delegated to an existing implementation of MPI. Fortunately, the global communicator the `MPI_Init()` function creates can be overridden. In particular, after the `MPI_Init()` function assigns the original global process ranks, dBug creates a separate global communicator, with ranks ordered according to its deterministic thread identifiers, and uses this communicator in place of the default one.

Nondeterministic Message Matching

One of the sources of nondeterminism in the MPI specification is the point-to-point communication between MPI processes. To receive a message from a particular sender, an MPI process may call the `MPI_Recv()` function, identifying the sender from which to receive a message. However, this function also allows a *wildcard* in place of a sender identifier, expressing the intent to receive a message from any sender. Although the MPI specification mandates that messages sent from the same sender are received in the order they were sent, messages sent from different senders can be received in any order.

To account for this nondeterminism, dBug keeps track of the content of all MPI message queues and the *test* function component (cf. Section 3.2.3) of the `MPI_Recv()` function model computes all possible matches of a wildcard invocation. When the dBug arbiter makes a scheduling decision involving a wildcard invocation, the intended match is communicated back to the dBug interposition layer. The dBug interposition layer then delegates invocation of the `MPI_Recv()` function to an existing implementation of MPI, replacing the wildcard with identifier of the match.

Non-blocking Communication Primitives

In addition to traditional blocking communication, the MPI specification facilitates latency hiding by offering non-blocking versions of all communication primitives. For example, the `MPI_Irecv()` function is a non-blocking version of the `MPI_Recv()` function. Instead of blocking until a message is received, it returns immediately, providing a handle that can be used to check for completion.

To retain control over scheduling of MPI operations and matching of MPI messages, dBug decomposes the program logic of non-blocking communication primitives. When the dBug interposition layer intercepts an invocation of a non-blocking communication primitive, it internally spawns a thread that waits to be scheduled by the dBug arbiter and then invokes a blocking version of the communication primitive. Once the child thread is spawned, the parent thread returns a handle that can be used to check whether the internally spawned thread has terminated.

The MPI specification offers both blocking and non-blocking primitives for checking whether an invocation of a non-blocking communication primitive has completed. The `MPI_Wait()`, `MPI_Waitall()`, `MPI_Waitany()`, and `MPI_Waitsome()` functions are blocking and the `MPI_Test()`, `MPI_Testall()`, `MPI_Testany()`, and

`MPI_Testsome()` functions are their non-blocking counterparts. In general, the functions that check for completion can return one of several values and the corresponding *test* function components of the dBug arbiter model this nondeterminism and enumerate all possibilities.

Collective Communication Primitives

Besides point-to-point communication primitives, the MPI standard also specifies a number of convenience primitives for collective communication. These primitives operate over a collection of processes and represent typical communication patterns. Based on the communication pattern, these primitives can be divided into four groups:

- *all-to-all* primitives move data between every pair of processes involved in the communication
- *all-to-one* primitives move data from all processes involved in the communication to one distinct process
- *one-to-all* primitives move data from one distinct process to all processes involved in the communication
- *scan* primitives move data between processes involved in the communication in the increasing order of their process rank

As far as synchronization is concerned, the all-to-all primitives are effectively a barrier and the scan primitives are a serialization. In other words, scheduling of these primitives is deterministic. In contrast to that, the all-to-one and one-to-all primitives allow a number a scheduling scenarios. For all-to-one primitives the only requirement is that the process receiving data returns after all other processes invoke the primitive, while for one-to-all primitives the only requirement is that the process sending data invokes the primitive before all other processes return. Consequently, all-to-one and one-to-all primitives involving n processes can execute in $(n-1)!$ ways.

However, simply delegating the all-to-one and one-to-all primitives to an existing implementation of MPI does not work. In general, an implementation of MPI may choose to introduce additional synchronization, which prevents simple delegation from begin able to exercise all possible scenarios. To avoid this problem, the dBug interposition layer decomposes these primitives to a collection of point-to-point communication primitives. The point-to-point communication primitives are delegated to an existing implementation of MPI, while dBug implements the necessary synchronization.

7.3 Evaluation

To evaluate the benefits of using a custom abstraction of a higher-level interface, our implementation of support for MPI in dBug was evaluated using 8 benchmarks from the NAS Parallel Benchmarks (NPB) [8] suite.

Our benchmark selection was driven by the fact that NPB contains both MPI and OpenMP [111] implementations. OpenMP is an API for shared-memory parallel programming and its specification defines a collection of compiler directives that can be used for parallel execution of otherwise sequential code. Typically, OpenMP implementations translate these compiler directives to `pthreads` invocations, which enables systematic testing of OpenMP programs in dBug. Our choice of benchmarks thus allowed us to contrast the results for custom abstraction of MPI programs against the default abstraction of both MPI and OpenMP programs.

The benchmarks selected for our evaluation are summarized in Table 7.1. For each benchmark, the table lists an identifier, a short description, the programming language of the benchmark, and the lines of code of the MPI and OpenMP implementations.

Identifier	Description	Language	MPI	OpenMP
BT	block tri-diagonal solver	Fortran	9,348	5,295
CG	conjugate gradient	Fortran	1,878	1,262
EP	embarrassingly parallel	Fortran	368	297
FT	discrete 3D fast Fourier transform	Fortran	2,172	1,206
IS	integer sort	C	1,150	1,058
LU	lower-upper Gauss-Seidel solver	Fortran	5,797	5,403
MG	multi-grid on a sequence of meshes	Fortran	2,641	1,533
SP	scalar penta-diagonal solver	Fortran	5,036	3,385

Table 7.1: Overview of NAS Parallel Benchmarks

7.3.1 Experimental Setup

Our evaluation was carried out using Susitna nodes from PRObE [58]. A Susitna node is configured with 64 cores (quad socket, 2.1 GHz 16-core AMD Opteron), and 128 GB of memory. Our experiments used Ubuntu 12.04, the `mpich` library version 3.0.4 as the MPI implementation, the GNU `libgomp` library version 3.1 as the OpenMP implementation, and NPB version 3.3.1.

The NPB benchmarks come with several input configurations. Our experiments used the "S" configuration, representing a small instance, for all the benchmarks. Even for small instances, the extent of state space explosion is considerable and the effort needed to check large instances stretches beyond the practical limits of existing systematic testing tools. The BT and SP benchmarks were run using four threads because their MPI implementation requires the number of threads to be a square number. All other benchmarks were run with two threads.

For each benchmark, dBug was used to systematically explore different interleavings of interposed events using a number of different configurations. The OpenMP implementation was tested using the default abstraction and the MPI implementation was tested using both the default and custom abstraction. To evaluate the interaction between abstraction reduction and dynamic partial order reduction (DPOR) [51], all

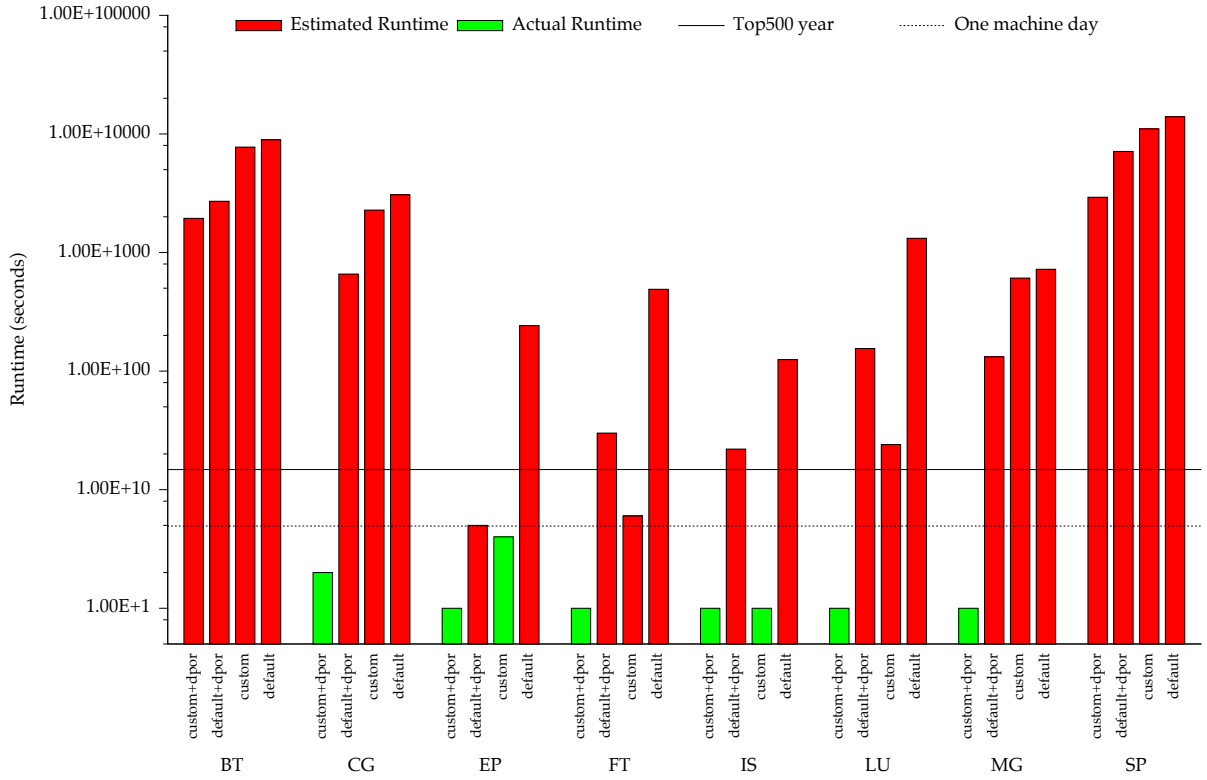


Figure 7.5: Contrasting Custom and Default Abstraction

our experiments were carried out both with DPOR enabled and disabled. Finally, each configuration was run in dBug until all interleavings were explored or a 24-hour timeout was reached, in which case dBug used the lazy strategy, the weighted-backtrack estimator, and the empty fit (cf. Chapter 4) to estimate the total runtime.

7.3.2 Results

The results of our experiments are presented as graphs in Figures 7.5 and 7.6. The horizontal axes of these graphs identify the benchmark and the reduction configuration. The log-scale vertical axes of these graphs identify the runtime. Red values represent runtime estimates for experiments that timed out after 24 hours, while green values represent actual runtimes for experiments that finished in less than 24 hours.

Figure 7.5 presents a comparison of the default and custom abstraction of MPI programs. These results demonstrate that combining custom abstraction and DPOR is able to reduce the number of interleavings of interposed events to a set that can be exhaustively enumerated for 6 out of 8 benchmarks. As it turns out, the semantics of MPI primitives used by these benchmarks allow only interleavings that are all considered equivalent by DPOR, thus reducing the state space down to a single equivalence class. This compelling result highlights the synergy between abstraction reduction and dynamic partial order reduction.

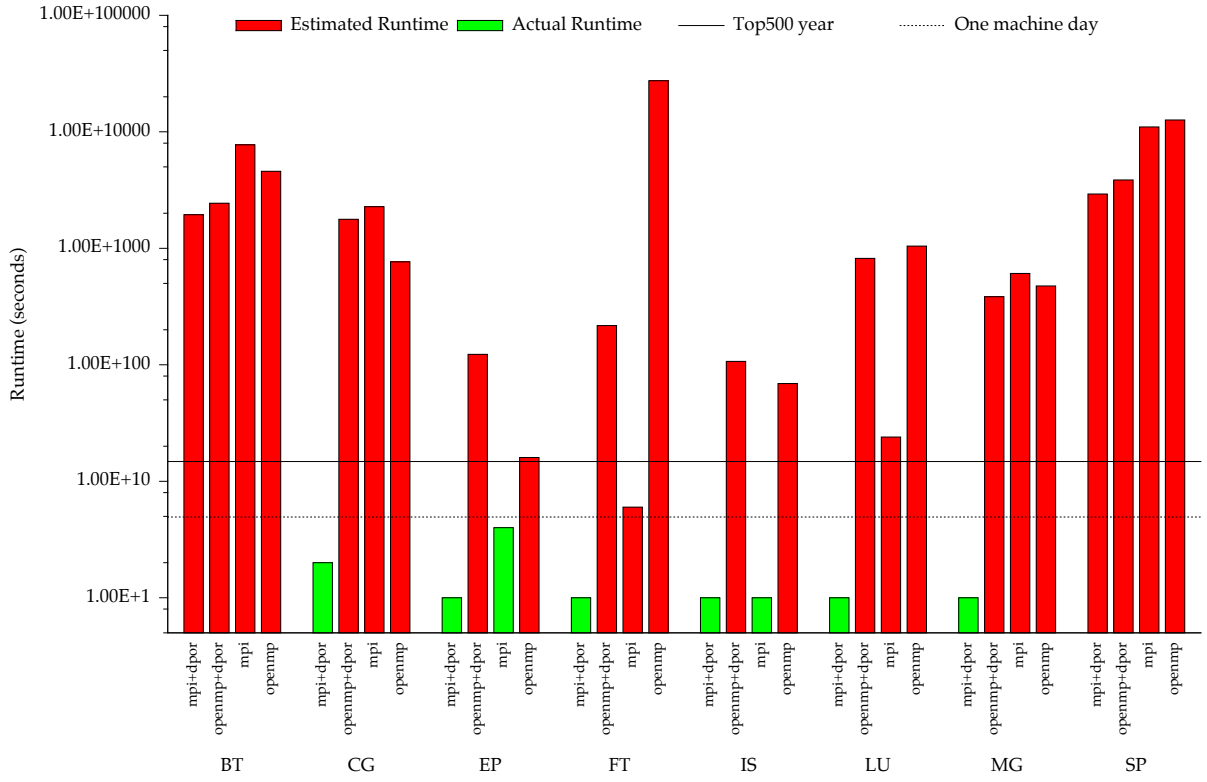


Figure 7.6: Contrasting MPI and OpenMP

Figure 7.6 presents a comparison of the custom abstraction of MPI programs and the default abstraction of OpenMP programs. This comparison reveals that using a higher-level interface to implement program logic in combination with abstraction reduction can help systematic testing tools thoroughly check the program logic. For OpenMP implementations, which `libgomp` implements through `pthreads` primitives, using dBug with DPOR fails to avoid state space explosion of possible interleavings of `pthreads` primitives for any of the benchmarks. In contrast, for MPI implementations, using dBug with abstraction reduction and DPOR avoids state space explosion of possible interleavings of MPI primitives for 6 out of 8 of the benchmarks.

7.4 Related Work

Abstraction

The seminal abstract interpretation paper by Cousot and Cousot [38] provides a formal framework for describing program abstractions and establishes conditions under which a program property can be studied using a program abstraction. Our approach to systematic testing of scheduling nondeterminism can be thought of as abstract interpretation of programs. A key aspect of the abstractions used in our work is that they model the state necessary for controlling input and scheduling nondeterminism, enabling for deterministic replay and systematic state space exploration.

The work of Kurshan [86] demonstrates how complex systems may be abstractly modeled as a collection of many small state machines. This concept is similar to our work that essentially views each intercepted interface event as a simple state machine that interacts with other events through an abstraction of the global state.

Clarke et al. study abstraction interpretation in the context of model checking [34] and demonstrate that combining abstraction with symbolic representation [26] can scale hardware verification to designs with up to 10^{1300} states. In contrast to model checking, our work checks behaviors of the actual programs and uses abstraction to group interleavings of concrete program transitions into equivalence classes.

Tools

Given the widespread popularity of MPI in the high-performance computing community, it is not surprising that a fair amount of research has focused on helping software engineers to test and debug MPI programs.

The Umpire tool [152] uses the MPI profiling layer to monitor an execution of an MPI program and check this execution for a set of correctness properties. The mechanism Umpire uses to monitor a program execution is similar to the one used by the dBug interposition layer. In contrast to our work, Umpire does not control scheduling nondeterminism in program executions or enumerate different scheduling scenarios.

The MPI-SPIN tool [133] is an MPI extension of the stateful model checker SPIN [72]. The main drawback of MPI-SPIN is that, although SPIN supports a subset of C, in general, engineers that want to check their MPI programs with MPI-SPIN need to create Promela [70] models of their MPI programs.

The TASS toolkit [134] is a suite of integrated tools to model and formally analyze parallel programs. Notably, TASS uses symbolic expressions to represent program inputs and, similar to other symbolic execution tools [27, 29], systematically enumerates non-equivalent inputs. Compared to our work, TASS only supports programs written in a subset of C, limiting the scope of its applicability for testing of MPI programs.

The ISP tool [118, 147, 148] is in spirit very similar to dBug's support for MPI. The main difference is that ISP has only been demonstrated to work on toy examples [147] and supports a small subset of all MPI primitives. Further, its authors argue that the standard happens-before relation [88] and dynamic partial order reduction [51] are inapplicable to MPI programs, furnishing ISP with custom alternatives to these time-tested techniques.

The DAMPI tool [155, 156] is a successor of the ISP tool, inheriting most of its limitations. The main innovation of DAMPI is that it offers parallel exploration capabilities and better, yet still non-standard, happens-before tracking. In contrast to ISP, the evaluation of DAMPI demonstrates feasibility of systematic testing for complex MPI programs. However, lacking the ability to change levels of abstraction or estimate length of systematic tests, DAMPI's evaluation focuses on measuring runtime overhead.

7.5 Conclusions

This chapter explored the potential of abstraction to help mitigate state space explosion stemming from scheduling nondeterminism. The key insight of this chapter is that the boundary between a program and its environment separates the program logic to be tested by systematic testing from the program logic of library services that are assumed to be independently tested. Using this insight, this chapter studied the effects of treating an implementation of a higher-level inter-process communication interface as a part of the environment as opposed to a part of the program.

In particular, this chapter explored this idea by implementing and evaluating abstraction reduction for the message-passing interface (MPI) [103], an inter-process communication interface widely used in the high-performance computing community. To this end, our systematic testing tool dBug was extended with support for intercepting and modeling events at the MPI level of abstraction. The NAS Parallel Benchmarks (NPB) suite [8] was then used to contrast the extent of state space explosion for systematic testing based on the POSIX interface abstraction to the MPI abstraction. In addition, both MPI and OpenMP implementations of NPB benchmarks were considered, enabling a comparison between different programming abstractions.

Our experimental results confirmed our intuition that using abstraction helps to mitigate state space explosion stemming from scheduling nondeterminism. What was surprising was the extent of the realized reduction; for the benchmarks considered in our evaluation the reduction ranged between 10^{106} and 10^{1773}. Our evaluation also generated evidence of synergy between abstraction reduction and dynamic partial order reduction [51]. Namely, the combination of these two reduction techniques correctly infers that for 6 out of 8 of the benchmarks considered by our evaluation, all interleavings of the interposed events can be grouped into a single equivalence class.

Chapter 8

Related Work

While the previous chapters discussed work closely related to the research undertaken by each chapter, this chapter provides a broader view, helping to place the sum of the research carried out by this thesis in the context of related work. To this end, the subject of this thesis is characterized as "*systematic* testing of *distributed and multithreaded* programs using *dynamic* analysis and *stateless* exploration to check for *safety* properties" and this characterization is used to identify dimensions for comparison to related work.

8.1 Testing / Offline Debugging / Online Debugging

An important concern of distributed and multithreaded programs, not addressed in this thesis, is how to handle faults that are detected when the program is already deployed. An important requirement for these methods is that they add only negligible overhead to the runtime of a program. *Offline* debugging meets this requirement by using a lightweight mechanism for recording a compact log of key execution events. The idea behind logging these events is to enable a deterministic replay and analysis of possibly faulty executions offline. For instance, the WiDS checker [94] achieves deterministic replay through program annotation, while the Friday tool [54] uses the `liblog` [55] infrastructure to provide for deterministic replay of unmodified distributed programs. More recently, Altekar et al. [3] have shown how to enable deterministic replay for multiprocessor architectures.

In contrast to offline debugging, *online* debugging analyzes the execution of a program while the program is running. For instance, the D3S [93] project uses binary instrumentation in Box [68] to extend legacy distributed and multithreaded programs with filters that stream execution information to a background checker that monitors correct execution of the program and can check for system-wide invariants.

In comparison to debugging, the methods and tools presented in this thesis are intended to be used for in-house software testing before the software is deployed. Notably, instead of simply monitoring an execution, systematic testing methods strive to achieve good coverage by systematically exploring different behaviors of the system.

8.2 Distributed / Multithreaded / Sequential Programs

Advances in testing and verification of *sequential* programs permeate the history of computer science. Early work by King [83] identified the value of symbolic execution as a mechanism for systematically generating test cases that explores distinct branches and paths of a computer program. This mechanism prompted the creation of symbolic evaluation tools such as Dissect as early as 1977 [74]. In recent years, researchers have exploited advancements in automated constraint solving and proven automated high-coverage testing of unmodified sequential programs to be not only feasible, but also practical [27, 29, 64, 131].

Systematic testing of *multithreaded* programs was pioneered by Godefroid in his work on VeriSoft [61, 62]. Interestingly, VeriSoft also contained experimental support for remote processes that communicated with a centralized scheduler using a local proxy process. To the best of our knowledge, this constitutes the first attempt at systematic testing of distributed and multithreaded programs. More recently, the CHESS tool [107] implemented systematic testing for unmodified multithreaded Windows-specific programs.

Though in theory both multithreaded and distributed program are concurrent systems, in practice the methods for systematic testing of each are far from identical. First, unlike a multithreaded program that runs in the context of a single process and operating system, a distributed program generally has no centralized entity that would already hold a global view of the program state. Second, multithreaded programs typically coordinate via shared state, while distributed program often coordinate via both shared state and message passing. The latter difference plays an important role in methods for systematic testing of unmodified programs, which in the case of distributed programs need to account for additional communication and coordination primitives.

Systematic testing of unmodified *distributed* programs was first investigated in detail by the MaceMC [82] tool. The limitation of MaceMC is that it only handles programs written in the Mace [81] programming language. More recently, the MoDist [160] tool extended the same concept to unmodified legacy distributed programs written for the Windows platform. The MoDist tool relies on automated instrumentation of Windows API [68] and uses this instrumentation to explore different execution orders, inject faults, and simulate timeouts.

The methods and tools presented in this thesis push the limits of previous approaches in several directions. State space estimation enables quantification of systematic testing progress and coverage, parallel state space exploration scales systematic testing to large computational clusters, and restricted runtime scheduling and abstraction reductions scale systematic testing to more complex programs. Further, dBug enables systematic testing irrespective of programming languages and instruction sets. The only prerequisite of dBug is that the underlying system is POSIX-compliant and supports runtime interposition. Once this prerequisite is met, one can use dBug to systematically test unmodified binaries of any program.

8.3 Dynamic / Static Analysis

In general, certain programming concepts such as pointer arithmetic, function pointers, or library calls, all of which are used frequently in systems code, represent obstacles for the use of static analysis tools. These obstacles are typically overcome by making assumptions about the programming language and constructs of the system, the environment in which the systems runs, or both. Unlike static analysis tools, dynamic analysis tools do not make assumptions about either the system or the environment. Instead, the dynamic analysis runs the actual system within some specific environment. The advantage of a dynamic analysis is that it observes an actual behavior of the system. The disadvantage is that its observations are made in the context of a specific environment.

Static analysis of programs for concurrency faults has been an active area of research for decades. A notable recent research effort produced the HAVOC tool [87] for automatic detection of faults in multithreaded C programs. To combat the state space explosion, the exploration algorithm of HAVOC explores possible context-switching scenarios only to a certain depth – a technique known as *context-bounding* [106]. The tool has been used to find faults in Windows device drivers. Interestingly, the experiments required a creation of a model of the Windows operating system environment.

Besides detecting faults, static analyses are also used to prevent faults by providing compile time guarantees. For example, Engler et al. [47] have demonstrated how to write system-specific checks that prevent certain types of faults from happening. Reflecting on their experience a decade later [19], the authors identified the ability to avoid false positives as the most important step towards practicality of static analysis.

The methods and tools presented in this thesis refrain from the use of static analysis in order to avoid false positives and programming language dependency.

8.4 Stateless / Stateful Search

All systematic testing tools need a mechanism for navigating the space of possible program states. While going forward can be achieved by simply executing instructions of the program, going backward is not as straightforward and one needs to store the program state. In principle, this can be achieved either by storing the state explicitly (*stateful search*), or implicitly as a sequence of nondeterministic choices leading to the state from some previous state (*stateless search*) – trading space for time. A separate concern of the stateless approach is the need to reconstruct the initial state.

Examples of methods implementing the stateful search include the C Model Checker (CMC) [105], the FiSC tool [165], or the KLEE tool [29]. These tools store a portion of the in-memory and on-disk state so that they can later reconstruct this state. In contrast to that, the stateless search was first proposed in VeriSoft [61] and later adopted by the CHESS [107] verification tool. Both of these tools assume the existence of an initialization function that reconstructs the initial state.

The advantage of the stateful approach is the speed of state reconstruction at the expense of the space needed to store the information needed for the reconstruction. The advantage of the stateless approach is the small space requirement needed to store a state at the cost of needing to replay part of the execution. Notably, stateful searches typically first run out of memory, while stateless searches typically first run out of time. Finally, some tools such as the SPIN model checker [71] try to balance the time and space requirements by combining the stateful and the stateless approaches.

Similar to VeriSoft, the methods presented in this thesis use a stateless exploration and assume existence of a mechanism that sets up the initial exploration state.

8.5 Safety / Liveness / Data Race Checkers

Informally, a *safety* property is defined to be a property that can be violated by a finite execution of a program, while a *liveness* property is defined to be a property that can only be violated by an infinite execution of a program. A *data race* occurs when two or more threads perform non-commutative data operations concurrently. Although many data races are intentional or benign, some data races can corrupt the program state or lead to a crash [122].

With the exception of MaceMC [82], execution-based checkers search for violation of safety properties only. The reason for this is that the checking of an unmodified legacy program for liveness properties is generally undecidable [121]. Although there has been some progress on automated proving of liveness properties [36, 37], the state of the art methods do not scale to the size and complexity of unmodified legacy programs.

Given the wide-spread occurrence of data races, a number of tools focus on detection, avoidance, and prediction of data races dating. Notable early work on data race detection includes Mellor-Crummey's on-the-fly detection of data races in fork-join programs [101] and Eraser [128], a tool for detecting data races in legacy programs. Further research in this space produced both static [46] and dynamic [25, 50, 169] methods for detecting data races. Many of these tools make use of the *happens-before* relation [88], a powerful mechanism for reasoning about concurrent program events. The improvements in data race detection tools tend to reduced both the false positive rate and the runtime overhead. Recently, a light-weight form of memory access sampling [48] has been shown to be effective at detecting data races in the Windows operating system kernel.

The methods and tools presented in this thesis focus on safety properties and, for the reasons discussed in Chapter 3, dBug considers program transitions at the granularity of function call interleavings. Consequently, the methods and tools presented in this thesis focus on coarse-grained concurrency errors such as deadlocks or incorrect uses of an API. Fortunately, many of the existing data race checkers [25, 128, 169] are compatible with systematic testing of dBug and can be use to increase its precision.

Chapter 9

Conclusions

This thesis makes two important points: 1) it demonstrates the practicality of the systematic approach to testing of concurrent programs, and 2) it quantifies and advances the practical limits of systematic testing of concurrent programs.

The practicality of the systematic testing approach is demonstrated by presenting two tools. ETA [138] is a tool that targets systematic testing of components of the Omega cluster management system [129], while dBug [137] is a tool that targets systematic testing of unmodified binaries of concurrent programs written for POSIX-compliant operating systems. These tools were used to systematically test over 100 different concurrent programs written in a number of languages, including C, C++, and Fortran, spanning a range of programming paradigms, such as actors [2], `pthreads` [120], OpenMP [111], and MPI [103].

To measure and advance the practical limits of systematic testing, this thesis presents novel results in several research directions ranging from state space estimation to parallel state space exploration to state space reduction.

To measure the extent of state space explosion, this thesis pioneers research on state space estimation of systematic tests, presenting a number of techniques that can be used for predicting the length of a systematic test. The techniques are independent of the exploration algorithm and thus compatible with a wide range of existing systematic testing tools. The techniques have been implemented in both ETA and dBug and used to estimate the extent of state space explosion for over 100 different programs. The evaluation of these techniques demonstrated that the techniques achieve good accuracy and that the they can be used to drive efficient allocation of testing resources.

To speed up long-running systematic tests, this thesis improves on previous work on efficient state space exploration algorithms, enabling systematic testing on large scale computational clusters. In particular, our scalable implementation of dynamic partial order reduction [139], a state of the art algorithm for efficient state space exploration, has been demonstrated to achieve strong scaling on a cluster of over 1,000 machines, exploring millions of different test executions in a matter of minutes.

To mitigate state space explosion for multithreaded programs, this thesis combines systematic testing with restricted runtime scheduling. To this end, dBug has been

integrated with Parrot, an implementation of restricted runtime scheduling for POSIX-compliant operating systems [39]. The integration takes advantage of modular design of both of the tools and is accomplished through a simple coordination API. The end result demonstrates the synergy between systematic testing and restricted runtime scheduling. Systematic testing checks the schedules allowed by restricted runtime scheduling, while restricted runtime scheduling reduces the number of schedules systematic testing needs to check. Arguably, the state space reduction accomplished by combining systematic testing with restricted runtime scheduling represents a leap forward in combating the state space explosion problem. Notably, unlike previous work [51, 67], this reduction technique scales to many threads and long test executions.

To mitigate state space explosion for multiprocess programs, this thesis makes use of abstraction and demonstrates how high-level inter-process coordination APIs such as MPI [103] can be used to soundly abstract executions of multiprocess programs, reducing the number of abstract program states that need to be examined. To demonstrate the practical potential of reduction through abstraction, dBug has been extended with support for MPI. Our evaluation shows that abstraction reduction is orthogonal to other reduction techniques, such as dynamic partial order reduction [51], and their combination helps to thoroughly test multiprocess programs.

As for future work, there are several directions in which the research exploration carried out by this thesis could be followed up. First, given the value provided by state space estimates, it would be interesting to see if their accuracy can be improved by adopting a machine learning approach that learns the typical structure of execution trees of real-world programs over time. Second, given the benefit of combining systematic testing with restricted runtime scheduling of intra-process synchronizations, it would be interesting to see whether a similar path could be taken for restricted runtime scheduling of inter-processes synchronizations. The main challenge of this research direction is the creation of a practical runtime for deterministic and stable multiprocessing. Third, given the wide spectrum of possible execution tree exploration strategies, it would be interesting to carry out a study that considers a sizable set of concurrent programs and contrasts different exploration strategies using metrics such as time and space complexity, estimate accuracy convergence, and bug coverage.

In conclusion, this thesis provides strong evidence in support of its statement that existing concurrent software can be tested in a scalable and systematic fashion using testing infrastructure which controls nondeterminism and mitigates state space explosion.

Appendix A

dBug and POSIX

The following list enumerates 92 functions from the POSIX interface [119] for which dBug implements non-trivial handlers:

- *Barriers:*
  - `pthread_barrier_init`
  - `pthread_barrier_wait`
  - `pthread_barrier_destroy`
- *Condition Variables:*
  - `pthread_cond_broadcast`
  - `pthread_cond_signal`
  - `pthread_cond_destroy`
  - `pthread_cond_timedwait`
  - `pthread_cond_init`
  - `pthread_cond_wait`
- *I/O Functions:*
  - `accept`
  - `bind`
  - `close`
  - `dup`
  - `dup2`
  - `epoll_create`
  - `epoll_create1`
  - `epoll_ctl`
  - `epoll_wait`
  - `fcntl`
  - `listen`
  - `open`
  - `pipe`
  - `poll`
  - `read`
  - `readv`
  - `recv`
  - `recvfrom`
  - `send`
  - `sendto`
  - `select`
  - `socket`

- `socketpair`
- `unlink`
- `write`
- `writev`

- *Mutexes:*
  - `pthread_mutex_destroy`
  - `pthread_mutex_init`
  - `pthread_mutex_lock`
  - `pthread_mutex_timedlock`
  - `pthread_mutex_trylock`
  - `pthread_mutex_unlock`

- *Non-reentrant Functions:*
  - `gethostbyaddr`
  - `gethostbyname`
  - `inet_ntoa`
  - `strtok`

- *Processes:*
  - `execl`
  - `execv`
  - `execle`
  - `execve`
  - `execlp`
  - `execvp`
  - `exit`
  - `_exit`
  - `fork`
  - `posix_spawn`
  - `posix_spawnp`
  - `setpgid`
  - `setpgrp`
  - `wait`
  - `waitpid`

- *Read-write Locks:*
  - `pthread_rwlock_destroy`
  - `pthread_rwlock_init`
  - `pthread_rwlock_rwlock`
  - `pthread_rwlock_tryrdlock`
  - `pthread_rwlock_timedrdlock`
  - `pthread_rwlock_timedwrlock`
  - `pthread_rwlock_trywrlock`
  - `pthread_rwlock_unlock`
  - `pthread_rwlock_wrlock`

- *Scheduling and Time:*
  - `clock`
  - `gettimeofday`
  - `nanosleep`
  - `sleep`
  - `usleep`
  - `sched_yield`
  - `time`

- *Semaphores:*
  - `sem_close`
  - `sem_destroy`
  - `sem_init`
  - `sem_open`
  - `sem_post`
  - `sem_unlink`
  - `sem_wait`
- *Spin Locks:*
  - `pthread_spin_destroy`
  - `pthread_spin_init`
  - `pthread_spin_lock`
  - `pthread_spin_trylock`
  - `pthread_spin_unlock`
- *Threads:*
  - `pthread_cancel`
  - `pthread_create`
  - `pthread_detach`
  - `pthread_exit`
  - `pthread_join`

Appendix B

dBug and MPI

The following list enumerates 58 functions from the MPI standard [103] for which dBug implements non-trivial handlers:

- *Collective Communication*:
  - `MPI_Allgather`
  - `MPI_Allgatherv`
  - `MPI_Allreduce`
  - `MPI_Alltoall`
  - `MPI_Alltoallv`
  - `MPI_Alltoallw`
  - `MPI_Barrier`
  - `MPI_Bcast`
  - `MPI_Exscan`
  - `MPI_Gather`
  - `MPI_Gatherv`
  - `MPI_Reduce`
  - `MPI_Reduce_scatter`
  - `MPI_Reduce_scatter_block`
  - `MPI_Scan`
  - `MPI_Scatter`
  - `MPI_Scatterv`
- *Communicator Management*:
  - `MPI_Cart_sub`
  - `MPI_Comm_compare`
  - `MPI_Comm_create_group`
  - `MPI_Comm_dup`
  - `MPI_Comm_dup_with_info`
  - `MPI_Comm_free`
  - `MPI_Comm_group`
  - `MPI_Comm_split`
  - `MPI_Comm_split_type`

- *Non-Blocking Point-To-Point Communication*:
  - `MPI_Ibsend`
  - `MPI_Imrecv`
  - `MPI_Improbe`
  - `MPI_Iprobe`
  - `MPI_Irecv`
  - `MPI_Irsend`
  - `MPI_Isend`
  - `MPI_Issend`
  - `MPI_Request_free`
  - `MPI_Test`
  - `MPI_Test_all`
  - `MPI_Test_any`
  - `MPI_Test_some`
  - `MPI_Wait`
  - `MPI_Wait_all`
  - `MPI_Wait_any`
  - `MPI_Wait_some`
- *Point-To-Point Communication*:
  - `MPI_Bsend`
  - `MPI_Buffer_detach`
  - `MPI_Mprobe`
  - `MPI_Mrecv`
  - `MPI_Probe`
  - `MPI_Recv`
  - `MPI_Rsend`
  - `MPI_Send`
  - `MPI_Send_recv`
  - `MPI_Send_recv_replace`
  - `MPI_Ssend`
- *Process Management*:
  - `MPI_Finalize`
  - `MPI_Init`
  - `MPI_Init_thread`

Bibliography

[1] Aget: Multithreaded HTTP Download Accelerator [online]. 2009. URL: http://www.enderunix.org/aget/. [Cited on page 80.]

[2] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. AI TR. MIT Press, 1986. [Cited on pages 11, 15, 69, and 103.]

[3] Gautam Altekar and Ion Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 193–206, New York, NY, USA, 2009. ACM. [Cited on page 99.]

[4] Don Anderson. 3n + 1 benchmark [online]. 2011. URL: http://libdb.wordpress.com/3n1/. [Cited on page 81.]

[5] Apache Thrift [online]. 2013. URL: http://thrift.apache.org/. [Cited on pages 39 and 88.]

[6] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley Pub. Co., 1996. [Cited on page 88.]

[7] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient System-Enforced Deterministic Parallelism. In *Proceedings of the 9th USENIX Symposium on Operating System Design and Implementation (OSDI '10)*, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association. [Cited on pages 73, 74, 75, and 85.]

[8] David H. Bailey, Eric Barszcz, John T. Barton, David S. Browning, Russell L. Carter, Leo Dagum, Rod A. Fatoohi, Paul O. Frederickson, Thomas A. Lasinski, Rob S. Schreiber, Horst D. Simon, V. Venkatakrishnan, and Sisira K. Weeratunga. The NAS Parallel Benchmarks — Summary and Preliminary Results. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC '91)*, pages 158–165, New York, NY, USA, 1991. ACM. [Cited on pages 81, 87, 93, and 98.]

[9] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough Static Analysis of Device Drivers. In *Proceedings of the 1st ACM SIGOPS European Conference on Computer Systems (EuroSys '06)*, pages 73–85, New York, NY, USA, 2006. ACM. [Cited on page 5.]

[10] Albert-László Barabási and Réka Albert. Emergence of Scaling in Random Networks. *Science*, 286:509–512, 1999. [Cited on page 60.]

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 164–177, New York, NY, USA, 2003. ACM. [Cited on page 31.]

[12] Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. DiVinE 3.0: An Explicit-State Model Checker for Multithreaded C and C++ Programs. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV '13)*, pages 863–868, Berlin, Heidelberg, 2013. Springer-Verlag. [Cited on pages 6 and 72.]

[13] Gerd Behrmann, Kim Guldstrand Larsen, and Radek Pelánek. To Store or Not to Store. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV '03)*, pages 433–445, 2003. [Cited on page 7.]

[14] Michel Bérard, Béatrice Bodiot, Alain Finkel, Francois Laroussinie, Antoine Petit, Philippe Schnoebelen, Laure Petrucci, and Pierre McKenzie. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2010. URL: `http://books.google.com/books?id=pj78kQAACAAJ`. [Cited on page 5.]

[15] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In James C. Hoe and Vikram S. Adve, editors, *Proceedings of 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 53–64, 2010. [Cited on pages 73, 74, and 85.]

[16] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic Process Groups in dOS. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association. [Cited on page 85.]

[17] Emery Berger, Tongping Liu, and Charlie Curtsinger. dthreads: Efficient and Deterministic Multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, 2011. [Cited on pages 73, 74, 75, and 85.]

[18] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe Multithreaded Programming for C/C++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*, pages 81–96, New York, NY, USA, 2009. ACM. [Cited on pages 73, 74, 75, and 85.]

[19] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of ACM*, 53(2):66–75, 2010. [Cited on page 101.]

[20] Christian Bienia, Sanjeev Kumar, and Kai Li. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on Chip-Multiprocessors.

In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '08)*, pages 47–56, 2008. [Cited on page 81.]

[21] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, October 2008. [Cited on page 81.]

[22] Gunnar Blom, Lars Holst, and Dennis Sandell. *Problems and Snapshots from the World of Probability*. Springer, 1994. [Cited on page 3.]

[23] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A Type and Effect System for Deterministic Parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*, pages 97–116, New York, NY, USA, 2009. ACM. [Cited on page 85.]

[24] William J. Bolosky, John R. Douceur, and Jon Howell. The Farsite Project: A Retrospective. *SIGOPS Operating Systems Review*, 41(2):17–26, 2007. [Cited on pages 5 and 23.]

[25] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. PACER: Proportional Detection of Data Races. In *Proceedings of the 31th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*, pages 255–268, New York, NY, USA, 2010. ACM. [Cited on pages 31 and 102.]

[26] Randy Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. [Cited on page 97.]

[27] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel Symbolic Execution for Automated Real-World Software Testing. In *Proceedings of the 6th ACM SIGOPS European Conference on Computer Systems (EuroSys '11)*, pages 183–198, 2011. [Cited on pages 7, 72, 97, and 100.]

[28] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, New York, NY, USA, 1997. ACM. [Cited on page 31.]

[29] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 209–224, 2008. [Cited on pages 6, 97, 100, and 101.]

[30] Satish Chandra, Patrice Godefroid, and Christopher Palm. Software Model Checking in Practice: An Industrial Case Study. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, pages 431–441, 2002. [Cited on page 12.]

[31] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, Inc., Orlando, FL, USA, 1973. [Cited on page 5.]

[32] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. BigTable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 205–218, 2006. [Cited on page 66.]

[33] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986. [Cited on page 5.]

[34] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994. [Cited on page 97.]

[35] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999. [Cited on page 5.]

[36] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving That Programs Eventually Do Something Good. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*, pages 265–276, New York, NY, USA, 2007. ACM. [Cited on page 102.]

[37] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination Proofs for Systems Code. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*, pages 415–426, New York, NY, USA, 2006. ACM. [Cited on page 102.]

[38] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs By Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252, New York, NY, USA, 1977. ACM. [Cited on pages 14, 87, and 96.]

[39] Heming Cui, Jiří Šimša, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth Gibson, and Randy Bryant. PARROT: A Practical Runtime for Deterministic, Stable, and Reliable Threads. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, 2013. [Cited on pages 4, 14, 73, 74, 75, 80, 82, and 104.]

[40] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient Deterministic Multithreading through Schedule Relaxation. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, 2011. [Cited on pages 73, 74, 75, and 85.]

[41] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable Deterministic

Multithreading through Schedule Memoization. In *Proceedings of the 9th USENIX Symposium on Operating System Design and Implementation (OSDI '10)*, October 2010. [Cited on pages 73, 74, 75, and 85.]

[42] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 85–96, New York, NY, USA, 2009. ACM. [Cited on pages 73, 74, and 85.]

[43] David A. Duffy. *Principles of Automated Theorem Proving*. Wiley, 1991. [Cited on page 5.]

[44] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective Distributed Scheduling of Parallel Workloads. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '96)*, pages 25–36, New York, NY, USA, 1996. ACM. [Cited on page 76.]

[45] Matthew B. Dwyer, Sebastian Elbaum, Suzette Person, and Rahul Purandare. Parallel Randomized State-Space Search. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society. [Cited on page 71.]

[46] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. *SIGOPS Operating Systems Review*, 37:237–252, October 2003. [Cited on pages 86 and 102.]

[47] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI '00)*, pages 1–1, Berkeley, CA, USA, 2000. USENIX Association. [Cited on page 101.]

[48] John Erickson, Madanlal Musuvathi, Sebastian Bruckhardt, and Kirk Olynyk. Effective Data-Race Detection for the Kernel. In *Proceedings of the 9th USENIX Symposium on Operating System Design and Implementation (OSDI '10)*, Berkeley, CA, USA, 2010. USENIX Association. [Cited on page 102.]

[49] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Springer, 2012. [Cited on page 5.]

[50] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*, pages 121–133, New York, NY, USA, 2009. ACM. [Cited on page 102.]

[51] Cormac Flanagan and Patrice Godefroid. Dynamic Partial Order Reduction for Model Checking Software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '05)*, pages 110–121, New York, NY, USA, 2005. ACM. [Cited on pages 4, 6, 8, 9, 10, 15, 17, 21, 45, 61, 62, 81, 84, 85, 94, 97, 98, and 104.]

[52] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th USENIX Symposium on Operating System Design and Implementation (OSDI '10)*, pages 1–7, Berkeley, CA, USA, 2010. USENIX Association. [Cited on pages 66 and 69.]

[53] Leonor Frias and Johannes Singler. Parallelization of bulk operations for stl dictionaries. In *Euro-Par 2007 Workshops: Parallel Processing*, pages 49–58, Berlin, Heidelberg, 2008. Springer-Verlag. [Cited on page 80.]

[54] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global Comprehension for Distributed Replay. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI '07)*, 2007. [Cited on page 99.]

[55] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay Debugging for Distributed Applications. In *Proceedings of the 11th USENIX Annual Technical Conference (USENIX '06)*, pages 27–27, Berkeley, CA, USA, 2006. USENIX Association. [Cited on pages 35 and 99.]

[56] David Gelperin and Bill Hetzel. The Growth of Software Testing. *Communications of ACM*, 31(6):687–695, 1988. [Cited on pages 2 and 5.]

[57] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. *SIGOPS Operating Systems Review*, 37(5):29–43, 2003. [Cited on page 66.]

[58] Garth Gibson, Gary Grider, Andree Jacobson, and Wyatt Lloyd. PRObE: A Thousand-Node Experimental Cluster for Computer Systems Research. *;login:*, 38(3):37–39, June 2013. [Cited on page 94.]

[59] Jeff Gilchrist. Parallel BZIP2 [online]. 2011. URL: `http://compression.ca/pbzip2/`. [Cited on page 80.]

[60] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Springer-Verlag, 1996. [Cited on pages 6, 8, 9, 10, 15, 17, and 27.]

[61] Patrice Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 174–186. ACM, 1997. [Cited on pages 5, 6, 7, 10, 11, 12, 43, 62, 85, 100, and 101.]

[62] Patrice Godefroid. Software Model Checking: The VeriSoft Approach. *Formal Methods in System Design*, 26(2):77–101, 2005. [Cited on pages 10 and 100.]

[63] Patrice Godefroid, Robert S. Hanmer, and Lalita Jategaonkar Jagadeesan. Model Checking Without a Model: An Analysis of the Heart-Beat Monitor of a Telephone Switch Using VeriSoft. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '98)*, pages 124–133, 1998. [Cited on page 12.]

[64] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed Automated

Random Testing. In *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pages 213–223, New York, NY, USA, 2005. ACM. [Cited on page 100.]

[65] Google Data Centers [online]. 2013. URL: http://www.google.com/about/datacenters/. [Cited on pages 20 and 69.]

[66] Google Test [online]. 2013. URL: http://code.google.com/p/googletest/. [Cited on page 19.]

[67] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 265–278, New York, NY, USA, 2011. ACM. [Cited on pages 72, 81, 84, 85, and 104.]

[68] Zhenyu Guo, Xi Wang, Xuezheng Liu, Wei Lin, and Zheng Zhang. BOX: Icing the APIs. Technical Report MSR-TR-2008-03, Microsoft Research, 2008. [Cited on pages 14, 99, and 100.]

[69] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI '11)*, Berkeley, CA, USA, 2011. USENIX Association. [Cited on page 59.]

[70] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991. [Cited on page 97.]

[71] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. [Cited on pages 5, 6, 7, and 102.]

[72] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2004. [Cited on pages 5 and 97.]

[73] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm Verification Techniques. *IEEE Transactions on Software Engineering*, 37:845–857, 2011. [Cited on page 72.]

[74] William E. Howden. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Transactions on Software Engineering*, 3:266–278, July 1977. [Cited on page 100.]

[75] Nicholas Hunt, Tom Bergan, Luis Ceze, and Steven D. Gribble. DDOS: Taming Nondeterminism in Distributed Systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 499–508, New York, NY, USA, 2013. ACM. [Cited on page 85.]

[76] ImageMagick [online]. 2013. URL: http://www.imagemagick.org. [Cited on page 80.]

[77] David B. Jackson, Quinn Snell, and Mark J. Clement. Core Algorithms of the Maui Scheduler. In *Proceedings of the 7th International Workshop on Job Scheduling*

Strategies for Parallel Processing (JSSPP '01), pages 87–102, London, UK, UK, 2001. Springer-Verlag. [Cited on page 59.]

[78] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated Concurrency-Bug Fixing. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 221–236, Berkeley, CA, USA, 2012. USENIX Association. [Cited on page 86.]

[79] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock Immunity: Enabling Systems to Defend Against Deadlocks. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 295–308, Berkeley, CA, USA, 2008. USENIX Association. [Cited on page 86.]

[80] Philip Kilby, John K. Slaney, Sylvie Thiébaux, and Toby Walsh. Estimating Search Tree Size. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI '06)*, pages 1014–1019, 2006. [Cited on pages 41, 44, 46, 48, and 59.]

[81] Charles E. Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language Support for Building Distributed Systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pages 179–188, 2007. [Cited on pages 11, 13, and 100.]

[82] Charles E. Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI '07)*, 2007. [Cited on pages 7, 10, 11, 13, 43, 72, 100, and 102.]

[83] James C. King. Symbolic Execution and Program Testing. *Communications of ACM*, 19:385–394, July 1976. [Cited on pages 2 and 100.]

[84] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 207–220. ACM, 2009. [Cited on page 5.]

[85] Donald E. Knuth. Estimating the Efficiency of Backtrack Programs. *Mathematics of Computation*, 29(129):121–136, 1975. [Cited on page 46.]

[86] Robert P. Kurshan. Analysis of Discrete Event Coordination. In *Proceedings on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, pages 414–453, New York, NY, USA, 1990. Springer-Verlag. [Cited on page 97.]

[87] Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. Static and Precise Detection of Concurrency Errors in Systems Code Using SMT Solvers. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09)*, pages 509–524, Berlin, Heidelberg, 2009. Springer-Verlag. [Cited on page 101.]

[88] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of ACM*, 21(7):558–565, 1978. [Cited on pages 10, 15, 17, 97, and 102.]

[89] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998. [Cited on page 14.]

[90] Kenneth Levenberg. A Method for the Solution of Certain Non-linear Problems in Least Squares. *Quarterly of Applied Mathematics*, 2(2):164–168, 1944. [Cited on page 48.]

[91] Nancy G. Leveson and Clark S. Turner. An Investigation of the Therac-25 Accidents. *Computer*, 26(7):18–41, 1993. [Cited on page 73.]

[92] The GNU C Library [online]. 2013. URL: `http://www.gnu.org/software/libc/manual`. [Cited on page 90.]

[93] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: Debugging Deployed Distributed Systems. In *Proceedings of the 5th USENIX Conference on Networked Systems Design and Implementation (NSDI '08)*, pages 423–437, Berkeley, CA, USA, 2008. USENIX Association. [Cited on page 99.]

[94] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation (NSDI '07)*, pages 19–19, Berkeley, CA, USA, 2007. USENIX Association. [Cited on page 99.]

[95] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis*. North-Holland Publishing Company, 1978. [Cited on page 5.]

[96] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI: Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, pages 103–116, New York, NY, USA, 2007. ACM. [Cited on page 86.]

[97] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes: A Comprehensive Study on Real-World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 329–339, New York, NY, USA, 2008. ACM. [Cited on page 73.]

[98] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pages 190–200, New York, NY, USA, 2005. ACM. [Cited on page 31.]

[99] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specifications*. The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, 1992. [Cited on pages 12 and 19.]

[100] Donald W. Marquardt. An Algorithm for Least-Squares Estimation of Nonlinear

Parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):431–441, 1963. [Cited on page 48.]

[101] John Mellor-Crummey. On-the-fly Detection of Data Races for Programs with Nested Fork-join Parallelism. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (SC '91)*, pages 24–33, New York, NY, USA, 1991. ACM. [Cited on page 102.]

[102] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst Simon. Top 500 Supercomputing Sites [online]. 2013. URL: `http://www.top500.org`. [Cited on pages xv, 1, 2, and 82.]

[103] MPI: A Message-Passing Interface Standard Version 3.0 [online]. September 2012. URL: `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`. [Cited on pages 4, 11, 13, 22, 87, 88, 90, 98, 103, 104, and 109.]

[104] MPlayer [online]. 2013. URL: `http://www.mplayerhq.hu/design7/news.html`. [Cited on page 80.]

[105] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, 2002. [Cited on pages 6, 10, 11, 12, and 101.]

[106] Madanlal Musuvathi and Shaz Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pages 446–455, 2007. [Cited on pages 10, 11, and 101.]

[107] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam A. Nainar, and Iulian Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 267–280, 2008. [Cited on pages 7, 10, 11, 14, 43, 62, 100, and 101.]

[108] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the 4th USENIX Annual Technical Conference (USENIX '99)*, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association. [Cited on pages 13, 14, and 80.]

[109] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 97–108, New York, NY, USA, 2009. ACM. [Cited on pages 73, 74, and 85.]

[110] OpenLDAP [online]. 2013. URL: `http://www.openldap.org/`. [Cited on page 80.]

[111] OpenMP Application Program Interface [online]. July 2011. [Cited on pages 81, 87, 94, and 103.]

[112] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of 3rd International Conference on Distributed Computing Systems (ICDCS '82)*, pages 22—-30, 1982. [Cited on page 76.]

[113] Parallel File Scanner [online]. 2013. URL: http://ostatic.com/pfscan. [Cited on page 80.]

[114] Chang-Seo Park and Koushik Sen. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08)*, pages 135–145, New York, NY, USA, 2008. ACM. [Cited on page 86.]

[115] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, pages 25–36, New York, NY, USA, 2009. ACM. [Cited on page 86.]

[116] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 14th ACM SIGMOD International Conference on Management of Data (SIGMOD '88)*, pages 109–116, New York, NY, USA, 1988. ACM. [Cited on page 13.]

[117] Charles E. Perkins and Elizabeth M. Royer. Ad-hoc On-Demand Distance Vector Routing. In *Proceedings of 2nd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '99)*, pages 90–100, New Orleans, LA, February 1999. [Cited on page 12.]

[118] Salman Pervez, Ganesh Gopalakrishnan, Robert M. Kirby, Robert Palmer, Rajeev Thakur, and William Gropp. Practical Model-Checking Method for Verifying Correctness of MPI Programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 344–353, Berlin, Heidelberg, 2007. Springer-Verlag. [Cited on pages 10, 11, 13, and 97.]

[119] Portable Operating System Interface (POSIX) Base Specifications, Issue 7 [online]. December 2008. URL: http://pubs.opengroup.org/onlinepubs/9699919799/. [Cited on pages 11, 15, 20, 32, 87, and 105.]

[120] POSIX Threads [online]. 1997. URL: http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html. [Cited on pages 4, 76, 81, and 103.]

[121] Emil L. Post. A Variant of a Recursively Unsolvable Problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946. [Cited on page 102.]

[122] Kevin Poulsen. Software Bug Contributed to Blackout [online]. February 2004. URL: http://www.securityfocus.com/news/8016. [Cited on pages 73 and 102.]

[123] Michel Raynal and Mukesh Singhal. Logical Time: Capturing Causality in Distributed Systems. *Computer*, 29(2):49–56, 1996. [Cited on page 32.]

[124] Redis [online]. 2013. URL: http://redis.io/. [Cited on page 80.]

[125] John A. Robinson and Andrei Voronkov. *Handbook of Automated Reasoning*. Elsevier, 2001. [Cited on page 5.]

[126] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware '01)*, pages 329–350. Springer-Verlag, 2001. [Cited on page 13.]

[127] Yasushi Saito. Jockey: A User-Space Library for Record-Replay Debugging. In *Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging (AADEBUG '05)*, pages 69–76, New York, NY, USA, 2005. ACM. [Cited on page 35.]

[128] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997. [Cited on pages 31, 86, and 102.]

[129] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM SIGOPS European Conference on Computer Systems (EuroSys '13)*, pages 351–364, New York, NY, USA, 2013. ACM. [Cited on pages 15, 59, 69, and 103.]

[130] Koushik Sen. Race Directed Random Testing of Concurrent Programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*, pages 11–21, New York, NY, USA, 2008. ACM. [Cited on page 86.]

[131] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, pages 263–272, New York, NY, USA, 2005. ACM. [Cited on page 100.]

[132] Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow, Carl Beame, Mike Eisler, and David Noveck. Network File System (NFS) version 4 Protocol [online]. April 2003. URL: https://www.ietf.org/rfc/rfc3530.txt. [Cited on page 13.]

[133] Stephen F. Siegel. Model checking nonblocking mpi programs. In *Proceeding of 8th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '07)*, pages 44–58, Berlin, Heidelberg, 2007. Springer. [Cited on page 97.]

[134] Stephen F. Siegel and Timothy K. Zirkel. TASS: The Toolkit for Accurate Scientific Software. *Mathematics in Computer Science*, 5(4):395–426, 2011. [Cited on page 97.]

[135] Jiří Šimša. dBug: Systematic Testing of Distributed and Multi-Threaded Systems [online]. 2013. URL: http://www.cs.cmu.edu/~jsimsa/dbug. [Cited on

pages 10, 11, 20, and 30.]

[136] Jiří Šimša, Randy Bryant, and Garth Gibson. dBug: Systematic Evaluation of Distributed Systems. In *Proceedings of the 5th International Conference on Systems Software Verification (SSV '10)*, pages 1–9, Berkeley, CA, USA, 2010. USENIX Association. [Cited on pages 10 and 30.]

[137] Jiří Šimša, Randy Bryant, and Garth Gibson. dBug: Systematic Testing of Unmodified Distributed and Multi-threaded Systems. In *Proceedings of the 18th International Workshop on Model Checking of Software (SPIN '11)*, pages 188–193, Berlin, Heidelberg, 2011. Springer-Verlag. [Cited on pages 10, 20, 30, 43, 62, 85, and 103.]

[138] Jiří Šimša, Randy Bryant, Garth Gibson, and Jason Hickey. Efficient Exploratory Testing of Concurrent Systems. Technical Report CMU-PDL-11-113, Parallel Data Laboratory, Carnegie Mellon University, November 2011. [Cited on pages 10, 11, 15, 43, and 103.]

[139] Jiří Šimša, Randy Bryant, Garth Gibson, and Jason Hickey. Scalable Dynamic Partial Order Reduction. In *Proceedings of the 3rd International Conference on Runtime Verification (RV '12)*, 2013. [Cited on pages 14, 59, 62, 69, and 103.]

[140] Jiří Šimša and John Wilkes. ETA Exploration Traces [online]. September 2012. URL: `http://code.google.com/p/googleclusterdata/wiki/ETAExplorationTraces`. [Cited on pages 43, 44, and 50.]

[141] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *SIGARCH Computer Architecture News*, 20(1):5–44, March 1992. [Cited on page 81.]

[142] Johannes Singler, Peter Sanders, and Felix Putze. MCSTL: The Multi-core Standard Template Library. In *Euro-Par 2007 Parallel Processing*, pages 682–694, Berlin, Heidelberg, 2007. Springer-Verlag. [Cited on page 80.]

[143] Matt Staats and Corina Păsăreanu. Parallel Symbolic Execution for Structural Test Generation. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '10)*, pages 183–194, New York, NY, USA, 2010. ACM. [Cited on page 72.]

[144] Ulrich Stern and David L. Dill. Parallelizing the MurPhi Verifier. *Formal Methods in System Design*, 18(2):117–129, 2001. [Cited on page 72.]

[145] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '01)*, San Diego, CA, September 2001. [Cited on page 13.]

[146] Ali Taleghani and Joanne M. Atlee. State-Space Coverage Estimation. In *Proceedings of 28th IEEE/ACM International Conference Automated Software Engineering (ASE '09)*, pages 459–467, 2009. [Cited on page 59.]

[147] Sarvani Vakkalanka. *Efficient Dynamic Verification Algorithms for MPI Applications*. PhD thesis, University of Utah, August 2010. [Cited on pages 13, 88, and 97.]

[148] Sarvani Vakkalanka, Subodh Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: A Tool for Model Checking MPI Programs. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, pages 285–286, 2008. [Cited on pages 10 and 97.]

[149] Guido Van Rossum. *Python Reference Manual*. iUniverse, 2000. [Cited on page 88.]

[150] Rob van Stee. *Combinatorial Algorithms for Packing and Scheduling Problems*. PhD thesis, Universität Karlsruhe, June 2008. [Cited on pages 59 and 60.]

[151] Joannès Vermorel and Mehryar Mohri. Multi-Armed Bandit Algorithms and Empirical Evaluation. In *Proceedings of the 16th European Conference on Machine Learning (ECML '05)*, pages 437–448, Berlin, Heidelberg, 2005. Springer-Verlag. [Cited on page 60.]

[152] Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (SC '00)*, 2000. [Cited on page 97.]

[153] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engineering*, 10(2):203–232, April 2003. [Cited on page 59.]

[154] Jeffrey S. Vitter. Random Sampling With a Reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, March 1985. [Cited on page 49.]

[155] Anh Vo. *Scalable Formal Dynamic Verification of MPI Programs through Distributed Causality Tracking*. PhD thesis, University of Utah, August 2011. [Cited on pages 88 and 97.]

[156] Anh Vo, Sriram Aananthakrishnan, Ganesh Gopalakrishnan, Bronis R. de Supinski, Martin Schulz, and Greg Bronevetsky. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*, pages 1–10, 2010. [Cited on page 97.]

[157] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott Mahlke. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 281–294, Berkeley, CA, USA, 2008. USENIX Association. [Cited on page 86.]

[158] Jingyue Wu, Heming Cui, and Junfeng Yang. Bypassing Races in Live Applications with Execution Filters. In *Proceedings of the 9th USENIX Symposium on Operating System Design and Implementation (OSDI '10)*, pages 1–13, Berkeley, CA, USA, 2010. USENIX Association. [Cited on page 86.]

[159] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma.

Ad-hoc Synchronization Considered Harmful. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI '10)*, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association. [Cited on page 81.]

[160] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MoDist: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Conference on Networked Systems Design and Implementation (NSDI '09)*, pages 213–228, April 2009. [Cited on pages 7, 10, 11, 14, 43, 62, 72, 85, and 100.]

[161] Junfeng Yang, Heming Cui, and Jingyue Wu. Determinism Is Overrated: What Really Makes Multithreaded Programs Hard to Get Right and What Can Be Done about It? In *Proceedings of the 5th USENIX Workshop on Hot Topics in Parallelism (HotPar '13)*, June 2013. [Cited on page 75.]

[162] Junfeng Yang, Heming Cui, Jingyue Wu, Yang Tang, and Gang Hu. Determinism Is Not Enough: Making Parallel Programs Reliable with Stable Multithreading. *To Appear in Communications of ACM*, 2014. [Cited on page 75.]

[163] Junfeng Yang, Can Sar, and Dawson R. Engler. eXplode: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 131–146. USENIX Association, 2006. [Cited on pages 7, 10, 11, 13, and 43.]

[164] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. *ACM Transactions on Computer Systems*, 24(4):393–423, 2006. [Cited on page 10.]

[165] Junfeng Yang, Paul Twohey, Dawson R. Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 273–288, 2004. [Cited on pages 6, 10, 11, 13, 43, and 101.]

[166] Yu Yang, Xiaofang Chen, and Ganesh Gopalakrishnan. Inspect: A Runtime Model Checker for Multithreaded C Programs. Technical Report UUCS-08-004, University of Utah, 2008. [Cited on page 72.]

[167] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Distributed Dynamic Partial Order Reduction Based Verification of Threaded Software. In *Proceedings of the 14th International Workshop on Model Checking of Software (SPIN '07)*, pages 58–75. Springer-Verlag, 2007. [Cited on pages 20, 61, 62, 63, 65, and 72.]

[168] Richard M. Yoo, Anthony Romano, and Christos Kozyrakis. Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC '09)*, pages 198–207, Washington, DC, USA, 2009. IEEE Computer Society. [Cited on page 81.]

[169] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *Proceedings of the 20th ACM Symposium*

on Operating Systems Principles (SOSP '05), pages 221–234, New York, NY, USA, 2005. ACM. [Cited on pages 31, 86, and 102.]

[170] Wei Zhang, Chong Sun, and Shan Lu. ConMem: Detecting Severe Concurrency Bugs Through an Effect-Oriented Approach. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 179–192, New York, NY, USA, 2010. ACM. [Cited on page 86.]