EFFICIENT DYNAMIC VERIFICATION OF CONCURRENT PROGRAMS

by

Yu Yang

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

May 2009

Copyright © Yu Yang 2009

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Yu Yang

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Ganesh Gopalakrishnan
	Aarti Gupta
	Robert M. Kirby
	John Regehr
	Gary Lindstrom

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Yu Yang in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Ganesh Gopalakrishnan
Chair: Supervisory Committee

Approved for the Major Department

Martin Berzins
Chair/Director

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Revealing concurrency errors in multithreaded programs is difficult. Many "unexpected" thread interactions can only be manifested with intricate low probability event sequences. As a result, they often escape conventional testing, and manifest years after code deployment.

Dynamic verification methods have proven promising for revealing errors in the implementation of real world concurrent programs. They work on real applications and libraries, and side-step the high complexity of model construction and state capture by concretely executing programs, and replaying the executions for covering the different thread interleavings. However, with the increase of the program size, the state space – specifically the number of schedules – of a concurrent program grows exponentially. To solve this problem and make dynamic verification be applicable to general concurrent programs, we make the following contributions.

First, we propose a family of algorithms for reducing the search spaces of concurrent programs for dynamic verification. The first algorithm is stateful dynamic partial order reduction, which detects visited states by capturing the states of multithreaded programs in concrete executions in a light-weight way. The second algorithm is property-driven pruning, which infers properties of a subspace by exploring only part of the subspace. The third algorithm is automatic symmetry discovery, which reveals symmetry in multithreaded programs using dynamic program analysis. The fourth algorithm is distributed dynamic partial order reduction, which uses computer clusters to speed up the search and at the same time still get the benefit of dynamic partial order reduction.

Second, we built `Inspect`, a framework for dynamic verification of multithreaded C programs. `Inspect` is able to either expose concurrency bugs inside a multithreaded program, or guarantee that the program is free from concurrency errors under the specific

test harnesses. `Inspect` combines program analysis, program instrumentation and dynamic model checking in a unique way to realize efficient dynamic verification.

We have implemented our efficient dynamic verification algorithms in `Inspect` and applied `Inspect` to examine realistic multithreaded applications. The experiments show that our algorithms can significantly reduce the time to verify realistic multithreaded applications while guaranteeing completeness and soundness.

CONTENTS

ABSTRACT . . . . . **iv**

LIST OF FIGURES . . . . . **ix**

LIST OF TABLES . . . . . **xi**

CHAPTERS

1. INTRODUCTION **1**

1.1 Motivation 1
1.1.1 The Problem of Concurrency Bugs 1
1.1.2 The Need for Dynamic Verification 2
1.2 Related Work 3
1.2.1 Static Analysis 3
1.2.2 Dynamic Detection 4
1.2.3 Formal Static Verification 5
1.3 Problem Statement 6
1.4 Dissertation Statement 6
1.4.1 Stateful dynamic partial order reduction [99] 7
1.4.2 Property driven pruning for detecting data races [94] 7
1.4.3 Symmetry discovery with dynamic program analysis [100] 7
1.4.4 Distributed dynamic partial order reduction [98] 8
1.4.5 Inspect: a Framework for Dynamic Verification 8
1.5 Contributions 9
1.6 Outline 9

2. PRELIMINARIES **11**

2.1 Concurrent Programs 11
2.2 Partial Order Reduction 12
2.3 Dynamic Partial Order Reduction 13
2.3.1 An Example on DPOR 16
2.4 Summary 16

3. INSPECT: A DYNAMIC VERIFICATION FRAMEWORK **17**

3.1 Overview 17
3.2 An Example 18
3.3 Inspect 20
3.3.1 Identifying Threads and Shared Objects Across Executions 20
3.3.2 Instrumenting the Code 20

3.3.3 Avoiding Redundant Backtracking 22
3.4 Implementation 24
3.5 Related Work 25
3.6 Summary 26

4. STATEFUL DYNAMIC PARTIAL ORDER REDUCTION 27

4.1 Capturing Local States of Threads 30
4.2 Stateful Dynamic Partial Order Reduction 34
4.2.1 Efficient SDPOR 41
4.3 Implementation 41
4.4 Experimental Results 43
4.5 Related Work 43
4.6 Summary 45

5. PROPERTY-DRIVEN PRUNING FOR FAST DATA RACE DETECTION 46

5.1 Race-Free Search Subspace 48
5.1.1 Set of Lock Sets 49
5.1.2 Handling the Other Branch 50
5.1.3 Checking Race-Free Subspace 52
5.2 The Overall Algorithm 55
5.2.1 The Running Example 57
5.2.2 Proof of Correctness 58
5.3 Experiments 58
5.4 Experimental Results 59
5.5 Summary 61

6. DISTRIBUTED DYNAMIC PARTIAL ORDER REDUCTION 62

6.1 Algorithm 63
6.1.1 Load Balancing 65
6.1.2 Worker Routine 67
6.1.3 Distributed DPOR 67
6.1.4 Updating the Backtrack Set 69
6.1.5 Correctness 73
6.2 Implementation and Experiments 73
6.3 Related Work 78
6.4 Summary 79

7. AUTOMATIC TRANSITION SYMMETRY DISCOVERY 80

7.1 A Simple C-like Programming Language 83
7.1.1 Residual Code of Threads 84
7.1.2 Symmetry 85
7.2 Discovering Transition Symmetry 87
7.2.1 Inferring Syntactic Equivalence Among Residual Code 87
7.2.2 Discovering Symmetric Transitions 89
7.2.3 Soundness 90

7.3 Dynamic Partial Order Reduction with Symmetry Discovery 92
7.4 Implementation and Evaluation 93
7.5 Experiment Results 95
7.6 Related work 97
7.7 Summary 97

8. CONCLUSION AND FUTURE WORK 99

8.1 Future Research Directions 99
8.1.1 Using Dynamic Program Analysis to Improve Test Case Generation 100
8.1.2 Program Analysis at the Binary Level 100
8.1.3 Testing Concurrent Programs under Open Environments 100

REFERENCES 101

LIST OF FIGURES

2.1 Dynamic partial-order reduction 15
3.1 `Inspect`'s workflow 18
3.2 An example of concurrent operations in a shared database 19
3.3 Syntax of a simple language that is similar to C 21
3.4 `Inspect`'s instrumentation 22
3.5 Instrumented code for the class A thread of Figure 3.2 23
3.6 The internal design of `Inspect` 24
4.1 A simple example for illustrating the idea 27
4.2 Two different executions of a program lead to the same state 28
4.3 Depth-first search with a light-weight state capturing scheme 32
4.4 Computing the local state identity of a thread 33
4.5 Stateful dynamic partial order reduction 37
4.6 Updating the backtrack sets for states in the search stack 38
4.7 SDPOR$_k$ only have immediate backtracking at the first k revisited states. . . 39
4.8 The combination of SDPOR with the light-weight state capturing scheme . 42
5.1 Race condition on accessing variable y (assume that $x = y = 0$ initially) . 47
5.2 Instrumenting the branching statements of each thread 51
5.3 Checking whether the search subspace from s_i is race-free at run time . . . 52
5.4 Computing locksets that may be held by each transition in T_τ 53
5.5 Multiple branches in an execution trace (observed and unobserved branches) 54
5.6 Property driven pruning based dynamic race detection algorithm 56
6.1 The message flow among the load balancer and the workers 64
6.2 The load balancing algorithm 66
6.3 The routine that runs on each worker 67
6.4 Distributed dynamic partial order reduction 68
6.5 A simple example on the potential redundant search among nodes 70
6.6 An initial trace of the program. 70

6.7 Distributing the task between two nodes 71
6.8 An example on the redundant backtracking points 71
6.9 Updating the backtrack sets of states in the distributed context 73
6.10 The intial trace of the program with DDPORUPDATEBACKTRACKSETS . . 74
6.11 Distributing tasks with DDPORUPDATEBACKTRACKSETS 74
6.12 Speedups on *indexer* and *fsbench* 76
6.13 Speedups on the bounded buffer example 77
6.14 Speedups on the aget example 78
7.1 Threads that are forked from the same routine may behave differently 81
7.2 Syntax of a simple language that is similar to C 83
7.3 Examples on the residual code of threads 84
7.4 Rules for inferring syntactic equivalence among residual code of threads . 88
7.5 Checking whether two transitions enabled at s by a and b are symmetric . . 90
7.6 Dynamic partial order reduction with symmetry discovery 94
7.7 The procedure of the probing thread 95

LIST OF TABLES

4.1 Experimental results on the comparison between DPOR and SDPOR 44
5.1 Comparing the performance of two race detection algorithms 60
6.1 Checking time with the sequential `Inspect` 75
7.1 Experimental results on symmetry 96

CHAPTER 1

INTRODUCTION

With the prevalence of software, software reliability becomes more and more important. Software released with undetected bugs may severely affect the dependability of systems and causes huge loss and damage. Concurrency bugs are among the most troublesome software bugs. They are hard to detect and to reproduce. In the past, they have caused several disasters such as the Therac-25 tragedy [62], the dysfunction of the Mars Pathfinder [45], and the 2003 North American Blackout [42]. With the prevalence of multicore hardware, it becomes more urgent to find more effective methods to detect the concurrency bugs.

1.1 Motivation

1.1.1 The Problem of Concurrency Bugs

Software is ubiquitous. Unfortunately, most software contains bugs. Previous studies have shown that software bugs cause about 25-35% of system down time [65, 84] and 50% of security vulnerabilities [46]. A survey in 2002 [73] showed that software bugs cost the US economy around 60 billion dollars annually.

Concurrent programs are software systems that conduct parallel execution of multiple tasks. Concurrency bugs are programming errors that occur in the executions of concurrent programs due to the nondeterminism of parallel execution. Data races and deadlocks are two typical kinds of the concurrency bugs. Among all types of software bugs, concurrency bugs are the most troublesome ones.

As the program's size increases, the number of possible parallel executions grows exponentially. For instance, a concurrent program that has five threads with five steps each may have $25!/(5!)^5 >$10 billion possible executions. Besides, the parallel execution

is nondeterministic. With the same input (a bug triggering one), a concurrency bug may manifest in some runs and disappear in many other runs. Due to these factors, concurrency bugs may only be exposed with a low probability sequence of interactions among threads [59]. This makes concurrency bugs troublesome; they are hard to detect and are hard to reproduce when they appear. Because of this, with the conventional testing methods, a program with concurrency bugs can pass a thorough testing composed of a large number of inputs. As a result, many concurrency bugs sneak into production runs and manifest "unexpectedly" at users' sites under special low probability event sequences [59]. In the past decades, concurrency bugs have caused several disasters. In the 1980s, a concurrency bug in Therac-25, a radiation therapy machine, caused radiation overdoses and killed six patients, with more patients severely injured [62]. In the late 1990s, a concurrency bug in Mars PathFinder made the rover keep rebooting on Mars so that it could not continue its exploration tasks [45]. More recently, a race condition in the energy management system of some power facilities prevented alerts from being raised to the monitoring technicians, eventually leading to the 2003 North American Blackout, which left 50 million people in northeastern America without power and cost around 6 billion dollars of financial loss [42].

Recently, the concurrency bug problem has become more severe. With multicore machines becoming the mainstream hardware, concurrent programs are entering the mainstream software community in order to leverage the multicore resource. These new concurrent programs will inevitably produce many new concurrency bugs in our software.

1.1.2 The Need for Dynamic Verification

Dynamic verification methods have proven promising for revealing errors in the implementation of real world concurrent programs [72]. Dynamic verification [25, 72] uses model checking techniques to systematically explore the state space of a concurrent program under specific test harnesses by concretely executing the program. They work on real applications and libraries, and side step the high complexity of model construction and state capture by concretely executing the programs, and replaying the executions

for covering the different thread interleavings. Given their ready applicability, dynamic verification methods are increasingly successful in practice, as evidenced by projects such as CHESS [72], JPF [35] and MODIST [96] that have recently been proposed and widely used. Within our own research group, another project ISP [89, 92] has proven highly effective in verifying MPI programs directly, without model extraction.

In this dissertation, we will focus on how to improve dynamic verification to efficiently examine shared-memory-based concurrent programs, which are common in current desktop and server environments. We will assume that the executions of these programs follow the sequential consistency memory model [58]. We will also assume that for the compiled executable of a program, no bugs will be introduced in the stage of compiler optimization [4]. We do not consider the executions that are only possible under relaxed memory consistency models [31, 101].

1.2 Related Work

Concurrency bug detection has been studied for a long time. Many algorithms and tools have been designed to reveal concurrency bugs in software. They can be generally classified into several categories: static analysis, dynamic detection, formal static verification and dynamic verification.

1.2.1 Static Analysis

Static analysis detects concurrency bugs by examining the source code of the program. Static data race detection tools [20, 80, 55] detect potential races in concurrent programs by statically computing the lock sets that threads may hold while accessing the shared variables. A warning is generated if the intersection of two lock sets that are associated with the accesses of the same shared variable in different threads are empty. Due to the overapproximating nature of static analysis, the false positive (false alarm) rates of these tools can be high. Advanced program analysis techniques [55, 54] are proposed to improve precision of static analysis to reduce the false positives. Another approach to reveal concurrency bugs is to study the syntactic patterns of the programs. Abnormal code that does not follow a frequent pattern will be reported to the users as potential

bugs. Along this line, some simple patterns such that correlations of variables [63] can be detected automatically. The users can also provide patterns [34] to static analyzers.

Static analysis only requires source code to reveal potential bugs. This makes it appealing for detecting system software bugs (e.g., bugs in device drivers) when executing the software is difficult or otherwise cumbersome. However, static analysis also has several disadvantages. First, due to overapproximations, the rate of false positives (false alarms) that are reported by the static analyzer can be high [20, 63]. According to [28], a typical concurrency bug can take weeks or even a whole month to debug. Considering this, false alarms produced by a static analyzer can be extremely costly, as they can waste weeks of the programmer's time in confirming whether they are genuine or not. Second, the static analyzer cannot guarantee that the program is free from (the class of bugs of interest) bugs if it does not report any warnings after analyzing the program. In other words, a static analyzer can have significant omission rates. On the plus side, most static analyzers can scale to large program sizes.

1.2.2 Dynamic Detection

Dynamic detection is an approach that concretely executes the program under test and detects potential concurrency bugs by monitoring executions. Eraser [83] and Helgrind [75] are two examples of dynamic data race detectors that dynamically track the set of locks held by shared objects during the concrete execution. These tools predict potential data races by inferring them based on the observation of one feasible execution path. In more detail, these dynamic race detectors keep track of the lock sets that are associated with shared objects among threads. If they find that the intersection of two lock sets that are held by different threads on the same shared object are empty, they produce a warning on potential data races. SVD [95] and AVIO [64] are dynamic detectors that focus on detecting atomicity violations by dynamically monitoring the memory access sequences of concurrent programs. They report an atomicity violation if a nonserializable memory access sequence is observed.

Dynamic detectors only require the binary executables to detect potential errors. This makes it possible for us to reveal bugs in software even if the source code of the

program is not available. Besides, due to concrete execution of the program, dynamic detectors do not need to have special precautions to handle the pointer aliasing problem that static analyzers need to handle. This makes it straightforward to implement dynamic detection algorithms, compared with static analysis. However, as the dynamic detectors only passively observe the executions and do not have control over the scheduling of the program, dynamic detection cannot cover all possible executions of the program. As a result, dynamic detection cannot soundly guarantee that the program is free of concurrency bugs if no concurrency errors are found.

1.2.3 Formal Static Verification

The goal of software verification is to assure that software fully satisfies the expected requirements. Compared with static analysis and dynamic detection, formal verification techniques can not only reveal bugs in the program, but also provide a guarantee on the correctness of the program with respect to the verified properties if the program is bug free.

Formal static verification aims to verify properties of software by examining only the source code of the program. Explicit model checking [12] and symbolic methods (e.g., symbolic model checking [6], predicate abstraction [32], etc.) are the most commonly used techniques in static software verifiers. In explicit model checking, static software verifiers typically extract models [81, 3, 39] out of the programs, and use explicit model checking techniques to explore the state space of the model to guarantee full coverage. With the symbolic approach [32], the program and the properties to be verified are converted into a set of constraints that are checked by constraint solvers [68, 70, 16, 17] to see whether the properties hold or not. Many algorithms, including partial order reduction [24], counterexample guided abstract refinement (CEGAR) [11], symmetry reduction [51], etc., are developed to improve the scalability of software verifiers.

The advantage of formal static verification is that it can guarantee that the verified properties hold for the program by examining only the source code. Unfortunately, due to the limitation of modeling and constraint solving, the application of static software verification to the implementation of concurrent programs is still limited. With explicit

model checking, modeling library functions and the runtime environment of programs is difficult as well as error-prone; the gap between modeling languages and programming languages is unbridgeably large in many cases. Predicate-abstraction-based model checkers, such as [13, 8, 37, 52], have demonstrated the abilities to prove properties of sequential programs. However, these model checkers have not been able to verify realistic concurrent programs because of the incapability of the constraint solvers.

Dynamic verification [25, 72] uses model checking techniques to systematically explore the state space of a concurrent program under specific test harnesses by concretely executing the program. Dynamic verification methods have proven promising for revealing errors in the implementation of real world concurrent programs [72]. Compared with static analysis, the advantage of dynamic verification is that it does not give out false warnings. Compared with dynamic detection, dynamic verification can provide full coverage of the possible interleavings of concurrent programs under specific inputs. Compared with formal static verification approaches, dynamic verification side steps the high complexity of model construction, state capture and constraint solving. Furthermore, by directly examining the implementation, dynamic verification avoids the potential inconsistency problem of the models and the implementations.

1.3 Problem Statement

Although dynamic verification is a promising technique for revealing concurrency bugs, its application in general concurrency programs is still limited due to the state explosion problem. Given a concurrent program that has n threads with k steps each, the number of possible interleavings of the program is $(nk)!/(k!)^n$. While the size of the program increases, the number of schedules that dynamic verification needs to examine increases exponentially. Efficient dynamic verification algorithms are essential to make dynamic verification be applicable to general concurrent programs.

1.4 Dissertation Statement

Program analysis and parallelism can help dynamic verification to scale up to handle general concurrent programs. In order to prove this dissertation statement, first we de-

veloped a series of novel algorithms and corresponding well-engineered implementations that have been publicly released. These implementations leverage program analysis and parallelism to improve the efficiency of dynamic verification.

1.4.1 Stateful dynamic partial order reduction [99]

In applying stateless model checking methods to realistic multithreaded programs, we find that stateless search methods are often ineffective in practice, even with dynamic partial order reduction (DPOR) enabled. To solve the inefficiency of stateless runtime model checking, we propose a novel and conservative light weight method for storing abstract states at runtime to help avoid redundant searches, and a stateful dynamic partial order reduction algorithm (SDPOR) that avoids a potential unsoundness when DPOR is naively applied in the context of stateful search. Our stateful runtime model checking approach combines light weight state recording with SDPOR, and strikes a good balance between state recording overheads, on one hand, and the elimination of redundant searches, on the other hand.

1.4.2 Property driven pruning for detecting data races [94]

The main idea of this algorithm is to use a lock-set-based analysis of observed executions to help prune the search space to be explored by the dynamic search. We assume that a stateless search algorithm is used to systematically execute the program in a depth first search order. If our conservative lock set analysis shows that a search subspace is race free, it can be pruned away by avoiding backtracks to certain states in the depth first search. Our new dynamic race detection algorithm is both sound and complete.

1.4.3 Symmetry discovery with dynamic program analysis [100]

Although symmetry reduction has been established to be an important technique for reducing the search space in model checking, its application in concurrent software verification is still limited, due to the difficulty of specifying symmetry in realistic software. We propose an algorithm for automatically discovering and applying transition symmetry in multithreaded programs during dynamic model checking. Our main idea

is using dynamic program analysis to identify a permutation of variables and labels of the program that entails syntactic equivalence among the *residual code of threads* and to check whether the local states of threads are equivalent under the permutation. The new transition symmetry discovery algorithm can bring substantial state space savings during dynamic verification of concurrent programs.

1.4.4 Distributed dynamic partial order reduction [98]

Although stateless model checking avoids the memory blow-up problem by not recording the search history, checking time becomes a major limiting factor. We propose distributed dynamic partial order reduction (DDPOR), which can speed up stateless model checking using computer clusters, and get the benefit of dynamic partial order reduction (DPOR).

1.4.5 Inspect: a Framework for Dynamic Verification

Second, we developed `Inspect`, a framework that combines program analysis and model checking to dynamically verify multithreaded C programs. Given a multithreaded C program and specific test harnesses, first `Inspect` uses thread escape analysis to get the possible global objects accesses, and other analysis algorithms (e.g., lock set analysis) to collect facts of the program. Based on the analysis results, `Inspect` instruments the program with code that is used to communicate with the external scheduler at selected places of the program (e.g., right before possible global accesses, immediately after a branch is taken, etc.). After this, the instrumented program is compiled into an executable. `Inspect` repeatedly executes the program under given test harnesses and uses the external scheduler to guide the program to systematically explore different interleavings. `Inspect` is able to reveal concurrency related errors (data races, deadlocks, local assertion violation, etc.).

We implemented the efficient dynamic verification algorithms in `Inspect` and applied `Inspect` to check realistic multithreaded applications. Our experiments confirm the effectiveness of our approach.

1.5 Contributions

In this dissertation, we focus on improving the efficiency of dynamic verification of multithreaded programs, and make the following contributions:

In stateful dynamic partial order reduction, we develop a novel and conservative light weight method for storing abstract states at runtime to help avoid redundant searches. We also develop a stateful dynamic partial order reduction algorithm that avoids a potential unsoundness when dynamic partial order reduction is naively applied in the context of stateful search.

In the algorithm of symmetry discovery using dynamic analysis, we develop a novel symmetry discovery algorithm that uses dynamic program analysis to make automatic symmetry discovery possible.

In the algorithm of property driven pruning for race detection, we develop a new lock set analysis of the observed execution trace for checking whether the associated search subspace is race free. Besides, we show how to have property driven pruning in a backtracking algorithm using depth first search.

In distributed dynamic partial order reduction, our key contributions are, a practical algorithm for distributed dynamic partial order reduction, and the innovations that helped distributed `inspect` attain nearly linear (with respect to the number of CPUs) speedup on realistic examples.

Furthermore, we build `Inspect`, which is a dynamic verification framework for multithreaded C programs. To the best of our knowledge, `Inspect` is the first framework that combines program analysis and model checking to achieve efficient dynamic verification.

1.6 Outline

The remainder of this dissertation is organized as follows. Chapter 2 describes the background knowledge and the formal notations we are going to use in this dissertation. Chapter 3 presents `Inspect`, the framework we designed for dynamic verification of multithreaded C programs. Chapter 4 describes the stateful dynamic partial order reduction algorithm. Chapter 6 explains the the distributed dynamic partial order reduction

algorithms. Chapter 7 presents the symmetry discovery algorithm. Chapter 5 explains property driven pruning of the search space. Chapter 8 concludes and discusses future work.

CHAPTER 2

PRELIMINARIES

In this chapter, we introduce the background knowledge and the formal notations that we are going to use in the rest of this dissertation.

2.1 Concurrent Programs

We consider a multithreaded program with a fixed number of sequential threads as a state transition system. We use $Tid = \{1, \ldots, n\}$ to denote the set of thread identities. Threads communicate with each other via *global* objects that are visible to all threads. The operations on global objects are called *visible operations*, whereas thread local variable updates are *invisible operations*. A *state* of a multithreaded program consists of the global state, and the local state of each thread. The local state of a thread is a stack that is composed of frames. We assign each frame of the stack a unique identity. Hence, the local state can be viewed as a map from $Fid \subset \mathbb{N}$ to frames. A frame is a map from thread-local variables to the concrete values. Formally, the total system state (S), the local states of all threads ($Locals$), the local state of each thread ($Local$), and the frame of a local state ($Frame$) are defined as follows:

$$\begin{aligned} S &\subseteq Global \times Locals \\ Locals &= Tid \rightarrow Local \\ Local &= Fid \mapsto Frame \\ Frame &= \mathcal{V} \mapsto \mathbb{N} \end{aligned}$$

In the rest of this dissertation, we will assume that the state spaces we consider do not contain any cycles, and focus on detecting deadlocks and safety-property violations such as data races and assertion violations. Note that in model checking of software implementations, acyclic state spaces are quite common. The execution of most software applications eventually terminates due to the inputs or the run/test time bound.

For convenience, we use $g(s)$ to denote the global state in a state s, and use $l_\tau(s)$ to denote the local state of thread τ in s. We write $s[\tau := l']$ to denote a state that is identical to s except that the local state of thread τ is l'. We use $s[g := g'; \tau := l']$ to denote a state that is the same as the state s except that the global state is g' and the local state of thread τ is l'.

A transition advances the program from one state to a subsequent state by performing one visible operation of a thread, followed by a finite sequence of invisible operations of the same thread, ending just before the next visible operation of that thread. The transition t_τ of thread τ can be defined as $t_\tau : Global \times Local \rightarrow Global \times Local$.

Let $\mathcal{T}$ denote the set of all transitions of a program. A transition $t_\tau \in \mathcal{T}$ is enabled in a state s if $t_\tau(g(s), l_\tau(s))$ is defined. If t is enabled in s and $t(g, l) = \langle g', l' \rangle$, then we say the execution of t from s produces a successor state $s' = s[g := g'; \tau := l']$, written $s \xrightarrow{t} s'$.

The behavior of a multithreaded program P is given by a transition system $M = (S, R, s_0)$, where $R \subseteq S \times S$ is the transition relation, and s_0 is the initial state. $(s, s') \in R$ if and only if $\exists t \in \mathcal{T} : s \xrightarrow{t} s'$.

2.2 Partial Order Reduction

We briefly introduce some basic principles of partial-order reduction methods. The basic observation exploited by these techniques is that a concurrent system typically contains many paths that correspond simply to different execution orders of the same uninteracting transitions. Two transitions are *independent* if and only if they can neither disable nor enable each other, and swapping their order of execution does not change the combined effect [56].

Definition 2.1 $R \subseteq \mathcal{T} \times \mathcal{T}$ is an independence relation if and only if for each $\langle t_1, t_2 \rangle \in R$ the following two properties hold for all $s \in S$: (i) if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s if and only if t_2 is enabled in s'; and (ii) if t_1, t_2 are enabled in s, there is a unique state s' such that $s \overset{t_1 t_2}{\Rightarrow} s'$ and $s \overset{t_2 t_1}{\Rightarrow} s'$.

This definition characterizes the properties of possible "valid" dependency relations for the transitions of a given system. In practice, it is possible to give easily-checkable

conditions that are sufficient for transitions to be independent [24]. Dependency can arise between transitions of different processes that perform visible operations on the same shared object. For instance, two acquire operations on the same lock are dependent, and so are two write operations on the same variable; in contrast, two read operations on the same variable are independent, and so are two write or compare-and-swap operations on different variables.

Traditional partial-order algorithms operate as classical state-space searches except that, at each state s reached during the search, they compute a subset T of the set of transitions enabled at s, and explore only the transitions in T. Such a search is called a selective search and may explore only a subset of the concurrent system. Two main techniques for computing such sets T have been proposed in the literature: the persistent/ample/stubborn set and sleep set techniques [24]. The first technique actually corresponds to a family of algorithms [29, 30, 79, 90], which can be shown to compute persistent sets [24]. Intuitively, a subset T of the set of transitions enabled in a state s of a concurrent system is called *persistent in* s if whatever one does from s, while remaining outside of T, does not interact with T.

Definition 2.2 A set $T \subseteq \mathcal{T}$ of transitions enabled in a state s is persistent if and only if for all nonempty sequences of transitions $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$ from s in a concurrent system and including only transitions $t_i \notin T$, $1 \leq i \leq n$, t_n is independent with all the transitions in T.

A selective search of the concurrent system using persistent sets is guaranteed to visit all the deadlocks in the concurrent system [25]. Moreover, if the concurrent system is acyclic, a selective search using persistent sets is also guaranteed to visit all the reachable local states of every process in the system [25], and hence, can be used to detect violations of any property reducible to local state reachability, including violations of local assertions and of safety properties.

2.3 Dynamic Partial Order Reduction

Systematic state space exploration can be performed in a stateless fashion, by dynamically executing the program in a depth-first search order. Instead of enumerating

reachable states of the model as in classic model checkers, dynamic model checking focuses on exhaustively exploring the feasible executions. This is made possible by controlling and observing the execution of visible operations of all threads. In particular, the entire program is executed under the control of a special *scheduler*, which gives permission to, and observes the results of, all visible operations. Since the scheduler has full control of every context switch, systematic exploration becomes possible. In this context, backtracking or the roll-back of a partially executed sequence is implemented by restarting the program afresh under a different thread schedule.

Given the set of enabled transitions from a state s, partial order reduction algorithms attempt to explore only a (proper) subset of transitions that are enabled at s, and at the same time guarantee that the properties of interest will be preserved. Such a subset is called a *persistent set*. Static partial order reduction algorithms compute the persistent set of a state immediately after reaching it. As for multithreaded programs, persistent sets computed statically can be excessively large because of the limitations of static analysis. For instance, if two transitions leading out of s access an array `a[]` by indexing it at locations captured by expressions `e1` and `e2` (i.e., `a[e1]` and `a[e2]`), a static analyzer may not be able to decide whether `e1=e2` (and hence, whether the transitions are dependent or not).

In DPOR, given a state s, the persistent set of s is not computed immediately after reaching s. Instead, DPOR populates the persistent set of s while searching under s according to depth-first search (DFS). Figure 2.1 shows the DPOR algorithm. In DPOR, the scheduler maintains a *search stack* S of global states. Each state s in the stack is associated with a set $s.enabled$ of enabled transitions, a set $s.done$ of thread identities, and a *backtracking set* $s.backtrack$. In more detail, $s.done$ denotes the set of threads that have been examined at s. $s.backtrack$ refers to the sets of threads that need to be explored from s. The procedure UPDATEBACKTRACKSETS(S, t) is used to *dynamically* compute the persistent sets [24] of states.

```
1:  Initially: S.push(s_0); DPOR(S)

2:  DPOR(S) {
3:    let s = S.top;
4:    for each t ∈ s.enabled, UPDATEBACKTRACKSET(S, t);
5:    if (∃τ ∈ Tid : ∃t ∈ s.enabled : tid(t) = τ) {
6:      s.backtrack ← {τ};
7:      s.done ← ∅;
8:      while (∃q ∈ s.backtrack \ s.done)
9:        s.done ← s.done ∪ {q};
10:       s.backtrack ← s.backtrack \ {q};
11:       let t ∈ s.enabled such that tid(t) = q;
12:       let s' be a state such that s -t-> s';
13:       S.push(s');
14:       DPOR(S);
15:       S.pop();
16:     }
17:   }
18: }

19: UPDATEBACKTRACKSETS(S, t){
20:   let T be the sequence of transitions associated with S;
21:   let t_d be the latest transition in T that is dependent and may be co-enabled with t;
22:   if (t_d = null) return;
23:   let s_d be the state in S from which t_d is executed;
24:   let E be {q ∈ s_d.enabled | tid(q) = tid(t), or q in T, q was after t_d and there is a
        happens-before relation for (q, t). }
25:   if (E ≠ ∅)
26:     choose any q in E, add tid(q) to s_d.backtrack;
27:   else
28:     s_d.backtrack ← s_d.backtrack ∪ {tid(q) | q ∈ s_d.enabled};
29: }
```

Figure 2.1. Dynamic partial-order reduction

2.3.1 An Example on DPOR

Here we give out an example to illustrate the DPOR algorithm. Consider two threads t_1 and t_2 that share two variables x and y:

$$\begin{array}{ll} t_1: & x = 1; \quad x = 2 \\ t_2: & y = 1; \quad x = 3 \end{array}$$

Assume the first random execution of the program is:

$$t_1 : x = 1; \;\; t_1 : x = 2; \;\; t_2 : y = 1; \;\; t_2 : x = 3;$$

Before executing the last transition $t_2 : x = 3$, DPOR will add a backtracking point for thread t_2 before the last transition of t_1. With sleep sets enabled, DPOR will force the following execution (the sleeps sets are shown in the parentheses) :

$$(\varnothing)t_1 : x = 1; \;\; (\{t_1 : x = 2\})t_2 : y = 1; \;\; (\{t_1 : x = 2\})t_2 : x = 3; \;\; (\varnothing)t_1 : x = 2;$$

which in turn forces

$$(\{t_1 : x = 1\})t_2 : y = 1; \;\; (\{t_1 : x = 1\})t_2 : x = 3; \;\; (\varnothing)t_1 : x = 1; \;\; (\varnothing)t_1 : x = 2;$$

2.4 Summary

In this chapter, we present the formal notations that we are going to use for the rest of the dissertation. We also have a brief introduction on partial order reduction, and in particular on dynamic partial order reduction. Interested readers can find more information on partial order reduction algorithm in [24] and can find more information on dynamic partial order reduction in [21].

CHAPTER 3

INSPECT: A DYNAMIC VERIFICATION FRAMEWORK

In this chapter, we present `Inspect`, a framework that we designed for dynamic verification of multithreaded C program. `Inspect` can systematically explore all possible interleavings of a multithreaded C program under specific test harnesses, and guarantee the exposure of concurrency errors.

3.1 Overview

An overview of `Inspect` is shown in Figure 3.1. It consists of four parts: (i) a program analyzer that analyzes the program for possible global accesses and other information of the program, (ii) a program instrumentor that can instrument the program at the source code level with codes that are used to communicate with the scheduler, etc., (iii) a thread library wrapper that helps intercept the thread library calls, and (iv) an external scheduler that schedules the interleaved executions of the threads.

Given a multithreaded program, `Inspect` first instruments the program with code that is used to communicate with the scheduler. Thereafter, it compiles the program into an executable and runs the executable repeatedly under the control of the scheduler until all relevant interleavings among the threads are explored. Before performing any operation that might have side effects on other threads, the instrumented program sends a request to the scheduler. The scheduler can block the requester by postponing a reply. We use blocking sockets as communication channels between the threads and the scheduler. As the number of possible interleavings grows exponentially with the size of the program, we implemented the dynamic partial order reduction (DPOR) algorithm to reduce the search space. Note that the method we present in this paper can also be applied to C++

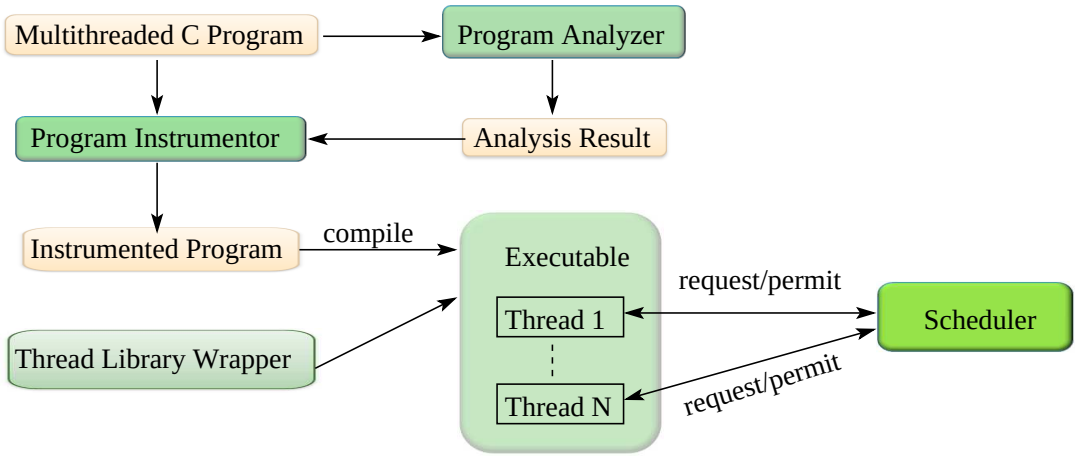


Figure 3.1. `Inspect`'s workflow

programs. However, due to the lack of a C++ front end for source code transformation, we need manual instrumentation for C++ programs.

3.2 An Example

In this section, we show how `Inspect` works with a simple example that captures a common concurrent scenario in database systems. Suppose that a shared database supports two distinct classes of operations, A and B. Let the semantics of the two types of operations be such that multiple operations of the same class can run concurrently, but operations belonging to different classes cannot be run concurrently. Figure 3.2 is a buggy implementation that can lead to a deadlock. Global variables `A_count` and `B_count` capture the number of threads that are performing operations A and B, respectively. Here, the variable `lock` is used for mutual exclusion between threads, and `mutex` is used to guarantee the atomicity of updating the counters, `a_count` and `b_count`.

Conventional testing might miss the potential deadlock hidden in the code as it runs with random scheduling. In general, it is difficult to get a specific scheduling that leads to the error. `Inspect` first instruments the program with code that can take the control of scheduling away from the runtime system. Then it compiles the instrumented code into

shared variables among threads:

```
pthread_mutex_t mutex, lock;
int A_count = 0;
int B_count = 0;
```

class A operation:

```
 1:   pthread_mutex_lock(&mutex);
 2:   A_count++;
 3:   if (A_count == 1) {
 4:      pthred_mutex_lock(&lock);
 5:   }
 6:   pthread_mutex_unlock(&mutex);
 7:   performing class A operation;
 8:   pthread_mutex_lock(&mutex);
 9:   A_count--;
10:   if (A_count == 0){
11:      pthread_mutex_unlock(&lock);
12:   }
13:   pthread_mutex_unlock(&mutex);
```

class B operation:

```
 1:   pthread_mutex_lock(&mutex);
 2:   B_count++;
 3:   if (B_count == 1){
 4:     pthred_mutex_lock(&lock);
 5:   }
 6:   pthread_mutex_unlock(&mutex);
 7:   performing class B operation;
 8:   pthread_mutex_lock(&mutex);
 9:   B_count--;
10:   if (B_count == 0){
11:     pthread_mutex_unlock(&lock);
12:   }
13:   pthread_mutex_unlock(&mutex);
```

Figure 3.2. An example of concurrent operations in a shared database

an executable. With the help of a centralized scheduler, `Inspect` can systematically explore relevant interleavings. As for this example, `Inspect` will report that $a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_4 \rightarrow a_6 \rightarrow b_1 \rightarrow b_2$ is an interleaving that leads to a deadlock (here a_i and b_i stand for line i of class A and class B threads).

3.3 Inspect

3.3.1 Identifying Threads and Shared Objects Across Executions

When `Inspect` reexecutes the program under test, the runtime environment may change across reexecutions. For instance, the threads may not be allocated to the same identity by the operating system. Also, dynamically-created shared objects (e.g., mallocs) may not reside in the same physical memory address in different runs. Handling these practical issues is essential for the success of runtime model checking.

We handle these issues in `Inspect` aided by a few (practically realistic) assumptions. First, we assume that given the same external input, multiple threads in a program are always created in the same order across different runs. Banking on this fact, we can identify threads (which may be allocated different thread IDs in various reexecutions) across two different runs by examining the sequence of thread creations. We let each thread register itself to the scheduler. If the threads are created in the same sequential order in different runs, threads will be registered in the same order. In this way, we can easily assign the same ID for the same thread across multiple runs.

Identifying the shared objects is done in the same manner: if two runs of the program have the same visible operation sequence, the shared objects will also be created with malloc, etc., in the same sequence. As a result, shared objects across multiple runs can be identified, even though the actual objects may be getting created at different memory addresses in different reexecutions.

3.3.2 Instrumenting the Code

We describe how `Inspect` works for a simple C-like language shown in Figure 3.3. A program is composed of a *main* function and a set of threads. Each thread is a sequence of statements. The condition expression in the *if* statement is simplified as a variable.

$$
\begin{array}{rcl}
P & ::= & T^* \\
T & ::= & main \mid thread \\
thread & ::= & Stmt^* \\
main & ::= & Stmt^* \\
Stmt & ::= & [l:]s \\
s & ::= & lhs \leftarrow e \mid \texttt{if } lhs \texttt{ then goto } l' \\
 & & \mid \texttt{create} \mid \texttt{join} \mid \texttt{exit} \\
 & & \mid \texttt{lock} \mid \texttt{unlock} \\
 & & \mid lhs \leftarrow \texttt{malloc} \\
lhs & ::= & v \mid \&v \mid *v \\
e & ::= & lhs \mid lhs \ \ op \ \ lhs \\
 & & where\ op \in \{+, -, *, /, \%, <, >, \leq, \geq, \neq, =, ...\}
\end{array}
$$

Figure 3.3. Syntax of a simple language that is similar to C

`Inspect` uses CIL [1] to convert more complex statements into this simplified form by introducing temporary variables

Figure 3.4 shows how `Inspect` instruments a multithreaded program based on the syntax shown in Figure 3.3. The instrumentation does the following things: (i) replace the function calls to the thread library routines with function calls to `Inspect` wrapper functions, (ii) add extra code before thread starting/exiting points to notify the scheduler, (iii) add object registration code at the beginning of *main* function for global objects, (iv) object registration code is also needed after *malloc* operations which allocate new objects which are shared among threads, and (v) for each read/write access on data objects that are shared among threads, `Inspect` intercepts the operations by adding a wrapper around it.

To realize step (v), we need to know whether an update to a data object is a visible operation or not. Doing this exactly is undecidable, as it amounts to context sensitive language reachability. We conservatively overapproximate this step by performing an interprocedural flow-sensitive alias analysis on the program. Based on the results of the alias analysis, we have a may-escape analysis [82] on the program to find all operations on shared objects that may "escape" the thread scope. As the may-escape analysis is an overapproximation of all-possible shared variables among threads, our instrumentation is

Before instrumentation	After instrumentation
`create`	`inspect_create()`
`join`	`inspect_join()`
`exit`	`inspect_exit()`
`lock`	`inspect_lock()`
`unlock`	`inspect_unlock()`

thread	`thread_begin`
	thread
	`thread_end`

main	`global_object_registration()`
	`thread_begin`
	main
	`thread_end`

// allocate memory for a shared object lhs ← `malloc`	lhs ← `malloc()` `object_registration`(&lhs)
// *if* lhs *is a shared object* lhs ← v	`write_shared`(&lhs, v)
// *if* v *is a shared object* lhs ← v;	lhs ← `read_shared`(&v)

Figure 3.4. `Inspect`'s instrumentation

safe for intercepting all the visible operations in the concrete execution. Figure 3.5 shows the instrumented code of class A thread in Figure 3.2. It is the engineering of these details that sets `Inspect` apart from previous prototype implementations of DPOR, and for the first time makes it possible to assess the impact of a well-engineered DPOR algorithm in the setting of realistic multithreaded C programs.

3.3.3 Avoiding Redundant Backtracking

One optimization we made on DPOR algorithm is on avoiding redundant backtracking. State backtracking is an expensive operation for runtime model checkers as the model checker needs to restart the program, and replay the program from the initial state until the backtrack point. Obviously, we want to avoid backtracking as much as possible to improve efficiency. Line 4 of Figure 2.1 is the place in DPOR where a backtrack point is identified. It treats t_d, which is dependent and may be co-enabled with t_n as a backtrack

```
void *thread_A(void *arg )
{
  void *__retres2 ;
  int __cil_tmp3 ;
  int __cil_tmp4 ;
  int  __cil_tmp5 ;
  int  __cil_tmp6 ;
  int  __cil_tmp7 ;
  int  __cil_tmp8 ;
  int  __cil_tmp9 ;
  int  __cil_tmp10 ;

  inspect_thread_start("thread_A");
  inspect_mutex_lock(& mutex);
  __cil_tmp7 = read_shared_0(& A_count);
  __cil_tmp3 = __cil_tmp7 + 1;
  write_shared_1(& A_count, __cil_tmp3);
  __cil_tmp8 = read_shared_2(& A_count);
  __cil_tmp4 = __cil_tmp8 == 1;
  if (__cil_tmp4) {
    inspect_mutex_lock(& lock);
  }
  inspect_mutex_unlock(& mutex);
  inspect_mutex_lock(& mutex);
  __cil_tmp9 = read_shared_3(& A_count);
  __cil_tmp5 = __cil_tmp9 - 1;
  write_shared_4(& A_count, __cil_tmp5);
  __cil_tmp10 = read_shared_5(& A_count);
  __cil_tmp6 = __cil_tmp10 == 0;
  if (__cil_tmp6) {
    inspect_mutex_unlock(& lock);
  }
  inspect_mutex_unlock(& mutex);
  __retres2 = (void *)0;
  inspect_thread_end();
  return (__retres2);
}
```

Figure 3.5. Instrumented code for the class A thread of Figure 3.2

point. However, *if two transitions that may be co-enabled are never co-enabled, we may end up exploring redundant backtrackings, and reduce the efficiency of DPOR.*

We use locksets to avoid exploring transition pairs that cannot be co-enabled. Each transition t is associated with the set of locks that are held by the thread that executes t. With these locks, we compute the may be co-enabled relation more precisely by testing whether the intersection of the locksets held by two threads is empty or not.

3.4 Implementation

`Inspect` is designed in a client/server style. The server side is the scheduler that controls the program's execution. The client side is linked with the program under test to communicate with the scheduler. The client side includes a wrapper for the pthread library, and facilities for communication with the scheduler.

Figure 3.6 shows the details of the implementation. The scheduler uses a message buffer to receive messages that come from different threads. By picking up the messages and sending permissions to the correspondent thread, the scheduler can guide the program under test through different interleavings.

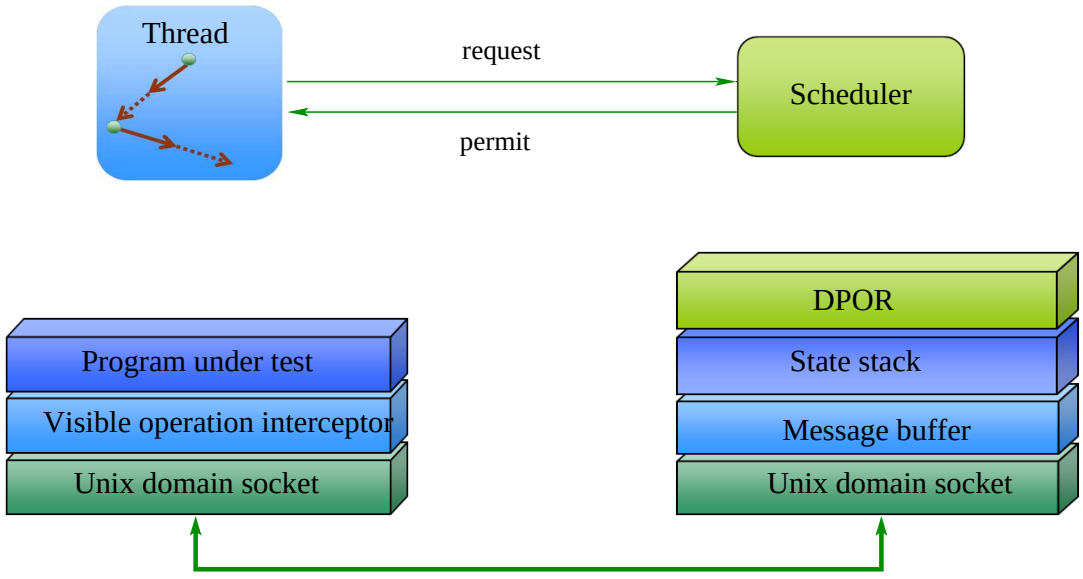


Figure 3.6. The internal design of `Inspect`

We have the scheduler and the program under test communicate using Unix domain sockets. Comparing with Internet domain sockets, Unix domain sockets are more efficient as they do not have the protocol processing overhead, such as adding or removing the network headers, calculating the check sums, sending the acknowledgments, etc. Besides, the Unix domain datagram service is reliable. Messages will neither get lost nor be delivered out of order. we use CIL [74] to implement the program analysis and transformation part. An initial version of `Inspect` is available at [47].

3.5 Related Work

CMC [71] verifies C/C++ programs by using a user-model Linux as a virtual machine. CMC captures the virtual machine's state as the state of a program. Unfortunately, CMC is not fully automated. As CMC takes the whole kernel plus the user space as the state, it is not convenient for CMC to adapt the dynamic partial order reduction method.

ConTest [18] debugs multithreaded programs by injecting context switching code to randomly choose the threads to be executed. As randomness does not guarantee all interleavings will be explored for a certain input, it is possible that ConTest can miss bugs. Given an input, if there exists some interleavings that can lead to error, `Inspect` can guarantee that the error will be caught.

Lei et al. [60] designed RichTest, which used reachability testing to detect data races in concurrent programs. Reachability testing views an execution of a concurrent program as a partially-ordered synchronization sequence. However, RichTest does not address the practical issues such as how to control the scheduling of multithreaded C program which `Inspect` solves.

Helmstetter et al. [36] show how to generate scheduling based on dynamic partial order reduction. It solves the problem of exploring different interleavings in the context of SystemC models. Inspect handles general multithreaded C applications, using a different method to handle the practical scheduling problem.

CHESS [72] is a dynamic model checker for testing multithreaded programs. CHESS assumes that a program is data-race free, every access to shared objects is protected by some locks. `Inspect` does not require this assumption. CHESS takes control of the

scheduling by intercepting only library calls. Inspect intercepts not only library calls, but also visible operations on data objects by performing source code transformation, as discussed in Section 3.3.2. As a result of this, CHESS captures the happen-before relation of the events, and reports a race when it observes two events between which there is no happen-before relation. Different from CHESS, `Inspect` reports a race if and only if a real racing scenario is observed.

3.6 Summary

In this chapter, we present the design and implementation of `Inspect`, which is a dynamic verification framework for multithreaded C programs. `Inspect` combines program analysis and dynamic model checking in a unique way to make it possible for us to do more efficient dynamic verification.

CHAPTER 4

STATEFUL DYNAMIC PARTIAL ORDER REDUCTION

Dynamic model checking [25, 72] is a promising method for bug detection. As model building, extraction, and model maintenance are expensive to carry out for thread programs written in practice, we believe in the importance of developing efficient runtime checking methods, as pioneered in [25]. However, even when running under specific inputs, the number of interleavings of a concurrent program can grow astronomically due to their internal concurrency.

Much of the interleaving explosion that occurs in practice during stateless runtime model checking can be attributed to redundant searches from already visited states. The example in Figure 4.1 illustrates this problem. This program has two threads that, in their own `atomic` blocks that are nested within loops, write to a shared variable `d`. Stateless search methods cannot handle this example even with the help of dynamic

```
    const int N = 64;
    int d = 0;

     thread1:                    thread2:
    local int i = 0;                 local int j = 0;
L0: while (i < N){           M0:   while (j < N){
L1:   atomic{                M1:     atomic{
        d = d + i;           M2:       d = d - j;
        assert(d%5!=4)                 assert(d%5!=4);
      }                              }
L2:   i = i + 5;             M3:     j = j + 2;
L3: }                        M4:   }
```

Figure 4.1. A simple example for illustrating the idea

partial order reduction (DPOR) [21]. This is because (i) the number of interleavings grows exponentially with respect to the number of loop iterations, and (ii) working under stateless DPOR, at any reached state where both threads are enabled, there exists no nontrivial persistent set.

However, with stateless search, many states of this program are revisited multiple times via different interleavings. For example, given `thread1` at `L1` and `thread2` at `M1`, whether `thread1` executing `L1, L2` followed by `thread2` executing `M1,M2`, or vice versa, the program reaches the same state. Figure 4.2 illustrates this, where `T1` represents `thread1` and `T2` represents `thread2`. The dotted states are the intermediate states attained after executing the visible operation of a transition; this detail illustrates a convention introduced in Section 2.1. Failing to detect visited states makes the stateless search methods repeatedly explore visited state spaces, and results in very low efficiency.

Although one straightforward solution that avoids redundant searches involves the use of visited states maintained in a hash table, this method is complicated owing to the difficulty of capturing the states of realistic multithreaded program at runtime. This is especially true for programs written in program languages such as C/C++. Although there have been model checkers such as CMC [71] and Java PathFinder [91] that have attempted such program state capture, these approaches are quite heavy-weight. For example, if we take CMC's approach, we need to capture the state of the kernel space plus the user space. Alternately, if we follow Java PathFinder's approach, we will have to build a virtual machine for C/C++ programs. Is there a light weight approach to recording

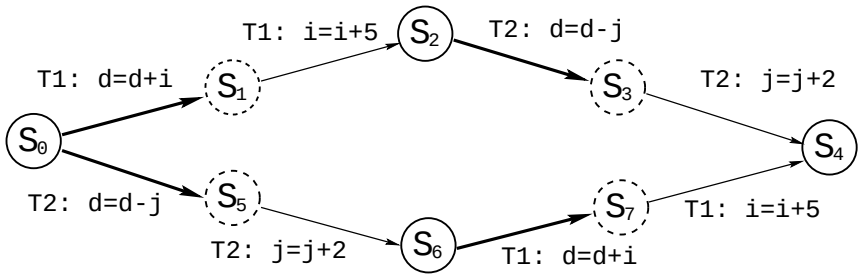


Figure 4.2. Two different executions of a program lead to the same state

the states of concurrent programs at runtime? If we have such an approach, how do we combine it with partial order reduction techniques soundly?

In this chapter, we present a novel light-weight scheme for capturing the *local* states of threads. We observe that whereas capturing the entire state of a realistic program at runtime is difficult and expensive, capturing the *changes* between two successive local states of a thread can be inexpensive and easier. Based on this observation, we abstract local states of threads with identities, and try to discover the same local state of a thread among different executions by tracking the changes (i.e., "deltas") between successive local states of threads. Whereas an actual total system state of a thread program with N threads would be a tuple $(g, (l_1, \ldots, l_N), (p_1, \ldots, p_N))$ where g is the global state, l_k are the actual thread local states and p_k are the actual thread PCs, an *abstract* state would be $(g, (i_1, \ldots, i_N), (p_1, \ldots, p_N))$ where i_k are *IDs* we assign for thread local states. These IDs are computed in a conservative way based on the sequence of deltas that each thread undergoes, as explained in Section 4.1.

We present a stateful dynamic partial order reduction (SDPOR) algorithm, which combines our light-weight runtime state capturing approach with dynamic partial order reduction. By introducing states in dynamic partial order reduction, an obvious soundness problem is not updating the backtrack set along a new path that revisits a state. To solve this problem efficiently, we dynamically construct a *visible operation dependency graph* while performing the search. When a visited state is encountered, we compute the summary of the visited subspace using the visible operation dependency graph. With the summary, we conservatively update the backtrack sets of states and guarantee the soundness of our approach.

We have implemented SDPOR within our runtime model checker `Inspect` [47], and evaluated our approach on a set of multithreaded C benchmarks. The experiments show that SDPOR is much more effective than the stateless DPOR.

The rest of the chapter is organized as follows. In Section 4.1, we describe how local states can be captured in a light-weight and conservative manner. Section 4.2 presents how the DPOR algorithm can be adapted, with the stateful search. Sections 4.3 and 4.4

then present the implementation details and the experimental results, respectively. In the end, we discuss related work and summarize this chapter.

4.1 Capturing Local States of Threads

Although the local states of threads are not easy to capture precisely at runtime, we observe that in many cases, the changes δ between successive local states are easy to capture. For example, due to the correlations among global objects [80], it is commonly the case that there exist sequences of transitions in which each transition has only the visible operation component, with the invisible operation component being absent. In this case, the local states of threads do not change. It is also common that the changes of local states only involve several local objects and are easy to capture. As an example, in the program of Figure 4.1, the local state change of `thread1` between two successive executions of the atomic statement labeled `L1` only involves the local object `i`. Likewise, the local state change of `thread2` between two successive executions of the atomic block at `M1` only involves the local object `j`. This motivates us to capture the local states of threads by tracking the changes among local states.

We now detail our algorithm for capturing local states of threads at runtime, in the context of a depth-first search of the state space of the threads. The key idea of the algorithm is to represent each local state of a thread with an abstract identity, and to *link* these IDs by tracking changes between successive local states of threads. This scheme helps conservatively determine whether local states of threads are repeating across different executions.

Let $LocalId$ denote the set of local state IDs (natural numbers). We define the *abstract state* of a multithreaded program formally as follows:

$$\begin{array}{rcl} S_a & \subseteq & Global \times Locals_a \times PCs \\ Locals_a & = & Tid \rightarrow LocalId \\ LocalId & \subset & \mathbb{N} \end{array}$$

With the local state IDs, a multithreaded program can be represented as a transition system $M_a = (S_s, s_{0_a}, \Gamma_a)$, where s_{0_a} is the initial state of the program, and $\Gamma_a \subseteq S_a \times S_a$ is the transition relation. Note that because of our conservative state maintenance

scheme which we present later in this section, there could be more than one abstract state associated with a real state.

Let s_a be an abstract state in S_a. When the context is clear, we still use $g(s_a) \in Global$ to denote the global state of s_a, and use $ls(s_a) \in Locals_a$ to denote the local states identities. We use $lid_\tau(s_a) \in LocalId$ to denote the assigned local state identity of thread τ. For $ls_a \in Locals_a$, we write $ls_a[\tau := x]$ to denote that the map that is identical to ls_a except that it maps the thread τ to the local state identity x.

As the state of global objects are in general easy to capture, we do not abstract the states of global objects. Let $s_a \in S_a$ be an abstract state and s be its corresponding state in S. We have $g(s_a) = g(s)$. Similarly, we also have $s_a.PCs = s.PCs$.

Let $s, s' \in S$ be two states, and t be a transition such that $s \xrightarrow{t} s'$. Let $\tau = tid(t)$. We define the changes of the local state of thread τ between s and s' as $\delta_\tau = l_\tau(s') \setminus l_\tau(s)$. We use δ_ε to represent that the local state does not change for a thread. That is, for the thread τ in the above, $l_\tau(s') = l_\tau(s)$. Also, we write $\delta_\perp$ to denote that the local state changes are unknown. $\delta_\perp$ is used when it is hard to capture the local state changes, e.g., when the transition t_l involves calls to library routines, etc. We use Δ to denote the set of all possible local state changes (deltas) for all threads in the program.

In order to detect that the same local state of thread is appearing in different executions of the multithreaded program, we maintain a *local state hash table* for each thread of the program. The local state hash table records the IDs of the local states that have been visited, as well as the changes between two successive local states. In more detail, for each thread τ, we have a local state hash table L_τ to store the IDs of the visited local states of τ. $L_\tau : LocalId \times \Delta \rightarrow LocalId$ is a mapping from a local state IDs plus the change to a local state, to a potentially new local state ID. However, if L_τ already contains the domain point, then the local state ID already in the hash table is returned. We use L to denote the set of local state hash tables for all threads.

Our basic search algorithm with abstract state recording is presented in Figures 4.3 and 4.4. The final SDPOR algorithm in Section 4.2 will build on this algorithm. Figure 4.3 shows DFS, a recursive procedure for depth-first search of the state space. DFS calls NEXTLOCAL of Figure 4.4 to compute the local state IDs of a thread. The main data

```
1: Initially: H is empty; ∀L_τ ∈ L : L_τ is empty; DFS(H, L, s_0, s_{0_a});

2: DFS(H, L, s, s_a) {
3:   if (s_a ∈ H) return;
4:   enter s_a in H;
5:   for each transition t that is enabled in s {
6:     let s' ∈ S such that s -t-> s';
7:     let τ = tid(t), δ_τ = l_τ(s') \ l_τ(s);
8:     let x = NEXTLOCAL(L_τ, lid_τ(s_a), δ_τ);
9:     let s'_a ∈ S_a s.t. g(s'_a) = g(s') ∧ ls(s'_a) = ls(s_a)[τ := x] ∧ s'_a.PCs = s'.PCs;
10:    DFS(H, L, s', s'_a);
11:  }
12: }
```

Figure 4.3. Depth-first search with a light-weight state capturing scheme

structures used are a hash table H to store all program states $s \in S_a$ that have already been visited during the search, and for each thread τ, a local state hash table L_τ to store the identities of the visited local states of τ.

DFS of Figure 4.3 has four parameters: the abstract state hash table H, the local state hash tables L, the current state s, and finally s_a, which is the abstract state of s. Starting from the initial state, DFS recursively explores the successor states of all states encountered during the search, provided that the correspondent abstract state is not in the hash table. For each visited state, DFS stores the correspondent abstract state in the hash table H. Each time we reach a state s' by executing a transition t which is enabled in a state s, we will compute the abstract state of s' (line 7-9 of Figure 4.3), and recursively call DFS to explore the next level of the state space.

Figure 4.4 shows the algorithm for computing the local state identity of a thread. In the procedure NEXTLOCAL, we consider four possible cases: (i) if the local state change is difficult to capture precisely, we simply return a new local state ID x, (ii) if the local state does not change (i.e., $\delta_\tau = \delta_\varepsilon$), the same ID is returned, (iii) if the hash table L_τ already has an entry for $(i, \delta_\tau) \rightarrow y$, then we return y as the ID, and (iv) otherwise, we return a new local state ID x, and at the same time add an entry $\langle (i, \delta_\tau) \rightarrow x \rangle$ to L_τ.

Now with the procedures DFS and NEXTLOCAL, we have Theorem 4.1.

```
1: NEXTLOCAL(L_τ, i, δ_τ) {
2:    let x ∈ LocalId be a unique new local state identity;
3:    if (δ_t = δ_⊥) return x;
4:    if (δ_τ = δ_ε) return  i;
5:    if (∃y : ⟨(i, δ_τ) → y⟩ ∈ L_τ) return y;
6:    add ⟨(i, δ_τ) → x⟩ to L_τ;
7:    return x;
8: }
```

Figure 4.4. Computing the local state identity of a thread

Theorem 4.1 Let $M = (S, s_0, \Gamma)$ be a multithreaded program. In a depth-first search on S following the algorithm of Figure 4.3, let $s, s' \in S$ be states that can be reached from s_0, and let $s_a, s'_a \in S_a$ be the abstract states corresponding to s and s'. Then $\forall \tau \in Tid : lid_\tau(s_a) = lid_\tau(s'_a) \implies l_\tau(s) = l_\tau(s')$.

Proof. In a depth-first search on S following the algorithm of Figure 4.3, let n be the number of times that NEXTLOCAL returns to DFS from line 4 or line 5 of Figure 4.4. That is, n is the number of times that NEXTLOCAL is invoked with either $\delta_\tau = \delta_\varepsilon$ or $\exists y.\langle(i, \delta_\tau) \rightarrow y\rangle \in L_\tau$. We now prove the theorem by induction on n.

- (Base case) $n = 0$: Following NEXTLOCAL, if $n = 0$, all calls to NEXTLOCAL must return from either line 3 or line 7. That is, NEXTLOCAL has never been invoked with $\delta_\tau = \delta_\varepsilon$ or $\exists y.\langle(i, \delta_\tau) \rightarrow y\rangle \in L_\tau$. Hence, following the algorithm of Figure 4.4, every local state that has been visited must be assigned a unique identity. Therefore, in this situation, $lid_\tau(s_a) = lid_\tau(s'_a)$ holds if and only if $s_a = s'_a$. Obviously, $l_\tau(s) = l_\tau(s')$ holds in this situation.
- (Induction hypothesis) Let $k \geq 0$. For all n, $n \leq k$, Theorem 1 holds.
- (Induction step) Let $n = k + 1$. Let $s' \in S$ be the state and $\tau \in Tid$ be the thread such that by invoking NEXTLOCAL$(L_\tau, lid_\tau(s'), \delta_\tau)$ at line 8, n changes from k to $k + 1$. Consider the situation that DFS has finished executing line 8 of Figure 4.3, but has not started executing line 9. Let $s \in S$ be a state and t be a transition such that $s \xrightarrow{t} s'$ and $\tau = tid(t)$. There are two cases with respect to s':

- If $\delta_\tau = \delta_\varepsilon$, according to line 7 of DFS, we have $l_\tau(s) = l_\tau(s')$, and $lid_\tau(s_a) = lid_\tau(s'_a)$. Obviously $lid_\tau(s_a) = lid_\tau(s'_a) \implies l_\tau(s) = l_\tau(s')$ holds. Hence the theorem holds.
- If $\exists y.\langle(lid_\tau(s_a), \delta_\tau) \rightarrow y\rangle \in L_\tau$: Let $s_1, s_2 \in S$ be the two states that have been visited and t_1 be the transition such that $\langle(lid_\tau(s_a), \delta_\tau) \rightarrow y\rangle$ was added to L_τ when DFS explored $s_1 \xrightarrow{t_1} s_2$. Let s_{1_a} and s_{2_a}, respectively, be the abstract state of s_1 and s_2. Obviously we have $lid_\tau(s_{1_a}) = lid_\tau(s_a)$, $lid_\tau(s_{2_a}) = y$, and $l_\tau(s_2) \setminus l_\tau(s_1) = l_\tau(s') \setminus l_\tau(s)$. According to the induction hypothesis, $l_\tau(s_1) = l_\tau(s)$ must hold. As $l_\tau(s_2) \setminus l_\tau(s_1) = l_\tau(s') \setminus l_\tau(s)$, we have $l_\tau(s_2) = l_\tau(s')$. Hence, the theorem holds. Otherwise, it contradicts the induction hypothesis. □

Theorem 4.1 shows that to detect the visited states at runtime, instead of capturing the local states of threads in detail, we can conservatively infer the equality of local states using the local state changes δ. In practice, capturing δ is usually much easier than capturing the whole local state. Therefore, the task of explicitly capturing states at runtime can be greatly simplified. In the next section, we show how to combine our approach of state capturing with dynamic partial order reduction.

4.2 Stateful Dynamic Partial Order Reduction

In a depth-first search with DPOR, it seems that if visited states can be recognized, DPOR can simply stop the search at the visited states and start backtracking. However, it is not that simple because the transitions to be executed after the visited states may update the backtrack sets of the states in the search stack. Simple backtracking may result in unsoundness. For example, suppose we have two different executions

$$S_1 = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_{u-1}} s_u \ldots$$

$$S_2 = s_0 \xrightarrow{t'_0} s'_1 \xrightarrow{t'_1} \ldots \xrightarrow{t'_{v-1}} s'_v \ldots$$

of a program starting from the same state s_0, and s'_v is a visited state, $s'_v = s_u$, $u, v \geq 0$ (a fact also noted in [78]). Also, suppose that S_2 is explored after S_1 in the depth-first search with DPOR. Now, for every transition t that is executed after s'_v, the backtrack

sets of states $s_0, s'_1, \ldots, s'_v$ of S_2 may have to be updated. As a result, if we simply stop exploring the state space after s'_v, we may miss exploring a subset of the state space, i.e., this naive approach is not sound.

A straightforward way to fix this problem is that when a visited state is encountered, for each state s in the search stack, we update the backtrack set as follows – for all $t \in s.enabled$ where $tid(t) \notin s.done$, add $tid(t)$ into $s.backtrack$. This solves the problem of missing potential backtrack sets. However, it may also have the side effect of introducing too many unnecessary backtrack points that would not be introduced in the stateless DPOR. This side effect may put significant overhead on the stateful approach in which the stateful DPOR can actually run *slower* than the stateless one. Our initial experiments confirmed this.

Our solution to avoid unsoundness is to employ an efficient mechanism called *visible operation dependency graph*. Let s_v be a visited state encountered in the stateful DPOR. As only the visible operations of transitions determine whether two transitions are dependent or not, our approach is to compute a summary for the state space the element of which can be reachable from s_v. This summary captures all the visible operations that might be executed from s_v in one or more steps. We can use this summary to update the backtrack sets of the states that are in the search stack.

Obviously, computing such a summary for every state is very heavy-weight. However, we observe that with multithreading, the programs are usually designed in such a way that each thread is assigned some specific tasks to get the most benefit out of parallelism. The number of resources that require mutual exclusive accesses, and the number of conditions that threads need to be synchronized are limited, and usually small in number. This implies that *while the number of states of a multithreaded program can be large, the number of visible operations that each thread may execute is limited*.

For instance, for the program of Figure 4.1, although the number of states can be large, the only visible operation that `thread1` and `thread2` may execute is updating the global object `d`.

Based on this observation, instead of trying to maintain a summary for each state and keep the summaries updated, we compute the summary dynamically only when a

visited state is encountered by looking up the visible operation dependency graph that is constructed dynamically during the search.

In more detail, let $M = (S, s_0, \Gamma)$ be a multithreaded program. Let $\mathcal{T}$ be the transition set of M. A visible operation dependency graph $G = \langle V, E \rangle$ for M is a directed graph that captures the happen-before relation of visible operations for the traversed state space. Every node $v \in V$ of G is a visible operation. That is, $\forall v \in V : \exists t \in \mathcal{T} : t_g = v$. For each transition sequence $s_1 \xrightarrow{t} s_2 \xrightarrow{t'} s_3$ we encounter during the search, we add a directed edge (t_g, t'_g) into the graph.

In a depth-first search, when a visited state s is encountered, all the states that are reachable from s must have been visited because of the depth-first search. Hence, all the visible operations that may be executed in states reachable from s must have been executed. Therefore, we can traverse the visible operation dependency graph to find out all the visible operations that may be executed from some transition in $s.enabled$, and use this as a summary to update the backtrack sets of the states that are in the search stack. As the size of the graph is proportional to the number of visible operations that a multithreaded program may execute, this is a light-weight method for computing summaries of the visited states.

Figure 4.5 presents our stateful dynamic partial order reduction algorithm (SDPOR). The procedure SDPOR takes three parameters: the search stack S, the state hash table H, and the visible operation dependency graph G. Similar to DPOR, given a multithreaded program P, SDPOR first explores an arbitrary interleaving of the program, and thereafter, continues explore alternative interleavings until all relevant interleavings are explored, i.e., when no backtrack points are in the search stack. The differences between SDPOR and DPOR are the following:

- SDPOR uses a hash table H to record the visited states (line 9 of Figure 4.5). When a visited state is encountered, SDPOR conservatively updates the backtrack sets for states in the search stack S, and starts backtracking (line 5-7 of Figure 4.5).
- SDPOR uses a visible operation dependency graph G to dynamically learn the happen-before relation of visible operations during the depth-first search (line 19

```
1:  Initially: S.push(s_0); H is empty; G is empty;

2:  SDPOR(S, H, G) {
3:    s ← S.top;
4:    if (s ∈ H) {
5:      let U = {v | ∃t ∈ s.enabled, v is reachable in G from the node t_g};
6:      for each t ∈ U, UPDATEBACKTRACKSETS(S, t);
7:      return;
8:    }
9:    add s into H;
10:   for each t ∈ s.enabled, UPDATEBACKTRACKSETS(S, t);
11:   if (∃ thread τ, ∃t ∈ s.enabled, tid(t) = τ){
12:     s.backtrack ← {τ};
13:     s.done ← ∅;
14:     while (∃h ∈ s.backtrack \ s.done) {
15:       s.backtrack ← s.backtrack \ {h};
16:       s.done ← s.done ∪ {h};
17:       let t ∈ s.enabled, tid(t) = h, and let s' = next(s, t);
18:       S.push(s');
19:       if ∃s_x ∈ S s.t. s_x --t_x--> s --t--> s', add a directed edge (t_{x_g}, t_g) to G
20:       SDPOR(S, H, G);
21:       S.pop();
22:     }
23:   }
24: }
```

Figure 4.5. Stateful dynamic partial order reduction

of Figure 4.5). G is used to compute the state summary $\mathcal{U}$ when a visited state is encountered (line 5 of Figure 4.5).

SDPOR uses UPDATEBACKTRACKSETS of Figure 4.6 to update the backtrack sets for states in the search stack. UPDATEBACKTRACKSETS is the same as that in the stateless DPOR. We present it here for completeness.

Let $M = (S, s_0, \Gamma)$ be a multithreaded program. Let s be a state in S. We use R_s to denote the set of states that are reachable from s by executing one or more transitions. Obviously we have $R_s \subseteq S$.

Let SDPOR$_k$ be the algorithm of Figure 4.7. Comparing with SDPOR, the only difference between SDPOR$_k$ and SDPOR is that SDPOR$_k$ takes one more parameter k,

```
1: UPDATEBACKTRACKSETS(S, t) {
2:    let T be the sequence of transitions that are executed from the initial state of the
      program, following the sequence of states in S;
3:    let t_d be the latest transition in T that is dependent and may be co-enabled with t;
4:   if (t_d ≠ null){
5:    let s_d be the state in S from which t_d is executed;
6:    let E be {q ∈ s_d.enabled | tid(q) = tid(t), or q in T, q happened after t_d and is
      dependent with some transition in T which was executed by tid(t) and happened
      after q }
7:    if (E ≠ ∅)
8:      choose any q in E, add tid(q) to s_d.backtrack;
9:    else
10:     s_d.backtrack ← s_d.backtrack ∪ {tid(q) | q ∈ s_d.enabled};
11:   }
12: }
```

Figure 4.6. Updating the backtrack sets for states in the search stack

which bounds SDPOR$_k$ to return only at the first k visited states. In more detail, SDPOR$_k$ uses a global counter c to record the number of visited states that it has encountered during the depth-first search (line 5 of Figure 4.7). When a visited state s_v is encountered, if $c \leq k$, then SDPOR$_k$ updates the backtrack sets of states in the search stack (line 7-8 of Figure 4.7) and returns immediately; otherwise, SDPOR$_k$ continues exploring R_{s_v}.

We have a class of algorithms {SDPOR$_0$, SDPOR$_1$, ... } by assigning k specific values. Let $\mathcal{A}$ denote an algorithm that explores the state space of M. Let $S_{\mathcal{A}} \subseteq S$ refer to the set of states that is explored using $\mathcal{A}$ by starting from s_0. Obviously, we have $S_{\text{DPOR}} = S_{\text{SDPOR}_0}$ and $S_{\text{SDPOR}} = S_{\text{SDPOR}_\infty}$.

To prove the correctness of Theorem 4.2, we first prove Lemma 4.1, which characterizes the relationship between SDPOR$_k$ and SDPOR$_{k+1}$.

Lemma 4.1 Let $M = (S, s_0, \Gamma)$ be a multithreaded program. Let $s, s' \in S$ and t be a transition of M such that $s \xrightarrow{t} s'$. Let $k \geq 0$. If $s \xrightarrow{t} s'$ is explored by SDPOR$_k$, it must be explored by SDPOR$_{k+1}$.

Proof. Let r be the value of the global variable c when we finish checking the state space of M using SDPOR$_k$. There are two cases with respect to r:

```
1:  Initially: c = 0; S.push(s_0); H is empty;

2:  SDPOR_k(S, H) {
3:    s ← S.top;
4:    if (s ∈ H) {
5:      c ← c + 1;
6:      if (c ≤ k) {
7:        let T_p = {t_g | t can be executed from states that are reachable from s };
8:        for each t ∈ T_p, UPDATEBACKTRACKSETS(S, t);
9:        return;
10:     }
11:   }
12:   add s into H;
13:   for each t ∈ s.enabled, UPDATEBACKTRACKSETS(S, t);
14:   if (∃ thread τ, ∃t ∈ s.enabled, tid(t) = τ){
15:     s.backtrack ← {τ};
16:     s.done ← ∅;
17:     while (∃h ∈ s.backtrack \ s.done) {
18:       s.backtrack ← s.backtrack \ {h}, s.done ← s.done ∪ {h};
19:       let t ∈ s.enabled, tid(t) = h, and let s' = next(s, t);
20:       S.push(s');
21:       SDPOR_k(S, H);
22:       S.pop();
23:     }
24:   }
25: }
```

Figure 4.7. SDPOR$_k$ only have immediate backtracking at the first k revisited states.

- If $r \leq k$, obviously that the state spaces traversed by SDPOR$_k$ and SDPOR$_{k+1}$ are identical. The lemma holds.
- If $r > k$, let v_i be the i-th visited state SDPOR$_k$ and SDPOR$_{k+1}$ encounter while exploring the state space of M. Let $\Gamma_i^x \subseteq \Gamma$ be the transition relation that has been explored by SDPOR$_x$ before reaching the i-th visited states. It is obvious that $\forall i, i \leq k+1 : \Gamma_i^k = \Gamma_i^{k+1}$ holds.

  Now we consider the exploration of the search space by SDPOR$_k$ and SDPOR$_{k+1}$ after they encounter v_{k+1}. Following the algorithm of Figure 4.7, while encountering v_{k+1}, SDPOR$_k$ is equivalent to a stateless search that does not record the

search history, and explores $R_{v_{k+1}}$. However, SDPOR$_{k+1}$ does not explore $R_{v_{k+1}}$. Before encountering v_{k+1}, the state spaces explored by SDPOR$_k$ and SDPOR$_{k+1}$ are identical. Hence, the search stacks of SDPOR$_k$ and SDPOR$_{k+1}$ are identical at the point of encountering v_{k+1}. Let s^k and s^{k+1} denote the correspondent states that are respectively in the search stacks of SDPOR$_k$ and SDPOR$_{k+1}$. To prove the lemma, all that we need to prove is that, when SDPOR$_k$ and SDPOR$_{k+1}$ backtrack from $s_{v_{k+1}}$, for all pairs of states $\langle s^k, s^{k+1}\rangle$, we have $s_{sdpor_k}.backtrack \subseteq s_{sdpor_{k+1}}.backtrack$.

This can be proven by contradiction. Suppose while backtracking from v_{k+1}, there exists $\langle s^k, s^{k+1}\rangle$ such that $s^k.backtrack \supset s^{k+1}.backtrack$. This implies that $\exists h \in Tid : h \in s^k.backtrack \wedge h \notin s^{k+1}.backtrack$. As the only difference between SDPOR$_k$ and SDPOR$_{k+1}$ is that SDPOR$_k$ explores $R_{v_{k+1}}$ while SDPOR$_{k+1}$ does not, this can happen if and only if $\exists s_1, s_2 \in R_{v_{k+1}}, \exists t \in s^{k+1} : s_1 \xrightarrow{t_1} s_2$ and t_1 is dependent with t. However, following the algorithm of Figure 4.7, if the execution of t_1 happens before backtracking v_{k+1} in SDPOR$_k$, $t_1 \in T_p$. Hence, if $h \in s^k.backtrack$, h must be in $s^{k+1}.backtrack$. This contradicts $h \notin s^{k+1}.backtrack$. □

Let $M = (S, s_0, \Gamma)$ be a multithreaded program. Let G be the transition dependency graph of M. G is dynamically constructed following the algorithm of Figure 4.5. Let $\mathcal{U}$ be the set of visible operations that is computed at line 5 of Figure 4.5. Let T_p be the set of visible operations that is computed as line 7 of Figure 4.7. We have the following lemma:

Lemma 4.2 $T_p \subseteq \mathcal{U}$.

Proof. Following the algorithm of Figure 4.5, it is clear that while backtracking from a state s, all transition dependency edges that can appear in R_s must have been added to G. Therefore, we have $\forall t \in T_p : t \in \mathcal{U}$. □

Theorem 4.2 Let $M = (S, s_0, \Gamma)$ be a multithreaded program. For every execution of a transition $s \xrightarrow{t} s'$ of M, if it is explored by the stateless DPOR, it must be explored by SDPOR.

Proof. With Lemma 4.1, Lemma 4.2, $S_{\text{DPOR}} = S_{\text{SDPOR}_0}$ and $S_{\text{SDPOR}} = S_{\text{SDPOR}_\infty}$, it is clear that the theorem holds. □

This theorem shows that given a multithreaded program, the *set* of states visited by SDPOR is a superset of the states visited by DPOR. This means that SDPOR is a conservative approach.

Note that DPOR may reexplore the *same* state space many times, while SDPOR will, whenever abstract states are found in the hash table, avoid all those revisits! Therefore, the bag of DPOR visited states usually has size far higher than the bag of states that SDPOR visits. This is the reason that SDPOR can be more efficient than DPOR in checking multithreaded programs. The experiments to be shown in Section 4.4 confirm this.

4.2.1 Efficient SDPOR

The algorithm of Figure 4.5 assumes that the model checker is capable of capturing the program states precisely. Here we present the practical algorithm that combines SDPOR of Figure 4.5 with the light-weight state capturing scheme that is presented in Section 4.1. Figure 4.8 shows the algorithm. Here the procedure SDPOR takes four parameters – although the parameter S is still the search stack, the element of the stack is a pair (s, s_a) such that $s \in S$, $s_a \in S_a$, and s_a is the abstract state of s. The parameter L is the set of local state hash tables. The parameters H, G are the same as in Figure 4.5.

Comparing with SDPOR in Figure 4.5, in this combined algorithm, line 18-19 are the new statements for computing the abstract states, line 3 and line 20 are modified to adapt the changes of the search stack, and line 22 is changed to adapt the local state hash tables. The rest of the algorithm is the same as in Figure 4.5.

4.3 Implementation

We implemented the algorithm of Figure 4.8 based on our stateless runtime model checker `Inspect` [47, 97]. `Inspect` can instrument a multithreaded C program with code to intercept the visible operations, compile the instrumented program along with a

```
1: Initially: S.push(s_0); H is empty; ∀L_τ ∈ L : L_τ is empty; G is empty;

2: SDPOR(S, H, L, G) {
3:   ⟨s, s_a⟩ ← S.top;
4:   if (s_a ∈ H) {
5:     let U = {v | ∃t ∈ s.enabled, v is reachable in G from the node t_g};
6:     for each t ∈ U, UPDATEBACKTRACKSETS(S, t);
7:     return;
8:   }
9:   add s_a into H;
10:  for each t ∈ s.enabled, UPDATEBACKTRACKSETS(S, t);
11:  if (∃ thread τ, ∃t ∈ s.enabled, tid(t) = τ){
12:    s.backtrack ← {τ};
13:    s.done ← ∅;
14:    while (∃h ∈ s.backtrack \ s.done) {
15:      s.backtrack ← s.backtrack \ {h};
16:      s.done ← s.done ∪ {h};
17:      let t ∈ s.enabled, tid(t) = h, and let s' = next(s, t);
18:      let δ_h = l_h(s') \ l_h(s), and x = NEXTLOCAL(L_h, lid_h(s_a), δ_h);
19:      let s'_a ∈ S_a s.t. g(s'_a) = g(s') ∧ ls(s'_a) = ls(s_a)[τ := x] ∧ s'_a.PCs = s'.PCs;
20:      S.push(⟨s', s'_a⟩);
21:      if ∃s_x ∈ S s.t. s_x --t_x--> s --t--> s', add a directed edge (t_{x_g}, t_g) to G
22:      SDPOR(S, H, L, G);
23:      S.pop();
24:    }
25:  }
26: }
```

Figure 4.8. The combination of SDPOR with the light-weight state capturing scheme

stub library into an executable, and uses a centralized monitor to systematically explore interleavings of the program by concretely executing the program.

`Inspect` uses escape analysis [82] to reveal potential visible operations in a multi-threaded program. Based on this, we implemented an intraprocedural forward data-flow analysis to determine the local state changes between successive visible operations. For any transition t, we treat δ_t as $\delta_\perp$ when (i) t_g of t is the first visible operation in the procedure, or (ii) there are function calls or updates of pointers between the previous

visible operation and t_g. Otherwise, we compute δ_t by capturing the changes of the local variables.

In [97], we described how automated instrumentation is done for stateless runtime checking. To capture the local state changes of threads, we instrument extra code into the program under test to inform the scheduler the local state changes.

4.4 Experimental Results

We performed the experiments on a set of multithreaded benchmarks: `example1` is the program shown in Figure 4.1, `sharr` is a benchmark from [33]. It has two threads that iteratively write to different elements of a shared array. `bbuf` is an implementation of a bounded buffer with concurrent producers and consumers. `bzip2smp` [41] and `pfscan` [44] are two real multithreaded applications. `bzip2smp` is a multithreaded compression program that uses multiple threads to speed up the compression of a file. `pfscan` is a multithreaded file scanner that uses multiple threads to search in parallel through directories. `bzip2smp` contains 6.4k lines of C code, and `pfscan` has 1k lines of C code.

Table 4.1 shows the experimental results using stateless DPOR and our stateful approach. All the experiments were performed on a PC with an Intel quad-core CPU of 2.4GHz and 2GB of memory. We use "-" to denote that the program cannot be completely checked within 24 hours (86400 seconds).

We compared SDPOR with DPOR on the number of executions (or runs) they require to check a program, the number of transitions explored, and the checking time. From the experimental results, it is clear that our stateful DPOR approach is more effective than the stateless DPOR, in reducing both the number of transitions to be explored and the checking time.

4.5 Related Work

There has been substantial work on stateful model checking. Model checkers such as SPIN [39] and Bogor [81] have been very successful in revealing bugs and proving the correctness of systems. However, it is difficult for classic model checkers to check

Table 4.1. Experimental results on the comparison between DPOR and SDPOR

Benchmarks	# trds	DPOR			SDPOR		
		runs	trans	time(s)	runs	trans	time(s)
`example1`	2	-	-	-	35	2084	1.4
`sharr`	2	-	-	-	98	18.6k	6
`bbuf`	4	47k	1058k	938	16.2k	349.7k	345
`bzip2smp`	4	-	-	-	4.6k	26.4k	1311
`bzip2smp`	5	-	-	-	18.7k	92.3	9456
`bzip2smp`	6	-	-	-	51.4k	236.8	25659
`pfscan`	3	84	1,157	1	71	967	0.49
`pfscan`	4	13.6k	189.2k	241	3.2k	40k	57
`pfscan`	5	-	-	-	272.9	3402k	5329

realistic multithreaded programs, which often heavily use library routines and have sophisticated memory manipulation operations. The advantage of our approach is that it is able to directly examine the programs and avoid the modeling overhead (and potential consistency issues).

Musuvathi et al. [71] developed CMC, which is a runtime model checker that can precisely capture the states of a concurrent program by snapshoting the kernel space plus the user space of the program. In our work, we do not capture the whole state of a multithreaded program. Instead, we abstract the local states of threads as identities, and try to recognize the same states in different executions by tracking the local state changes. Compared with CMC, our approach is more light-weight in capturing states at runtime.

Gueta et al. [33] proposed Cartesian partial order reduction, which reduces the search space by delaying unnecessary context switches using Cartesian vectors. Cartesian partial order reduction performs stateful search, and can deal with cyclic state space. However, their approach assumed that the model checker is capable of capturing the states precisely, and did not address the problem of practical state capturing at runtime. We present a light-weight method for capturing the states of concurrent programs at runtime, and show how to adapt the stateful search into dynamic partial order reduction.

Yi et al. [102] proposed another stateful dynamic partial order reduction method based on the summary of interleavings. [102] also assumed that the model checker is able to precisely capture the states, and did not address the problem of state capturing at runtime. Their definition of *summary* for interleavings is a set of happen-before transition mappings. In their method, each state is associated with a summary of interleaving information, which could be very expensive to store and to keep updated. When a visited state is encountered, our SDPOR computes a summary for the states that can be reached from the visited state in one or more steps. Different from their work, we use a visible operation dependency graph to dynamically compute the summary when a visited state is encountered. As a result, in our approach, the state summary computation is more light-weight.

4.6 Summary

We present an efficient stateful runtime model checking approach to testing multi-threaded C programs in this paper. It incorporates the dynamic partial order reduction (DPOR) with the state space of model checking. To overcome the problem of capturing local states of multithreaded C programs at runtime, we propose a novel light-weight state abstraction scheme to conservatively capture local states. We also propose a stateful dynamic partial order reduction algorithm, and show how to combine it with our light-weight state capturing scheme. Compared with the stateless DPOR approach, our approach is able to detect commutativity of transitions among different interleavings at runtime, and avoid exploring redundant interleavings. The experiments show that our approach is more efficient than stateless DPOR in checking realistic programs.

CHAPTER 5

PROPERTY-DRIVEN PRUNING FOR FAST DATA RACE DETECTION

Dynamic model checking as in [27, 21, 72, 99, 97] can directly check programs written in full-fledged programming languages such as C and Java. For detecting data races, these methods are sound (no bogus race) due to their concrete execution of the program itself as opposed to a model. Although a bounded analysis is used in [72], the other methods [27, 21, 97, 99] are complete for terminating programs (do not miss real races) by systematically exploring the state space without explicitly storing the intermediate states.

Although such dynamic software model checking is both sound and complete, the search is often inefficient due to the astronomically large number of thread interleavings and the lack of property-specific pruning. Dynamic partial order reduction (DPOR) techniques [21, 97, 23] have been used in this context to remove the redundant interleavings from each equivalence class, provided that the representative interleaving has been explored. However, the pruning techniques used by these DPOR tools have been generic, rather than *property-specific*.

Without a conservative or warranty type of analysis tailored toward the property to be checked, model checking has to enumerate all the equivalence classes of interleavings. Our observation is that, as far as race detection is concerned, many equivalence classes themselves may be redundant. Figure 5.1 shows a motivating example, in which two threads use locks to protect accesses to shared variables x, y, and z. A race condition between a_6 and b_{10} may occur when b_4 is executed before a_2, by setting c to 0. Let the first execution sequence be $a_1 \ldots a_{11} b_1 \ldots b_9 b_{11}$. According to the DPOR algorithm by Flanagan and Godefroid [21], since a_{10} and b_3 have a read-write conflict, we need to

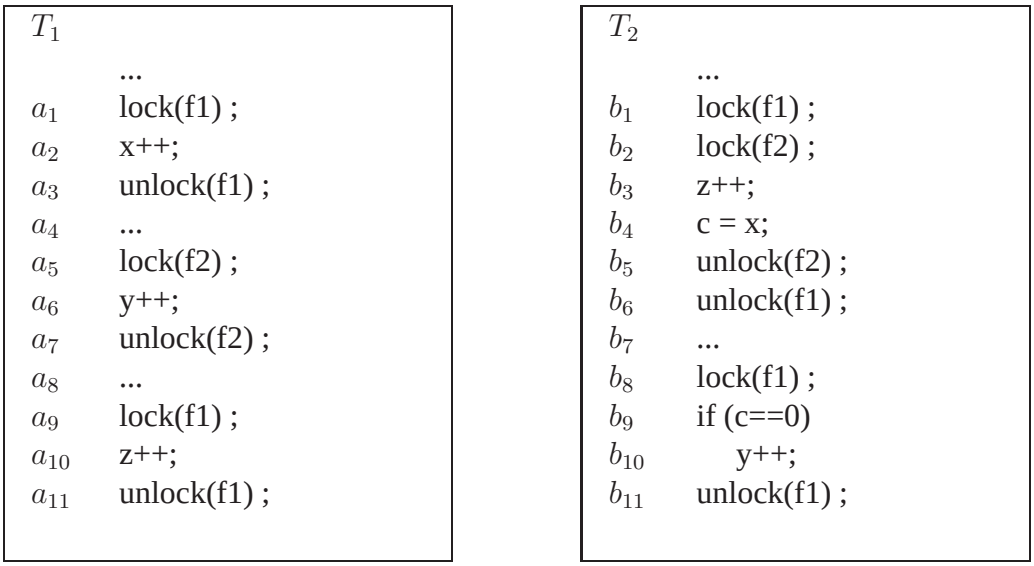


Figure 5.1. Race condition on accessing variable y (assume that $x = y = 0$ initially)

backtrack to a_8 and continue the search from $a_1 \ldots a_8 b_1$. As a generic pruning technique, this is reasonable since the two executions are not Mazurkiewicz-trace equivalent [67]. For data race detection, however, it is futile to search any of these execution traces in which a_6 and b_{10} cannot be simultaneously reachable (which can be revealed by a conservative lock set analysis). We provide a property-specific pruning algorithm to skip such redundant interleavings and backtrack directly to a_1.

In this paper, we propose a trace-based dynamic lock set analysis to prune the search space in the context of dynamic model checking. Our main contributions are (i) a new lock set analysis of the observed execution trace for checking whether the associated search subspace is race-free, and (ii) property driven pruning in a backtracking algorithm using depth-first search.

We analyze the various alternatives of the current execution trace to anticipate race conditions in the corresponding search space. Our trace-based lock set analysis relies on both information derived from the dynamic execution and information collected statically from the program; therefore, it is more precise than the purely static lock set analysis conducted a priori on the program [20, 22, 80, 55, 93]. Our method is also different from the *Eraser*-style dynamic lock set algorithms [83, 75], since our method decides whether

the entire search subspace related to the concrete execution generated is race-free, not merely the execution itself. The crucial requirement for a method to be used in our framework for pruning of the search space is completeness—pruning must not remove real races. Therefore, neither the aforementioned dynamic lock set analysis nor the various predictive testing techniques [85, 9] based on happens-before causality (sound but incomplete) can be be used in this framework. CHESS [72] can detect races that may show up within a preemption bound; it exploits the preemption bounding for pruning, but does not exploit the lock semantics to effect reduction.

In our approach, if the search subspace is found to be race-free, we prune it away during the search by avoiding backtracks to the corresponding states. Recall that essentially the search is conducted in a DFS order. If there is a potential race, we analyze the cause in order to compute a proper backtracking point. Our backtracking algorithm shares the same insights as the DPOR algorithm [21], with the additional pruning capability provided by the trace-based lock set analysis. Note that DPOR relies solely on the *independence relation* to prune redundant interleavings (if t_1, t_2 are independent, there is no need to flip their execution order). In our algorithm, *even if* t_1, t_2 *are dependent*, we may skip the corresponding search space if flipping the order of t_1, t_2 does not affect the reachability of any race condition. If there is no data race at all in the program, our algorithm can obtain the desired race-freedom assurance much faster.

5.1 Race-Free Search Subspace

Given an execution sequence $s_0, \ldots, s_i, \ldots, s_n$ stored in the stack S and a state s_i ($0 \leq i \leq n$), we check (conservatively) whether the search space starting from s_i is race-free. This search subspace consists of all the execution traces sharing the same prefix $s_0, \ldots, s_i$. During dynamic model checking, instead of backtracking for each conflicting transition pair as in DPOR, we backtrack to state s_i only if the corresponding search subspace has potential races.

5.1.1 Set of Lock Sets

Let $T = \{t_1, \ldots, t_n\}$ be a transition sequence such that $s_0 \xrightarrow{t_1} s_1 \ldots \xrightarrow{t_n} s_n$. First, we project T to each thread as a sequence $T_\tau = \{t_{\tau_1}, \ldots, t_{\tau_k}\}$ of thread-local transitions; that is, $\forall t \in T_\tau : tid(t) = \tau$. For the example in Figure 5.1, T is projected to $T_1 = \{a_1, \ldots, a_{11}\}$ and $T_2 = \{b_1, \ldots, b_9, b_{11}\}$. Next, we partition each thread-local sequence T_τ into smaller segments. In the extreme case, each segment would consist of a single transition. For each segment $seg_i \subseteq T_\tau$, we identify the global variables that may be accessed within seg_i; for each access, we also identify the corresponding *lockset*—the set of locks held by thread τ when the access happens.

Definition 5.1 For each segment seg_i and global variable x, the set $lsSet_x(seg_i)$ consists of all the possible locksets that may be held when x is accessed in seg_i.

By conservatively assuming that transitions of different threads can be interleaved arbitrarily, we check whether it is possible to encounter a race condition. Specifically, for each global variable x, and for each pair (seg_i, seg_j) of transition segments from different threads, we check whether $\exists set_1 \in lsSet_x(seg_i), set_2 \in lsSet_x(seg_j)$ such that $set_1 \cap set_2 = \emptyset$. An empty set represents a potential race condition—x is not protected by a common lock. The result of this analysis can be refined by further partitioning seg_i, seg_j into smaller fragments. To check whether the search space starting from s_i is race-free, we will conservatively assume that $t_{i+1}, \ldots, t_n$ (transitions executed after s_i in T) may interleave arbitrarily, subject only to the program orders.

Note first that the lock set analysis is thread-local, i.e., the analysis is performed on a single thread at a time. Second, a precise computation of $lsSet_x(seg_i)$ as in Definition 5.1 requires the inspection of all feasible execution traces (exponentially many) in which x is accessed in seg_i; we do not perform this precise computation. For conservatively checking the race-free subspace property, it suffices to consider a set of locksets S such that any constituent lock set of S is a subset of the actually held locks. For instance, the coarsest approximation is $lsSet_x(seg_i) = \{\emptyset\}$; that is, x is not protected at all. Under this coarsest approximation for seg_i, if another thread also accesses x in seg_j, our algorithm will report a potential race condition between seg_i and seg_j.

Consider again the example in Figure 5.1, let the first execution trace be partitioned into

$$seg_1 = a_1, \ldots, a_8 \qquad seg_3 = b_1, \ldots, b_7$$
$$seg_2 = a_9, \ldots, a_{11} \qquad seg_4 = b_8, \ldots, b_{11}.$$

Since seg_2 shares only z with seg_3, and z is protected by lock $f1$, any execution trace starting from seg_1 is race-free. Therefore, we do not need to backtrack to a_8.

However, the concrete execution itself may not be able to provide enough information to carry out the above analysis. Note that by definition, $lsSet_x(seg_i)$ must include *all the possible locksets* that may be formed in an interleaving execution of seg_i. In Figure 5.1, for instance, although y is accessed in both threads (a_6 and b_{10}), the transition b_{10} does not appear in seg_4 since the `else`-branch was taken. However, $lsSet_y(seg_4)$ is $\{\{f1\}\}$. In general, we need a *may-set* of shared variables that are accessed in seg_i and the corresponding *must-set* of locks protecting each access. We need the information of all the alternative branches in order to compute these sets at runtime.

5.1.2 Handling the Other Branch

Our solution is to augment all branching statements in the form of `if(c)-else`, through source code instrumentation, so that the information of not-yet-executed branches (computed *a priori*) is readily available to our analysis during runtime. To this end, for both branches of every `if-else` statement, we instrument the program by inserting calls to the following routines ('rec' stands for record):

- `rec-var-access-in-other-branch(`x, L_{acq}, L_{rel}`)` for each access to x; with the set L_{acq} of locks acquired and the set of L_{rel} of locks released before the access.
- `rec-lock-update-in-other-branch(`L_{acq}, L_{rel}`)`; with the set L_{acq} of locks acquired and the set L_{rel} of locks released in the other branch.

The instrumentation is illustrated by a simple example in Figure 5.2. In addition to the above routines, we also add recording routines to notify the scheduler about the branch start and end. When the `if`-branch is executed, the scheduler knows that, in the `else`-branch, x is accessed and lock C is acquired before the access (line 4); it also knows that C is the only lock acquired and no lock is released throughout that branch

```
1:  lock(B)
2:  if (c) {
3:    rec-branch-begin();                              //added
4:    rec-var-access-in-other-branch(x,{C},{});  //added
5:    rec-lock-update-in-other-branch({C},{});   //added
6:    lock(A);
7:    x++;
8:    unlock(A);
9:    y=5;
10:   lock(C);
11:   rec-branch-end();                                //added
12: }else {
13:   rec-branch-begin();                              //added
14:   rec-var-access-in-other-branch(x,{A},{ }); //added
15:   rec-var-access-in-other-branch(y,{A},{A}); //added
16:   rec-lock-update-in-other-branch({A,C},{A});//added
17:   lock(C);
18:   x++;
19:   rec-branch-end();                                //added
20: }
21: z++;
22: unlock(C);
23: unlock(B)
```

Figure 5.2. Instrumenting the branching statements of each thread

(line 5). Similarly, when the `else`-branch is executed, the scheduler knows that in the `if`-branch, x, y are accessed and lock A is protecting x but not y. According to lines 5 and 16, lock C will be held at the branch merge point because $(L_{acq} \setminus L_{rel}) = \{C\}$. Therefore, our algorithm knows that z is protected by both B and C.

The information passed to these recording routines need to be collected *a priori* by a static analysis of the individual threads. Note that neither the set of shared variables nor any of the corresponding lock sets L_{acq}, L_{rel} has to be precise. For a conservative analysis, it suffices to use an over-approximated set of shared variables, a subset $\check{L}_{acq} \subseteq L_{acq}$ of acquired locks, and superset $\hat{L}_{rel} \supseteq L_{rel}$ of released locks. By using $\check{L}_{acq}$ and $\hat{L}_{rel}$, we can compute a must-set $(\check{L}_{acq} \setminus \hat{L}_{rel})$, which is a subset of the actually held locks.

5.1.3 Checking Race-Free Subspace

The algorithm for checking whether a search subspace is race-free is given in Figure 5.3. For each transition $t \in T$ and global variable x, we maintain the following:

- $lsSet(t)$, the *set of lock sets* held on one of the paths by t.
- $mayUse(t, x)$ if t is a branch begin, the set of lock sets of x in the other branch.

In state s_i, the set ls_τ of locks held by thread τ is known. First, we use COMPUTELOCKSETS to update $lsSet(t)$ and $mayUse(t, x)$ for all variables x accessed and transitions t executed after s_i. Potential race conditions are checked by intersecting pairwise lock sets of the same variable in different threads. If any of the intersection in line 11 is empty, SUBSPACERACEFREE r returns FALSE.

In Figure 5.4, COMPUTELOCKSETS starts with ls_τ, which comes from the concrete execution and hence is precise. T_τ consists of the following types of transitions: (1) instrumented recording routines; (2) lock/unlock; (3) other program statements. The stack $update$ is used for temporary storage. Both $lsSet(t)$ and $mayUse(t, x)$ are *sets of lock sets*, of which each constituent lock set corresponds to a distinct unobserved path (a path skipped due to a false branch condition) or variable access. Note that we do not merge lock sets from different branches into a single must-lockset, but maintain them as separate entities in $lsSet(t)$ and then propagate to the subsequent transitions in T_τ.

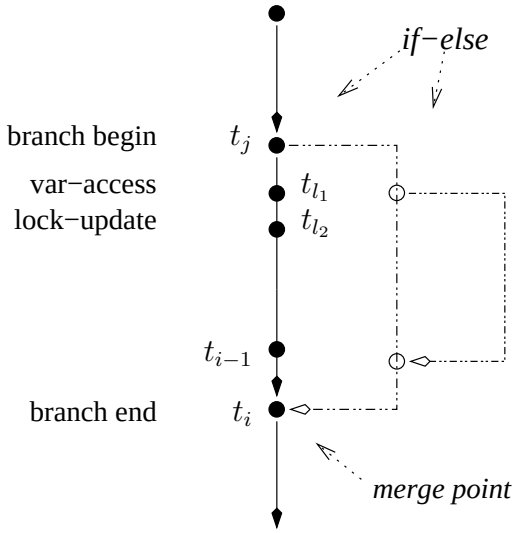
```
1:  SUBSPACERACEFREE(s_i) {
2:    let T = {t_1, t_2, ..., t_m} such that s_i -t_1-> ... -t_m-> s_{m+1} and s_{m+1}.enabled = ∅;
3:    for each (τ ∈ Tid) {
4:      let T_τ = {t_{τ_1}, ..., t_{τ_k}} be a subsequence of T such that ∀t ∈ T_τ : tid(t) = τ;
5:      let ls_τ be the set of locks held by thread τ at s_i;
6:      COMPUTELOCKSETS(ls_τ, T_τ);
7:    }
8:    for each (global variable x) {
9:      let t_1, t_2 ∈ T, tid(t_1) ≠ tid(t_2), both may access x, and at least one is a write;
10:     let ls_1 ∈ (lsSet(t_1) ∪ mayUse(t_1, x)), let ls_2 ∈ (lsSet(t_2) ∪ mayUse(t_2, x));
11:     if (∃ls_1, ls_2 such that ls_1 ∩ ls_2 = ∅) return FALSE;
12:   }
13:   return TRUE;
14: }
```

Figure 5.3. Checking whether the search subspace from s_i is race-free at run time

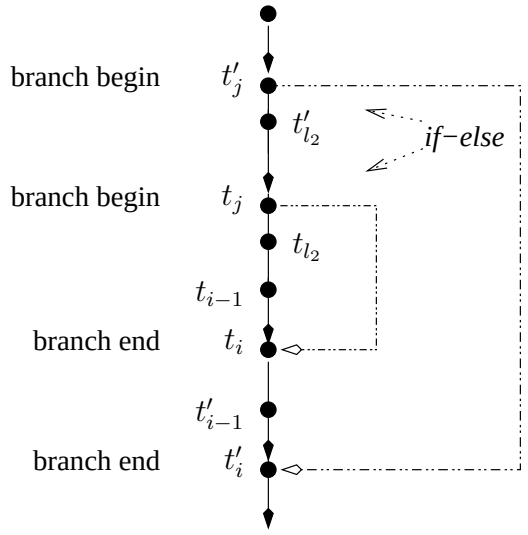
```
1:  COMPUTELOCKSETS(ls_τ, T_τ) {
2:    let lsSet(t_0) ={ ls_τ };
3:    let T_τ = {t_1, ..., t_k}; ∀t_i ∈ T_τ, ∀x : lsSet(t_i) ← ∅ and mayUse(t_i, x) ← ∅;
4:    i ← 1;
5:    while (i ≤ k) {
6:      if (t_i is rec-branch-begin)
7:        update.push(∅);
8:      if (t_i is lock(f1))
9:        lsSet(t_i) ← {ls ∪ {f1} | ls ∈ lsSet(t_{i−1});
10:     else if (t_i is unlock(f1))
11:       lsSet(t_i) ← {ls \ {f1} | ls ∈ lsSet(t_{i−1});
12:     else if (t_i is rec-var-access-in-other-branch(x, L_acq, L_rel))
13:       let t_j be the last branch begin that precedes t_i;
14:       mayUse(t_j, x) ← mayUse(t_j, x) ∪ {ls ∪ L_acq \ L_rel | ls ∈ lsSet(t_j)};
15:     else if (t_i is rec-lock-update-in-other-branch(L_acq, L_rel))
16:       let t_j be the last branch begin that precedes t_i;
17:       update.top() ← update.top() ∪ {ls ∪ L_acq \ L_rel | ls ∈ lsSet(t_j)};
18:     else if (t_i is rec-branch-end)
19:       lsSet(t_i) ← update.pop() ∪ lsSet(t_{i−1});
20:     else
21:       lsSet(t_i) ← lsSet(t_{i−1});
22:     i ← i + 1;
23:   }
24: }
```

Figure 5.4. Computing locksets that may be held by each transition in T_τ

Multiple branches may be embedded in the observed sequence T_τ, as shown in Figure 5.5. In Figure 5.5(a), the unobserved branch itself has two branches, each of which needs a recording routine in T_τ to record the lock updates. Inside COMPUTELOCKSETS, lock updates from t_{l_2} are stored temporarily in the stack $update$ and finally used to compute $lsSet(t_i)$ at the merge point. In Figure 5.5(b), the observed branch (from t'_j to t'_i) contains another observed branch (from t_j to t_i). This is why a stack $update$, rather than a set, is needed. Note that t'_{l_2} is executed before t_{l_2}, but $lsSet(t'_i)$ is computed after $lsSet(t_i)$.



(a) using t_{l_1} to compute $mayUse(t_j, x)$
using t_{l_2}, t_{i-1} to compute $lsSet(t_i)$



(b) using t_{l_2}, t_{i-1} to compute (inner) $lsSet(t_i)$
using t'_{l_2}, t'_{i-1} to compute (outer) $lsSet(t'_i)$

Figure 5.5. Multiple branches in an execution trace (observed and unobserved branches)

Let $s_j \xrightarrow{t_{j+1}} s_{j+1}$ be a branch begin and $s_{i-1} \xrightarrow{t_i} s_i$ be the matching branch end. From the pseudo code in Figure 5.4, it is clear that the following two theorems hold.

Theorem 5.1 $lsSet(t_i)$ contains, for each unobserved path from s_j and to s_i, a must-set of locks held at s_i (if that path were to be executed).

Theorem 5.2 $mayUse(t_i, x)$ contains, for each access of x in an unobserved path from s_j to s_i, a must-set of locks held when accessing x in that path.

Although the standard notion of lock sets is used in our analysis, the combination of dynamically computed information of the observed execution and statically computed information of not-yet-executed branches differentiates us from the existing dynamic [83, 75] and static [22, 20, 80, 55, 93] lock set algorithms. It differs from the Eraser-style lock set algorithms [83, 75] in that it has to consider not only the current execution but also the not-yet-activated branches. It differs from the purely static lock set analysis [22, 20, 80, 55, 93] in that it utilizes not only the statically computed program information, but also the more precise information derived dynamically from the execution. In particular, our lock set computation starts with a precise lock set ls_τ of the concrete execution (line 5 of Figure 5.3). In the presence of pointers to data and locks, a purely static analysis may be imprecise; the actual set of shared variables accessed or locks held during a concrete execution may be significantly smaller than the (conservatively computed) points-to sets of the pointers.

To sum up, we use code instrumentation, in particular, through the calls to recording routines, to provide the statically computed information about the not-yet-executed branches at runtime. During a dynamic execution, we start with the precise lock set (ls_τ) of a concrete transition subsequence $s_0, s_1, \ldots, s_i$, together with approximated lock sets ($\check{L}_{acq}, \hat{L}_{rel}$) of the not-yet-executed branches, to compute the set of lock sets that may be held in future (alternative) executions.

5.2 The Overall Algorithm

We rely on the conservative lock set analysis to prune the search space, and the concrete program execution to ensure that no bogus race is reported. The overall algorithm is given in Figure 5.6.

```
1: Initially: S is empty; PDPSEARCH(S, s_0)

2: PDPSEARCH(S, s) {
3:   if (s.enabled = ∅) {
4:     for (i = 0; i < S.size(); i++) {
5:       let s_b be the i-th element in S;
6:       for each (t ∈ s_b.enabled)
7:         PDPUPDATEBACKTRACKSETS(S, t);
8:     }
9:   }else {
10:    if (DETECTRACE(s)) exit (S);
11:    S.push(s);
12:    let τ ∈ Tid such that ∃t ∈ s.enabled : tid(t) = τ;
13:    s.backtrack ← {τ};
14:    s.done ← ∅;
15:    while (∃t: tid(t) ∈ s.backtrack and t ∉ s.done) {
16:      s.done ← s.done ∪ {t};
17:      s.backtrack ← s.backtrack \ {tid(t)};
18:      let s' ∈ S such that s -t-> s';
19:      PDPSEARCH(S, s');
20:      S.pop(s);
21: } } }

22: PDPUPDATEBACKTRACKSETS(S, t) {
23:   let T = {t_1, ..., t_n} be the sequence of transitions associated with S;
24:   let t_d be the latest transition in T that (1) is dependent and may be co-
      enabled with t, and (2) let s_d ∈ S be the state from which t_d is executed,
      SubspaceRaceFree(s_d) is FALSE;
25:   if (t_d ≠ null){
26:     let E be {q ∈ s_d.enabled | either tid(q) = tid(t), or q was executed after t_d in
      T and
            a happens-before relation exists for (q, t)}
27:     if (E ≠ ∅)
28:       choose any q in E, add tid(q) to s_d.backtrack;
29:     else
30:       s_d.backtrack ← s_d.backtrack ∪ {tid(q) | q ∈ s_d.enabled};
31:   }
32: }
```

Figure 5.6. Property driven pruning based dynamic race detection algorithm

The procedure PDPSEARCH, where PDP stands for Property-Driven Pruning, takes the stack S and a state s as input. Each time PDPSEARCH is called on a new state s, lines 10-24 will be executed. DETECTRACE(s) is used to detect race conditions in s during runtime. If a race condition is found, it terminates with a counterexample in S. When an execution terminates ($s.enabled = \varnothing$ of line 3), we update the backtracking points for the entire trace. This is significantly different from the DPOR algorithm, which updates the backtracking points for each state s when it is pushed into the stack S. Rather than updating the backtracking points in the preorder of DFS as in DPOR, our algorithm waits until the information pertaining to an entire execution trace is available. In line 27, for each state t_d that is dependent and may be co-enabled with t, we check (in addition to that of DPOR) whether the search subspace from s_d is race-free. If the answer is yes, we can safely skip the backtracking points at s_d. Otherwise, we proceed in the same fashion as DPOR.

5.2.1 The Running Example

We show how the overall algorithm works on the example in Figure 5.1. Assume that the first execution trace is

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_6} s_6 \ldots s_9 \xrightarrow{a_{10}} s_{10} \ldots s_{13} \xrightarrow{b_3} s_{14} \xrightarrow{b_4} s_{15} \ldots \xrightarrow{b_9} s_{20} \xrightarrow{b_{11}} s_{21} \ ,$$

produced by lines 11-20 of Figure 5.6. Since $s_{21}.enabled = \emptyset$, the call PDPSEARCH($S, s_{21}$) executes lines 3-9. For every $s_b \in S$, we update the backtrack sets; we go through the stack in the following order: $s_0, s_1, \ldots, s_{21}$.

- For $s_0, \ldots, s_{10}$, there is no need to add a backtracking point, because (per line 27) there is no t_d from a thread different from $tid(t)$.
- For s_{13}, the enabled transition b_3:`z++` is dependent and may be co-enabled with $t_d = a_{10}$:`z++`. (We assume *lock-atomicity* by grouping variable accesses with protecting lock/unlock and regarding each block as atomic.) However, since the search subspace from s_8 is race-free, we do not add backtracking points at s_8.
- For s_{14}, the enabled transition b_4:`c=x` is dependent and may be co-enabled with $t_d = a_2$:`x++`. Since the search subspace from s_0 has a potential race condition

between a_6 and b_{10}, we set $s_0.backtrack = \{2\}$ to make sure that in a future execution, thread T_2 is scheduled at state s_0.

After this, PDPSEARCH(S, s_i) keeps returning for all $i > 0$ as indicated by lines 20-21. Since $s_0.backtrack = \{2\}$, PDPSEARCH(S, s_0) executes lines 16-20. The next execution starts from $s_0 \xrightarrow{b_1} s'$.

5.2.2 Proof of Correctness

The correctness of the overall algorithm is summarized as follows: First, any race condition reported by PDPSEARCH is guaranteed to be real.

Second, if PDPSEARCH returns without finding any race condition, the program is guaranteed to be race-free under the given input. Finally, PDPSEARCH always returns a conclusive result (either race-free or a concrete race) for terminating programs. If a program is nonterminating, PDPSEARCH can be used for bounded analysis as in CHESS [72]—to detect bugs up to a bounded number of steps. The soundness is ensured by the fact that it is concretely executing the actual program within its target environment. The completeness (for terminating programs) can be established by the following arguments: (1) the baseline DPOR algorithm as in [21] is known to be sound and complete for detecting race conditions; and (2) our trace-based lock set analysis is conservative in checking race-free subspaces. The procedure returns 'yes' only if no race condition can be reached by any execution in the search subspace.

5.3 Experiments

We have implemented the proposed method on top of our implementation of the DPOR algorithm, inside Inspect [97]. We use CIL [74] for parsing, whole-program static analysis, and source code instrumentation. Our tool is capable of handling multithreaded C programs written using the Linux POSIX thread library. The source code instrumentation consists of the following steps: (1) for each shared variable access, insert a request to the scheduler asking for permission to execute; (2) for each thread library routine, add a wrapper function that sends a request to the scheduler before executing the actual library

routine; (3) for each branch, add recording routines to notify about the branch begin and end, the shared variables and the lock updates in the other branch.

In order to control every visible operation, we need to identify the set of shared variables during the source code instrumentation. Shared variable identification requires a conservative static analysis of the concurrent program, e.g., pointer and may-escape analysis [82, 55]. Since this analysis [82] is an over-approximated analysis, our instrumentation is safe for intercepting all visible operations of the program. This ensures that we do not miss any bug due to missing identification of a shared variable. Similarly, when a whole program static analysis is either ineffective or not possible (due to missing source code) to identify the precise lock sets, during instrumentation, we resort to subsets of acquired locks and supersets of released locks.

5.4 Experimental Results

We have conducted experimental comparison of our new method with the baseline DPOR algorithm. The benchmarks are Linux applications written in C using the POSIX thread library; many are obtained from public domain including `sourceforge.net` and `freshmeat.net`. Among the benchmarks, `fdrd2` and `fdrd4` are variants of our running example. `qsort` is a multithreaded quick sort algorithm. `pfscan` is a file scanner implemented using multiple threads to search directories and files in parallel; the different threads share a dynamic queue protected by a set of mutex locks. `aget` implements a ftp client with the capability of concurrently downloading different segments of a large file. `bzip2smp` is a multithreaded version of the Linux application `bzip`. All benchmarks are accompanied by test cases to facilitate the concrete execution. Our experiments were conducted on a workstation with 2.8 GHz Pentium D processor and 2GB memory running Fedora 5.

Table 5.1 shows the experimental results. The first two columns show the statistics of the test cases, including the name and the number of threads. Columns 3-10 compare the two methods in terms of the runtime, and the number of executed transitions, and the number of completed execution traces. In the experiment, we set the time out bound as one hour. For DPOR, every completed trace (reported in Column 5) belongs to a distinct

Table 5.1. Comparing the performance of two race detection algorithms

Test Program		Runtime (s)		# of Trans (k)		# of Traces		Race-free	
name	thrd	dpor	PDP	dpor	PDP	dpor	PDP	chks	yes
fdrd2	2	3	1	2	0.6	89	14	88	75
fdrd4	2	3	3	10	4	233	68	232	165
qsort	2	17	8	12	8	4	1	2	2
pfscan	2	179	15	71	10	2519	182	398	217
pfscan2	2	3	1	1	1	31	10	5	5
aget	3	183	1	103	0.1	3432	1	6	6
aget	4	>1h	1	-	0.1	-	1	9	9
aget	5	>1h	1	-	0.1	-	1	12	12
bzip2smp	4	128	3	63	2	1465	45	48	5
bzip2smp	5	203	4	99	2	2316	45	48	5
bzip2smp	6	287	4	135	2	3167	45	48	5
bzip2smp2	4	291	136	63	21	1573	45	48	5
bzip2smp2	5	487	155	85	21	2532	45	48	5
bzip2smp2	6	672	164	116	21	3491	45	48	5
bzip2smp2	10	1435	183	223	21	7327	45	48	5

equivalence class of interleavings; however, many of them are pruned away by PDP since they are redundant as far as race detection is concerned. Columns 9-10 provide the number of race-free checks and the number of race-free check successes.

The results show that our PDP method is significantly more efficient than DPOR in pruning the search space. For all examples, PDP took significantly less time in either finding the same data race or proving the race freedom; the number of transitions/traces that PDP has to check during the process was also significantly smaller. Although the average time for PDP to complete one execution is longer than DPOR, e.g., 4066 ms vs. 195 ms as indicated by data from the last row of Table 5.1 (due to the overhead of tracking branch begin/end and other auxiliary transitions), the overhead in PDP is well compensated by the skipped executions due to property driven pruning.

5.5 Summary

We have proposed a new data race detection algorithm that combines the power of dynamic model checking with property driven pruning based on a lock set analysis. Our method systematically explores concrete thread interleavings of a program, and at the same time, prunes the search space with a trace-based conservative analysis. It is both sound and complete (as precise as the DPOR algorithm); at the same time, it is significantly more efficient in practice, allowing the technique to scale much better to real-world applications. For future work, we would like to extend the proposed framework to check other types of properties. Since race detection is a problem of simultaneous reachability of two transitions, the techniques developed here should be readily applicable to checking deadlocks and many other simple safety properties.

CHAPTER 6

DISTRIBUTED DYNAMIC PARTIAL ORDER REDUCTION

Stateless runtime model checking, which is pioneered by Verisoft [25], is a promising verification methodology for real-world threaded software. It avoids the (implicit or explicit) overhead of modeling programs that is usually required by other model checkers [39, 81, 38, 3]. The precision of information available at run-time allows techniques such as dynamic partial order reduction (DPOR) [21] to dramatically cut down the number of interleavings examined.

Unfortunately, even with dynamic partial order reduction enabled, runtime is still a major limiting factor of stateless model checkers [72]. With the wide availability of computer clusters, it is desirable to use them to speed up the stateless model checking.

We observed that since stateless search does not maintain the search history, different branches of an acyclic state space can be explored concurrently with very loose synchronizations. This shows that stateless runtime model checkers are potentially "embarrassingly parallel" for distributed verification. We implemented a parallel stateless model checker based on this observation, employing a centralized load balancer to distribute work among multiple nodes. Initially, we failed to consistently obtain the linear speedup promised by the apparent parallelism. Deeper investigation revealed the reasons. These reasons, and other features of our algorithm are now summarized:

- **Avoiding Redundant Computations:** Despite our use of sleep sets [24] to avoid redundant interleavings among independent transitions, we found that redundant (and, in fact, identical) interleavings were being explored among multiple nodes. The problem was traced to the incremental way of computing backtrack sets in the DPOR algorithm (detailed in the rest of this paper), which is well suited for

a sequential implementation but not a loosely synchronized distributed implementation. We have developed a heuristic technique to update backtrack sets more aggressively, as detailed in Section 6.1.4.

- **Work Distribution Heuristics**: Numerous heuristics help achieve efficient work distribution in the parallel stateless model checker. These include (i) the straightforward method of employing a single load balancing node (process) and $N - 1$ worker nodes (processes), (ii) the concept of a soft limit on the number of backtrack points recorded within a worker node before that node decides to offload work to another worker, and (iii) minimizing communication by offloading work that lies deepest within the stack – points from where the largest number of program-paths are available – so that bigger chunks of work are shipped per communication.

In the rest of this chapter, we present the distributed dynamic partial order reduction (DDPOR) algorithm in detail. We implemented DDPOR on top of our stateless runtime model checker `Inspect` [97]. Our experiments show almost linear speedup with increasing number of nodes (CPUs). For example, one of our benchmarks that has eight threads and requires more than 11 hours to finish checking using sequential `Inspect` can be checked by the parallel `Inspect` within 11 minutes using 65 nodes. The parallel `Inspect` gives a speedup of 63.2 out of 65.

The rest of this chapter is organized as follows: Section 6.1 presents the DDPOR algorithm; Section 6.2 presents implementation detail, and the experiment results; Section 6.3 the related work; and Section 6.4 our concluding remarks.

6.1 Algorithm

In DPOR, the thread identities recorded in the backtrack set of a state s help generate different (nonequivalent) executions out of s. As DPOR is implemented through stateless search, it is completely safe to explore the different transitions in the backtrack sets of states concurrently, and with no (or very little) synchronization. With the wide availability of cluster machines, the potential for distributed verification is very high.

To have multiple nodes explore multiple backtrack points concurrently, each cluster node must know (i) the portion of the search stack from the initial state to the backtrack

point, (ii) the transition sequence from the initial state to reach the backtrack point, and (iii) the transition to be executed from a backtrack point. All this information is easily obtained from the search stack. To distribute work among multiple nodes, we can use a centralized load balancer to balance the work, employing very limited synchronizations.

The computer nodes in a cluster can be classified into three categories: (i) the load balancer, (ii) *busy worker* nodes that have been assigned tasks, and (iii) *idle worker* nodes that are waiting for tasks. Figure 6.1 illustrates how the workers and the load balancer collaborate. Let a be a busy worker node and b an idle one, with a trying to unload some work to b. First a sends a request to the load balancer. If there are idle nodes, the load balancer will return the identity of an idle node to the worker. In our example, the load balancer tells a that b is idle, whereupon node a will send an unload message to b with all the information needed for b to start searching from an unexplored backtrack point. When b finishes the assigned work, it sends a report to the load balancer.

In the rest of this section, we first present the load balancing algorithm (Section 6.1.1) and the computation of each worker (Section 6.1.2). Then, the DDPOR algorithm is presented over Sections 6.1.3 and 6.1.4.

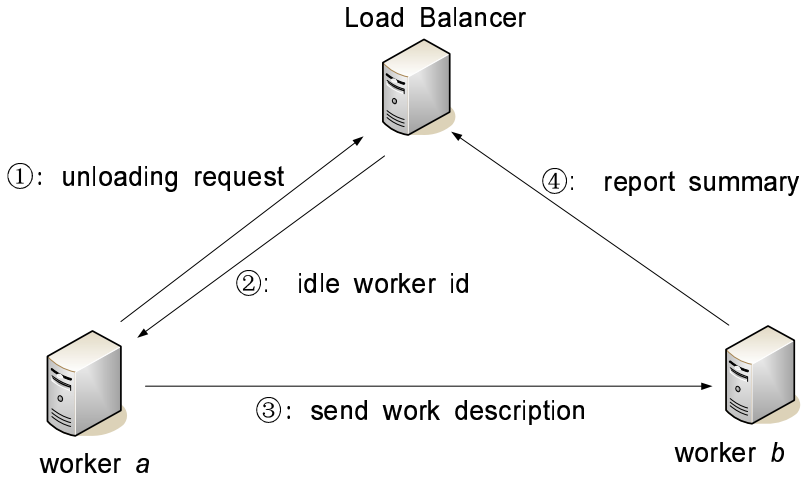


Figure 6.1. The message flow among the load balancer and the workers

6.1.1 Load Balancing

For simplicity, we assign one node of an N-node cluster as the centralized load balancer and the rest of $N-1$ nodes as workers. The load balancer monitors the status of all workers for the purpose of partitioning the workload. That is, we have one node execute the procedure LOADBALANCER (Figure 6.2), and the rest of the nodes execute the procedure WORKER (Figure 6.3).

We use W_b to denote the set of busy workers, and W_i to refer to the set of idle workers. Initially, all worker nodes are idle, and the load balancer randomly picks an idle worker w_0, and sends a message to w_0 to have it start checking the program (line 4-7 of Figure 6.2) . After this, the local balancer adds w_0 to the busy workers set W_b, and removes w_0 from the idle workers set W_i (line 8-9 of Figure 6.2). Then it keeps waiting for messages from busy workers until no work node is busy (line 10-26 of Figure 6.2). When the load balancer finishes the `while` loop, all workers must have finished exploring their part of state space, which means the whole state space has been explored. At this stage, the load balancer sends a termination message to every worker to terminate them and exit.

There are two categories of messages that the load balancer can receive from the workers:

- requests from busy workers to unload some work to idle workers.
- reports from busy workers after they finish exploring the assigned state space.

While exploring the assigned state space, if a worker ends up having more than a certain number of backtrack points in its stack, it implies that too much work might have been assigned to this worker. In this situation, this worker sends a work unloading request to the load balancer. If there are idle workers available, the load balancer passes along the idle worker's information (line 21-24 of Figure 6.2). Otherwise, it tells the requester that there are no idle workers available (line 17-20 of Figure 6.2).

```
1: Initially: W_b = ∅,
2:            W_i = { all worker nodes in the cluster };

3: LOADBALANCER(){
4:   let w_0 be a worker node such that w ∈ W_i;
5:   let S be an empty program state stack;
6:   S.push(s_0);
7:   send S to w_0;
8:   W_i ← W_i \ {w_0};
9:   W_b ← {w_0};
10:  while(W_b ≠ ∅)
11:  {
12:    receive event e from any worker w;
13:    if(e is work finish notification){
14:      W_b ← W_b \ {w};
15:      W_i ← W_i ∪ {w};
16:    }
17:    else if(e is new work request){
18:      if(W_i = ∅){
19:        reply "no idle workers" to w;
20:        continue;
21:      }
22:      let w' be a worker node such that in w' ∈ W_i;
23:      W_i ← W_i \ {w'};
24:      W_b ← W_b ∪ {w'};
25:      tell w that w' is idle;
26:    }
27:  }
28:  for each w ∈ W_i
29:    send a termination message to w;
30: }
```

Figure 6.2. The load balancing algorithm

```
1: WORKER(){
2:   while (true){
3:     let m be a received message;
4:     if (m is a command to terminate) return;
5:     receive the search stack S;
6:     DDPOR(S);
7:     send the report to the load balancer;
8:   }
9: }
```

Figure 6.3. The routine that runs on each worker

6.1.2 Worker Routine

Figure 6.3 shows the routine that runs on the worker nodes. The main body of the routine is a while loop. Each worker keeps passively waiting for work unloading messages, and has DDPOR-enabled depth-first search for each assigned state space (line 5-7 of Figure 6.3). The worker exits the while loop and terminates when a termination message is received (line 4 of Figure 6.3).

6.1.3 Distributed DPOR

Figure 6.4 shows the DDPOR algorithm. Comparing with the original DPOR algorithm in Figure 2.1, we made the following changes:

- add work unloading primitives (line 5 of Figure 6.4).
- to avoid the redundant exploration of the state space among multiple nodes, we use DDPORUPDATEBACKTRACKSETS to compute the backtrack points in a different way from the original DPOR algorithm. We will present the details in Section 6.1.4.

In DDPOR, each time after updating the backtrack points, we check whether the number of backtrack points in the search stack has exceeded a value n (line 5 of Figure 6.4). Here n is the number of backtrack points in the search stack. If so, the current node decides to unload some of this excess work to the other nodes, as captured in procedure UNLOADWORK.

```
1:  Initially: S is received from the work assigner;

2:  DDPOR(S){
3:    let s = S.top;
4:    for each t ∈ s.enabled, DDPORUPDATEBACKTRACKSETS(S, t);
5:    if (there are more than n backtrack points in the S) UNLOADWORK(S);
6:    if (∃p ∈ Tid : ∃t ∈ s.enabled : tid(t) = p) {
7:      s.backtrack ← {p};
8:      s.done ← ∅;
9:      while(∃q ∈ s.backtrack \ s.done){
10:       s.done = s.done ∪ {q};
11:       s.backtrack = s.backtrack \ {q};
12:       let t ∈ s.enabled such that tid(t) = q, and let s' be a state such that s -t-> s';
13:       S.push(s');
14:       DDPOR(S);
15:       S.pop();
16:     }
17:   }
18: }

19: UNLOADWORK(S) {
20:   send a work unload request to the load balancer;
21:   receive reply r from the load balancer;
22:   if(r shows no idle node available) return;
23:   let w_i be the idle worker node that the load balancer tells this node;
24:   let s be the deepest state in the search stack S such that s.backtrack ≠ ∅;
25:   let S_s be a copy of the sequence of states from s_0 to s;
26:   let s' be the last state in S_s (i.e. s' is a copy of s), and let τ ∈ s'.backtrack;
27:   s'.backtrack ← {τ};
28:   s'.done ← s'.done ∪ (s.backtrack \ {τ});
29:   send S_s to w_i;
30:   s.backtrack ← s.backtrack \ {τ};
31:   s.done ← s.done ∪ {τ};
32: }
```

Figure 6.4. Distributed dynamic partial order reduction

The UNLOADWORK routine first checks with the load balancer to see if there are any idle nodes. If not, the routine will return immediately (line 22 of Figure 6.4). Otherwise, it send the search stack S_s to the idle node w_i (line 24-29 of Figure 6.4). The algorithm in Figure 6.4 does the unload work request each time it enters the DPOR routine. This may lead to repeated failures if there are no idle nodes available for a while (not observed in our experiments). Various heuristic solutions are possible in case it arises in practice (e.g., send aggregated requests more infrequently).

To derive the most benefit per exchanged work unloading message, we observe that backtrack points situated deeper in the stack typically have larger numbers of program-paths emanating from them. Based on this heuristic, we choose the deepest state s in the search stack that satisfies $s.backtrack \neq \varnothing$ (line 24 of Figure 6.4). After unloading a backtrack point from s, on the current node, we will put the thread identity of the transition in $s.done$ to avoid it being explored by the current node (line 29-30 of Figure 6.4).

6.1.4 Updating the Backtrack Set

In dynamic partial order reduction, the persistent set of a given state is computed dynamically. Procedure UPDATEBACKTRACKSETS in Figure 2.1 shows how the backtrack points are computed. One problem we encountered with the procedure UPDATEBACKTRACKSETS is that with more than two threads, it may result in redundancy exploration of the same branch in parallel mode.

The example in Figure 6.5 illustrates this problem. The program has three threads, all of which first acquire the global lock t, and then release the lock. Obviously, there are $3! = 6$ different interleavings for this concurrent program with DPOR.

Assume we use a computer cluster that has only two worker nodes. We also assume that the bound n in Figure 6.4 for unloading is one. Let the two workers be n_0 and n_1, and let the three threads be t_0, t_1 and t_2. Figure 6.6, Figure 6.7 and Figure 6.8 show how the work would be distributed between the two nodes if we follow the UPDATEBACKTRACKSETS routine shown in Figure 2.1.

```
global: mutex  t;

thread t0:          thread t1:        thread t2:

lock(t);            lock(t);          lock(t);
unlock(t);          unlock(t);        unlock(t);
```

Figure 6.5. A simple example on the potential redundant search among nodes

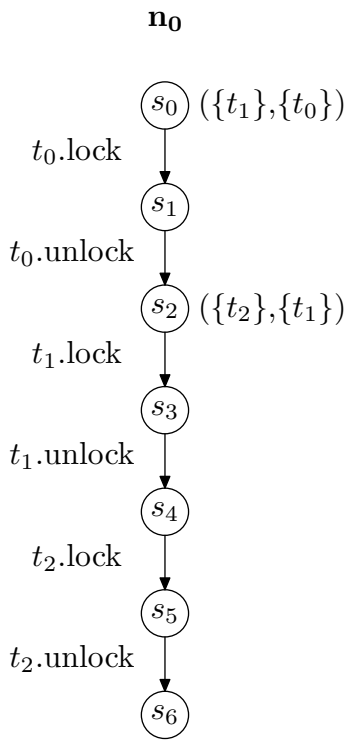


Figure 6.6. An initial trace of the program.

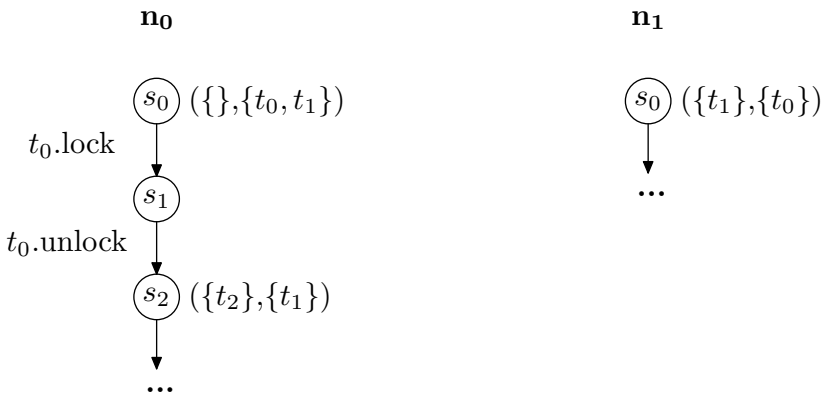


Figure 6.7. Distributing the task between two nodes

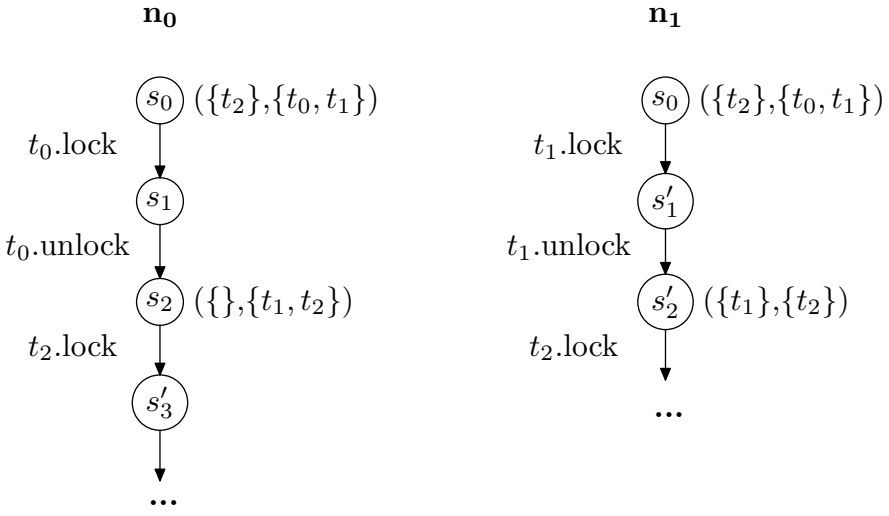


Figure 6.8. An example on the redundant backtracking points

Let n_0 start concretely executing the program first, and n_1 is idle. When n_0 reaches the end of its trace, we can observe the interleaving of three threads as in Figure 6.6. Here, two backtrack points at s_0 and s_2 have been recorded. When the work node n_0 detects this (i.e., more than one backtrack point in the search stack), it will send a request to the load balancer for unloading work. First the load balancer will tell n_0 that n_1 is idle. Second, n_0 will send the search stack to n_1, following the UNLOADWORK routine in Figure 6.4. Then the work node n_1 will receive the message and be ready for exploring the state space assigned to it. The left half of Figure 6.7 captures this scenario.

At this point, with respect to the situation in Figure 6.7, n_0 will explore transition $t_2.lock$ from the backtrack point s_2, while n_1 will explore transition $t_1.lock$ from s_0. Both nodes will update the backtrack information according to their own search stacks. The scenario in Figure 6.8 results, in which both n_0 and n_1 compute and place t_2 in $s_0.backtrack$ whose transition should be explored from s_0. This will result in redundant explorations being conducted by n_0 and n_1. In the worst case, this kind of redundancy may have all the workers explore the same interleaving, and result in little or no speedup (our experiments shown in Section 6.2 confirms this).

This problem is caused by the algorithm shown in Figure 2.1 computing $s.backtrack$ incrementally with respect to state s. In DDPOR, when a worker unloads work to some idle node, it is possible that the full backtrack set has not yet been associated with states in the copy of the stack being passed along. To solve this problem, given a state s, one must attempt to compute all transitions associated with $s.backtrack$ as aggressively as possible. The procedure UPDATEBACKTRACKSETS of Figure 2.1 only updates the latest state in the search stack from which the enabled transition is dependent and may be co-enabled with the next transition (Line 25-32 in Figure 2.1).

The backtrack sets updating routine in the distributed context, DDPORUPDATEBACKTRACKSETS, is shown in Figure 6.9. For each to be executed transition t, the new routine will check the stack to find all states from which a dependent and may be co-enabled transition was executed (Line 5 of Figure 6.9), and update the correspondent backtrack set. With the new routine, we will get the distributed scenario as shown in Figure 6.10

```
1: DDPORUPDATEBACKTRACKSETS(S, t) {
2:   let T be the transition sequence associated with S;
3:   for each(t_d ∈ T that is dependent and may be co-enabled with t){
4:     let s_d be the state in S from which t_d is executed;
5:     let E be {q ∈ s_d.enabled | tid(q) = tid(t), or q in T, q was after t_d and there is
       a happens-before relation for (q, t). }
6:     if (E ≠ ∅)
7:       choose any q in E, add tid(q) to s_d.backtrack;
8:     else
9:       s_d.backtrack = s_d.backtrack ∪ {tid(q) | q ∈ s_d.enabled};
10:  }
11: }
```

Figure 6.9. Updating the backtrack sets of states in the distributed context

and Figure 6.11. Note that this is only a heuristic; we do not know of a way to retain loose synchronizations between the threads and still avoid this redundancy.

6.1.5 Correctness

The soundness of the DDPOR algorithm follows from the fact that the parallel algorithm is guaranteed to compute at least all the backtrack set entries computed by the sequential algorithm for every state. We alter only where this information is computed.

6.2 Implementation and Experiments

We implemented the DDPOR on top of `Inspect` [97] using MPI [87, 49]. MPI (Message Passing Interface) is a message-passing library specification, designed to ease the use of message passing by end users. It is supported by virtually all supercomputers and clusters, and is the de facto standard of high performance computing.

We used the MPI routines `MPI_Send` and `MPI_Recv` for communication among nodes.

- `MPI_Send(obj, dest_node, msg_label)` sends `obj` to `dest_node`, and the message is labeled `msg_label`. Here `obj` is an object of any type, `dest_node` is a node of the cluster, `msg_label` is the label message.

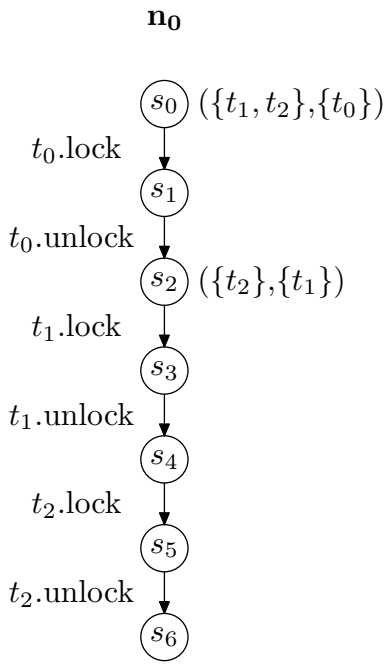


Figure 6.10. The intial trace of the program with DDPORUPDATEBACKTRACKSETS

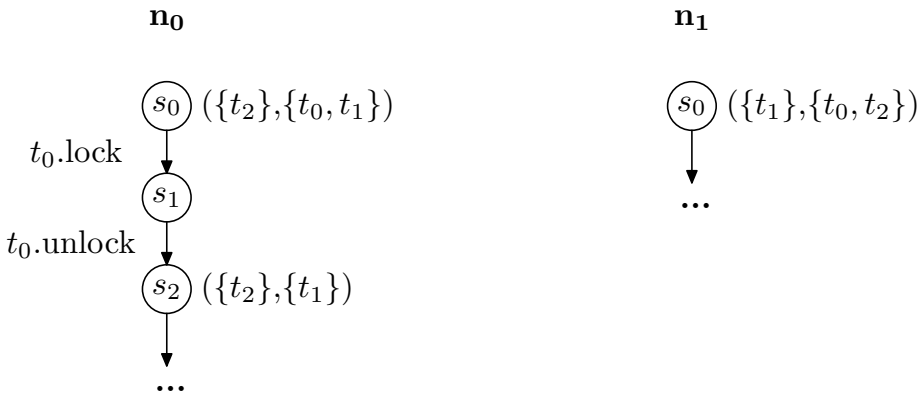


Figure 6.11. Distributing tasks with DDPORUPDATEBACKTRACKSETS

- `MPI_Recv(src_node, msg_label)` returns the message sent by the `src_node` to the current one with the label `msg_label`. We can set the `src_node` to `ANY_SOURCE` to retrieve the message without checking which node is the sender.

One interesting problem we encountered while we implemented the parallel `Inspect` is that the cluster's network file system can be a bottleneck for a parallel runtime checker if there are disk write operations in the program under test. This problem can be easily avoided by using the local disks.

We conducted our experiments on a 72-node cluster with 2GB memory and two 2.4GHz Intel XEON processors on each node. We compiled the program with gcc-4.1.0 and -O3 option. We used LAM-MPI 7.1.1 [48] as the message passing interface. The runtimes that we report are the average runtimes calculated over three runs.

Table 6.1 shows some benchmarks we have used to test the parallel `Inspect`. In Table 6.1, the second column is the number of threads in each benchmark, the third column shows the number of runs needed for runtime checking the program, and the last column shows the time that the sequential `Inspect` needs for checking the program.

The first two benchmarks, *indexer* and *fsbench*, are from [21]. *Indexer* captures the scenarios in which multiple threads insert messages into a hash table concurrently. *Fsbench* is an abstraction of the synchronization idiom in Frangipani file system. The third benchmark, *aget* [43], is an ftp client in which multiple threads are used to download different segments of a large file concurrently. The last benchmark, *bbuf*, is an implementation of a bounded buffer with four producers and four consumers that have put/get operations on it.

Table 6.1. Checking time with the sequential `Inspect`

benchmark	threads	runs	check using sequential `Inspect`(sec)
fsbench	26	8,192	291.32
indexer	16	32,768	1188.73
aget	6	113,400	5662.96
bbuf	8	1,938,816	39710.43

Indexer and *fsbench* are relatively small benchmarks. Using one node in the cluster, the sequential `Inspect` takes about 25 minutes to check *indexer*, and 5 minutes to check *fsbench*. Using parallel `Inspect` and at most 65 nodes (one node as the load balancer and 64 worker nodes), we can check both of them within 40 seconds. As the state spaces of these two benchmarks are relatively small, with the number of worker nodes increasing, the communication overhead increases more rapidly than the time reduction we get from distributing the work to more nodes. As a result, we see a degradation of speedup when we use more than 52 nodes to do parallel checking for *indexer*, and more than 48 nodes for *fsbench*.

Figure 6.12 shows the speedup we got using the parallel `Inspect` against the sequential `Inspect` on *indexer* and *fsbench*. As the performance of using DDPORUPDATEBACKTRACKSETS in Figure 6.9 does not differ significantly from using the original UPDATEBACKTRACKSETS in Figure 2.1, we do not show the comparison in Figure 6.12.

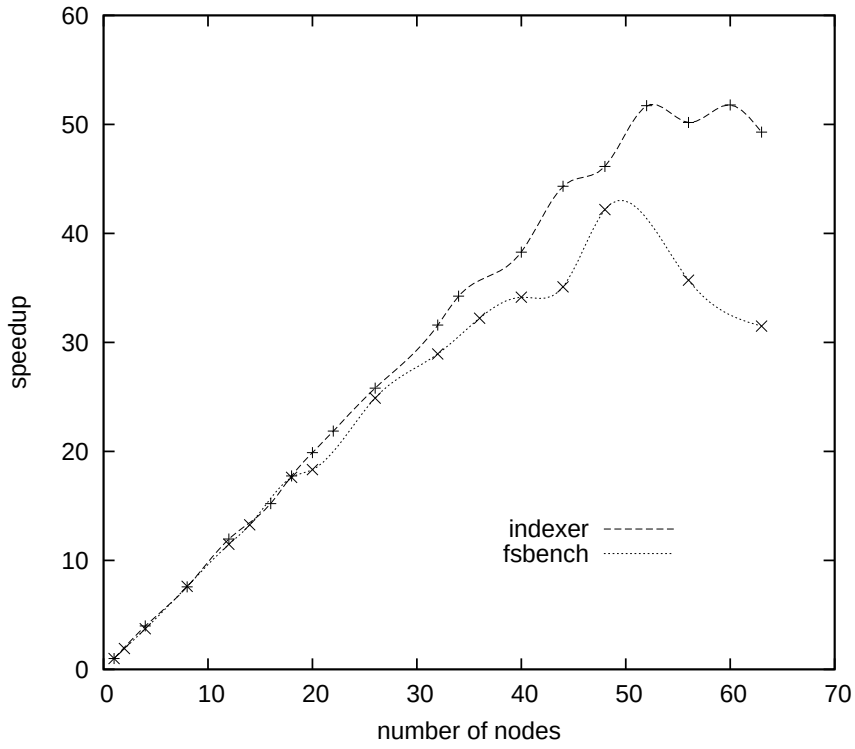


Figure 6.12. Speedups on *indexer* and *fsbench*

Figure 6.13 shows the speedup we got using the parallel `Inspect` on *bbuf*. The sequential `Inspect` needs more than 11 hours to finish checking the program. During this period of time, `Inspect` needs to rerun the program for more than 1.9 million times. As shown in Figure 6.13, the parallel `Inspect` can give us almost linear speedup. It turns out that we can get a speedup of 63.2 out of 64 worker nodes (totally 65 nodes, including the load balancer), and reduce the checking time to 11 minutes. In this figure, we also show the comparison between the speedup we got using DDPORUPDATEBACKTRACKSETS or UPDATEBACKTRACKSETS in DDPOR to update the backtrack sets of states (line 4 of Figure 6.4). As we can see, without the aggressive backtrack sets updates in DDPORUPDATEBACKTRACKSETS,we get little speedup while the number of nodes increases.

Figure 6.14 shows the speedup using the parallel `Inspect` on *aget*. There are data races in the original *aget*. We fixed those data races and did experiments on the fixed

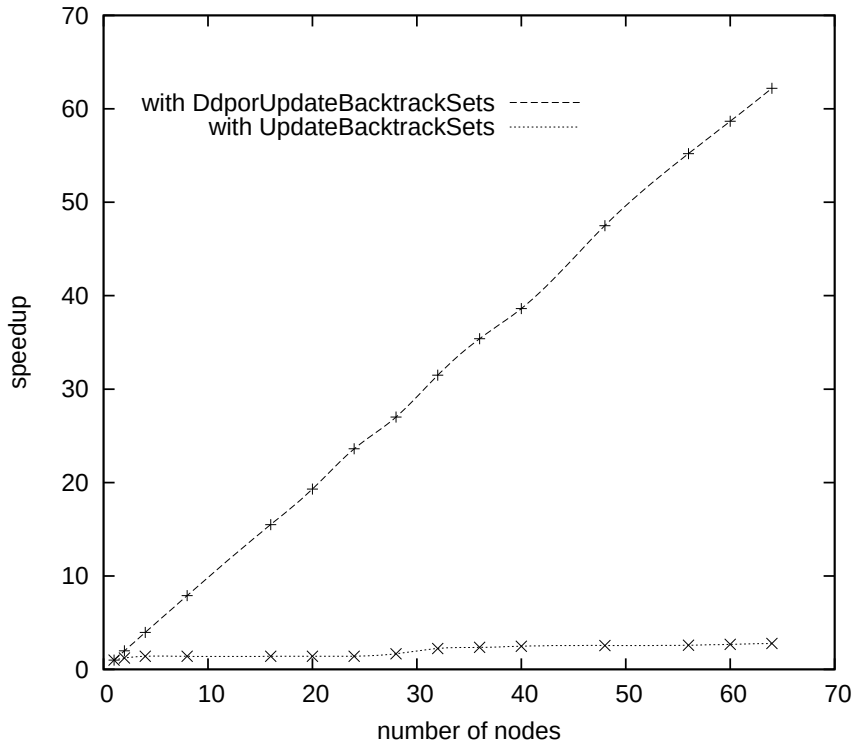


Figure 6.13. Speedups on the bounded buffer example

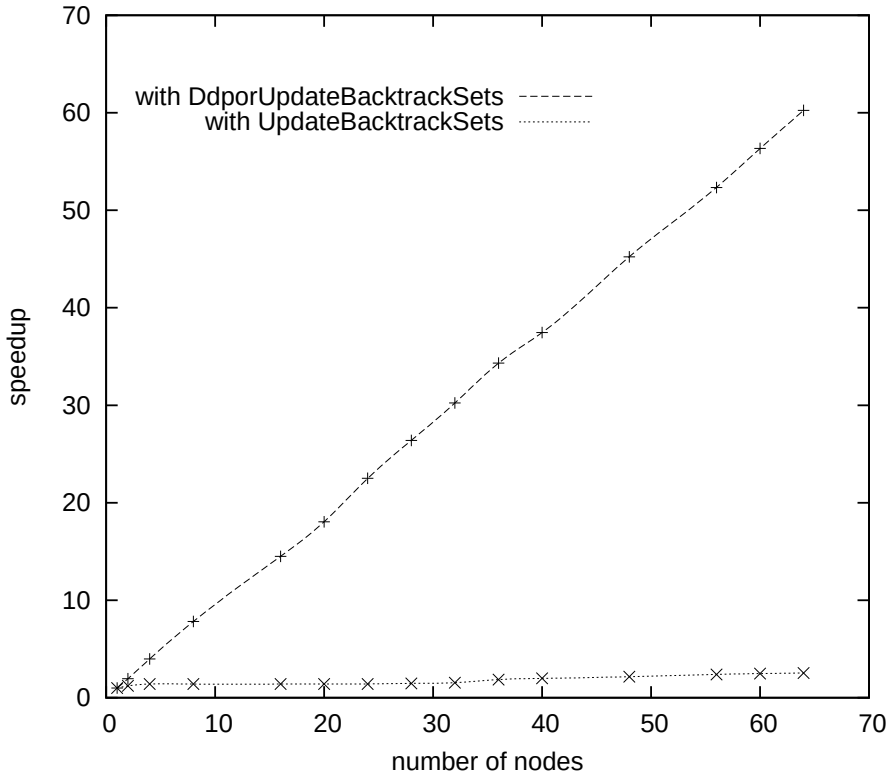


Figure 6.14. Speedups on the aget example

version. We reduced the size of the data package, which *aget* gets from the ftp server, to 512 bytes, to avoid the nondeterminism introduced by the network environment. The result again confirms that parallel `Inspect` can give out almost linear speedup, and our extension on the original DPOR is efficient.

6.3 Related Work

Parallel and distributed model checking has been a topic of growing interest, with a special conference series (PDMC) devoted to this topic. An exhaustive literature survey is beyond the scope of this paper. Quite a few distributed and parallel model checkers based on message passing have been developed for Murphi and SPIN [88, 69, 86, 57, 40]. Stern and Dill [88] developed a parallel Murphi that distributes states to multiple nodes for further exploration according to the state's signature. They pointed out the idea of coalescing states into larger messages for better network utilization in the context of

model checking. Eddy [69] extended the work and studies the parallel and distributed model checking under the multicore architecture. Kumar and Mercer [57] improved the load balancing method in parallel Murphi. Recently, Holzmann and Bosnacki [40] designed a multicore model checking algorithm to improve SPIN to fully utilize the multicore chips.

Brim et al. [5] proposed a distributed partial order reduction algorithm for generating a reduced state space. The algorithm exploits features of the partial order reduction which makes the idea of distributed DFS-based algorithm feasible. Palmer et al. [77, 76] propose another distributed partial order reduction algorithm based on the two-phase partial order reduction algorithm.

As far as the authors know, our work is the first effort on using parallelism to speed up runtime model checking for multithreaded programs.

6.4 Summary

Checking time has been the major bottleneck for stateless runtime model checkers. We propose a distributed dynamic partial order reduction algorithm to use parallelism to speed up the stateless model checking. Our experiments confirm that this algorithm scales well on a wide variety of nodes. It can give out almost linear speedup compared with the sequential stateless model checker.

CHAPTER 7

AUTOMATIC TRANSITION SYMMETRY DISCOVERY

Dynamic model checking [25, 72, 97] methods have proven promising for revealing errors in the implementation of real-world concurrent programs. They work on real applications and libraries, and side-step the high complexity of model construction and state capture by concretely executing the programs, and replaying the executions for covering the different thread interleavings. Although several techniques have been proposed for reducing the complexity of dynamic model checking [21, 99, 94], one important technique – namely *symmetry reduction* – has yet to be fully explored during dynamic model checking.

In contexts where models of the systems are verified, symmetry reduction has already been shown highly beneficial for reducing the search space [10, 19, 51]. The basic approach taken in these works is finding and exploring the state space of a *quotient* transition system defined by an underlying equivalence relation – usually over the system states. This approach does not work well during dynamic model checking because the "state" consists of the runtime state of the threads being subject to concrete execution: capturing and canonicalizing such states is practically difficult or impossible. In dynamic model checking of concurrent programs, transition symmetry [26] has been used as an effective way of pruning of the search space; however, it requires the user to provide a permutation function in order to check whether two transitions are symmetric. In this paper, we present an algorithm to automate transition symmetry discovery by employing dynamic analysis. To our best knowledge, this is the first approach to automate transition symmetry discovery for the dynamic verification of realistic concurrent programs.

We first note that purely static-analysis-based approaches are often insufficient for detecting symmetries across threads. To illustrate this point, consider a seemingly simplistic case where multiple threads are created out of the same thread routine with the same parameters; one may be tempted to conclude symmetry at the starting points of these threads. However, this may not be true because even if two threads share the same thread routine, there may be a state from which they execute along different paths of the thread routine (depending on the external input and the thread identity), in which case they are not symmetric. Figure 7.1 shows such an example, in which a thread can act as either a producer or a consumer, depending on its thread identity. Purely static analysis has well-known limitations in tracking such control flows accurately (of course, doing these computations exactly invites undecidability). Although it is possible to algorithmically infer that two threads share the same set of control flow paths, one still needs to take the concrete local states of threads into consideration in order to decide, in any particular program state, whether symmetric reduction can be applied. Furthermore, the criterion for whole thread symmetry, i.e., requiring two threads to be created from the same thread routine with the same parameters, is very limiting. In practice, threads can be created by different routines and yet still have transition symmetry.

In this chapter, we introduce a new algorithm for revealing symmetry in multithreaded programs and show how it can be combined seamlessly with dynamic partial order reduction (DPOR) [21] for more efficient dynamic model checking. We use dynamic program

```
thread tid:
  if (tid%2 == 0)
  {
    ...produce...
  }
  else
  {
    ...consume...
  }
```

Figure 7.1. Threads that are forked from the same routine may behave differently

analysis to discover symmetry in multithreaded programs; that is, instead of statically analyzing the program, we perform program analysis while concretely executing the program.

In more detail, we compute the *residual code* of threads while executing the program. The residual code of a thread τ at a state s is the code that may be executed by τ from s before τ's termination. If we find that the residual code of threads can be put into *syntactic* equivalence under a bijection over thread local variable names and label names, we further check whether the local states of threads can also be put into equivalence under this bijection. We shall prove that, in a state s of a multithreaded program, if the residual code of two threads at s are syntactically equivalent under a certain bijection of thread local variables and labels, and the local states of threads in s are identical under the same bijection, then there is a transition symmetry for the enabled transitions of the two threads at s (details in Section 7.2). Since transition symmetries induce an equivalence relation on states of the concurrent system, during model checking, we can discard a state s, or backtrack in the context of stateless search, if an equivalent state s' has been explored before. As pointed out in [26], transition symmetry reduction is orthogonal to, and can be combined with, partial order reduction techniques for safety verification.

We implemented this symmetry discovery algorithm on top of our dynamic model checker `Inspect` [97, 47]. Our experiments show that our symmetry discovery algorithm can successfully discover transition symmetries that are hard or otherwise cumbersome to identify manually, and can significantly reduce the checking time for realistic multithreaded applications.

The rest of this chapter is organized as follows. In Section 7.2, we present our algorithm for discovering symmetry in multithreaded programs. In Section 7.3, we show how to combine the symmetry discovery algorithm with a popular algorithm of dynamic partial order reduction (DPOR). In Sections 7.4 , we explain the implementation details and present our experimental results. We review related work in Section 7.6 and then summarize this chapter in Section 7.7.

7.1 A Simple C-like Programming Language

Taking the source code of a multithreaded program into consideration, a transition t is a sequence of statements $[m_1, m_2, \ldots, m_k]$ of the program, in which the first statement contains a visible operation on the global objects, and the rest of the statements are invisible operations on local objects of the same thread.

To simplify presentation, in this paper, we focus on a C-like simple programming language as shown in Figure 7.2. Nevertheless, the theorems we prove in this paper can be extended to accommodate the entire C language. In our actual implementation, we support the entire C language.

As shown in Figure 7.2, a program is composed of a set of threads. Each thread is a sequence of statements. A statement can be an assignment, a branch statement, or a function call. A label may be associated with a statement.

We use $\eta(s,t)$ to denote the list of statements that are exercised when a transition t is executed from the state s. We use $VL(\eta(s,t))$ to denote the set of variables and labels that appear in $\eta(s,t)$.

Let $\theta : VL(\eta(s,t)) \rightarrow VL'$ be a mapping from $VL(\eta(s,t))$ to another set of variables and labels VL'. We use $\eta(s,t)[\theta]$ to denote the statement list that we get by replacing each variable and label $v \in VL(\eta(s,t))$ with $\theta(v)$.

Definition 7.1 (syntactically equivalent transitions) Let t_1 and t_2 be two transitions of a transition system M that are enabled at state s_1 and s_2, respectively. We say t_1 and t_2

$$
\begin{array}{rcl}
P & ::= & thread^* \\
thread & ::= & Stmt^* \\
Stmt & ::= & [l:]s \\
s & ::= & lhs \leftarrow e \mid \texttt{if } lhs \texttt{ then goto } l' \mid f(params) \mid lhs \leftarrow f(params) \\
params & ::= & lhs^* \\
lhs & ::= & v \mid \&v \mid *v \\
e & ::= & lhs \mid lhs \ \diamond \ lhs \\
 & & where \diamond \in \{+, -, *, /, \%, <, >, \leq, \geq, \neq, =, ...\}
\end{array}
$$

Figure 7.2. Syntax of a simple language that is similar to C

are syntactically equivalent if there exists a bijection $\theta : VL(\eta(s_1, t_1)) \rightarrow VL(\eta(s_2, t_2))$ such that $\eta(s_1, t_1)[\theta] = \eta(s_2, t_2)$.

7.1.1 Residual Code of Threads

Let s be a state of a multithreaded program. Let τ be a thread that is enabled in s. The residual code of τ at s is the statement that may be executed by τ from s before its termination. We write $\mathcal{C}(s, \tau)$ to denote this.

The residual code of a thread τ at s captures the code paths that τ may take starting from s. Take the thread shown in Figure 7.3(a) as an example. Let the thread be at the point that it finishes line 1 and is going to execute line 2. Assume that the condition at line 2 is false in the state s. If we are able to infer that this condition is false, we can conclude that line 2-5 is the residual code of this thread. Otherwise, we can consider line 1-5, which is an overapproximation of line 2-5, as the residual code.

Figure 7.3(b) shows a thread that calls a recursive function. Let the execution context of the thread be $[\texttt{f}(0); \texttt{f}(1)]$, and the thread is in line 5, right before the third call to `f`. In this thread, the parameter `b` has multiple instances in different frames of the call stack, we use b_i to differentiate them. The residual code of the thread at this point is $\langle \texttt{f}(\texttt{b}_2);\ \texttt{print}(\texttt{b}_2);\ \texttt{print}(\texttt{b}_1) \rangle$. This can be easily computed by dynamically capturing the execution context of the thread and having an intraprocedural analysis for each function call in the execution context.

```
          thread:
1:   L1: a++;
2:        if (a < 5){
3:          goto L1;
4:        }
5:        local = a;
```

(a) a thread that has a branch statement

```
     thread:
1: f(int b){
2:   if (b > 2)
3:     return;
4:   b = b + 1;
5:   f(b);
6:   print(b);
7: }
```

(b) a thread that starts with $\texttt{f}(0)$

Figure 7.3. Examples on the residual code of threads

Let $c = \mathcal{C}(s, \tau)$ be the residual code of thread τ at s. We use $VL(c)$ to denote the set of variables and labels that appears in c. Let $\theta : VL(c) \rightarrow VL'$ be a mapping from $VL(c)$ to another set of variables and labels. We use $c[\theta]$ to denote the code that we get by replacing each occurrence of variables and labels $v \in VL(c)$ with $\theta(v)$. Let $l_\tau(s)$ be a local state of thread τ at s. We use $l_\tau(s)[\theta]$ to denote a new local state we get by renaming each variable v that appears in $l_\tau(s)$ with $\theta(v)$.

Definition 7.2 (syntactically equivalent residual code) Let $c_1 = \mathcal{C}(s_1, a)$ and $c_2 = \mathcal{C}(s_2, b)$ be the residual code of thread a at s_1 and thread b at s_2, respectively. We say that c_1 and c_2 are syntactically equivalent if there is a bijection $\theta : VL(c_1) \rightarrow VL(c_2)$ such that $c_1[\theta] = c_2$. We denote this with $c_1 \overset{\theta}{\sim} c_2$.

7.1.2 Symmetry

The main idea of symmetry reduction [10, 19, 51] is that symmetries in the system induce an equivalence relation on states of the concurrent system. While performing model checking, one can discard a state s if an equivalence state s' has been explored before. Here we briefly review the formal definition of symmetry.

Definition 7.3 (automorphism) An automorphism on a transition system $M = (S, R, s_0)$ is a bijection $\sigma : S \rightarrow S$ such that $\forall s_1, s_2 \in S : (s_1, s_2) \in R \iff (\sigma(s_1), \sigma(s_2)) \in R$.

Let M be a transition system of a program. Let $\mathbf{id}$ be the identity automorphism on M. For any set of automorphism $\mathcal{A}$, the closure $G_\mathcal{A}$ of $\mathcal{A} \cup \{\mathbf{id}\}$ under inverse and composition is a group. We call such a group *a symmetry group* of M.

The symmetry group $G_\mathcal{A}$ induces an equivalence relation $\equiv_\mathcal{A}$ on S such that $s_1 \equiv_\mathcal{A} s_2$ if $\exists \sigma \in G_\mathcal{A} : s_2 = \sigma(s_1)$. The equivalent class $[s] = \{\sigma(s) \mid \sigma \in G_\mathcal{A}\}$ of s under $\equiv_\mathcal{A}$ is called the *orbit* of s under $G_\mathcal{A}$.

Definition 7.4 (quotient transition system) Given a transition system $M = (S, R, s_0)$ and a symmetry group $G_\mathcal{A}$ of M, a *quotient transition system* for M modulo $G_\mathcal{A}$ is a transition system $M_{[\mathcal{A}]} = (S', R', s'_0)$ in which $S' = \{[s] \mid s \in S\}$, $R' = \{([s], [s']) \mid (s, s') \in R\}$, and $s'_0 = [s_0]$.

When we say that there is a symmetry in a program, we mean that there is an automorphism other then the identity automorphism $\mathbf{id}$ on the transition system of the program.

Theorem 7.1 (reachability) Given a transition system $M = (S, R, s_0)$ with a set of automorphism $\mathcal{A}$ on M, s is reachable from s_0 in M if and only if $[s]$ is reachable from $[s_0]$ in $M_{[\mathcal{A}]}$.

In practice, the quotient transition system of a system is usually generated on-the-fly using a canonicalization function ζ [51]. Let s be a state. This function maps s to a unique representative $\zeta(s)$ of the equivalence class $[s]$. Whenever a state s is visited, $\zeta[s]$ is stored in memory, e.g., in a hash table.

In dynamic model checking, computing the runtime states of the system precisely is often difficult. In this context, a state is identified by the sequence of transitions that were executed from the initial state s_0 to reach the state. Based on this observation, one can explore symmetry on transitions (e.g., as in [26]) instead of on states.

Definition 7.5 Let $t = (s, s')$ be a transition of a transition system M. Let σ denote an automorphism in a symmetry group $G_{\mathcal{A}}$ of M. We use $\sigma(t)$ to denote the transition $(\sigma(s), \sigma(s'))$. The relation $\equiv_{\mathcal{A}}$ on transitions is defined as: $t \equiv_{\mathcal{A}} t'$ if $\exists \sigma \in G_{\mathcal{A}} : t' = \sigma(t)$.

It is not difficult to prove that the relation $\equiv_{\mathcal{A}}$ on transitions is an equivalence relation. One can extend $\equiv_{\mathcal{A}}$ on transitions to sequences of transitions.

Definition 7.6 Let $w = t_1 t_2 \dots t_n$ be a nonempty sequence of transitions of a transition system M. Let $\mathcal{A}$ be a set of automorphism on M, and $G_{\mathcal{A}}$ be the symmetry group of $\mathcal{A}$. Let $\sigma \in G_{\mathcal{A}}$. We write $\sigma(w)$ to denote the transition sequence $\sigma(t_1)\sigma(t_2)\dots\sigma(t_n)$. The relation $\equiv_{\mathcal{A}}$ on nonempty sequences of transitions is defined as: $w \equiv_{\mathcal{A}} w'$ if $\exists \sigma \in G_{\mathcal{A}} : w' = \sigma(w)$.

Here $\equiv_{\mathcal{A}}$ is also an equivalence relation. Based on the above definition, we use $[t]$ to denote the equivalence class $[t] = \{\sigma(t) \mid \sigma \in G_{\mathcal{A}}\}$ of t under $\equiv_{\mathcal{A}}$. Similarly, we use $[w] = \{\sigma(w) \mid \sigma \in G_{\mathcal{A}}\}$ to denote the equivalence class of w under $\equiv_{\mathcal{A}}$.

Definition 7.7 (quotient transition system under $\equiv_{\mathcal{A}}$) Let $M = (S, R, s_0)$ be a transition system. Let $G_{\mathcal{A}}$ be a symmetry group of M. A quotient transition system

for M modulo $G_{\mathcal{A}}$ defined with an equivalence relation $\equiv_{\mathcal{A}}$ on sequences of transitions is a transition system $M_{[\mathcal{A}]} = (S', R', s'_0)$ where $S' = \{[w] \mid s_0 \stackrel{w}{\Rightarrow} s \text{ in } M\}$, $R' = \{([w], [wt]) | s_0 \stackrel{w}{\Rightarrow} s \text{ and } \exists s' \in R : s \stackrel{t}{\rightarrow} s'\}$, and $s'_0 = [\epsilon]$ (the empty word).

Theorem 7.2 Let $M = (S, R, s_0)$ be a transition system and $M_{[\mathcal{A}]} = (S', R', s'_0)$ be a quotient transition system for M module a symmetry group $G_{\mathcal{A}}$ of M. Let s be a state in S. s is reachable from s_0 in M via w if and only if $[w]$ is reachable from s'_0 in M'.

7.2 Discovering Transition Symmetry

Following the definitions in the previous section, to reveal symmetry in a transition system, we need to find conditions that imply the existence of an automorphism of the transition system. As it is difficult to capture the states of multithreaded programs at runtime, the method of using canonicalization functions to reveal symmetry does not work well here. Our main idea is dynamically analyzing the residual code and the local states of threads to discover symmetry.

Let s be a state in the transition systems, and τ be a thread that is enabled in a state s. In a concrete execution, the transition that τ can execute from s is determined by the residual code $\mathcal{C}(s, \tau)$, the global state $g(s)$, and the local state of τ in s, i.e., $l_\tau(s)$. In this section, we present a simple algorithm based on this idea.

Let t_a and t_b be two transitions that are enabled at s by thread a and b, and let $c_a = \mathcal{C}(s, a)$ and $c_b = \mathcal{C}(s, b)$ be the residual code of thread a and b at s, respectively. In our algorithm, first we try to construct a bijection $\theta : VL(c_a) \rightarrow VL(c_b)$ such that c_a and c_b are syntactically equivalent under θ, that is, $c_a \stackrel{\theta}{\sim} c_b$. If such a θ can be constructed, we check whether the local states of threads are equivalent under θ. We can prove that transitions t_a and t_b are symmetric if the c_1 and c_2 are syntactically equivalent, and the local state of thread are equivalent under θ (detailed proof is in Section 7.2.3).

7.2.1 Inferring Syntactic Equivalence Among Residual Code

Let c_a and c_b be residual code of threads a and b that follow the syntax in Figure 7.2. Figure 7.4 shows the rules we use to construct a bijection θ from $VL(c_a)$ to $VL(c_b)$. We only conclude that c_a and c_b are syntactically equivalent if such a bijection can be

$$\frac{(m_1^a,\ \ m_1^b) \vdash \theta_1;\quad ([m_2^a, \ldots, m_n^a],\ \ [m_2^b, \ldots, m_n^b]) \vdash \theta_2}{([m_1^a, m_2^a, \ldots, m_n^a],\ \ [m_1^b, m_2^b, \ldots, m_n^b]) \vdash \theta_1 \cup \theta_2}\ \text{R0}$$

$$\frac{(m^a,\ \ m^b) \vdash \theta;\quad (l^a,\ \ l^b) \vdash [l^a \mapsto l^b]}{(l^a : m^a,\ \ l^b : m^b) \vdash \theta \cup [l^a \mapsto l^b]}\ \text{R1}$$

$$\frac{(e^a,\ \ e^b) \vdash \theta;\quad (l^a,\ \ l^b) \vdash [l^a \mapsto l^b]}{(\texttt{if}\ e^a\ \texttt{then goto}\ l^a,\quad \texttt{if}\ e^b\ \texttt{then goto}\ l^b) \vdash \theta \cup [l^a \mapsto l^b]}\ \text{R2}$$

$$\frac{(h^a,\ \ h^b) \vdash \theta_1;\quad (e^a,\ \ e^b) \vdash \theta_2}{(h^a = e^a,\ \ h^b = e^b) \vdash \theta_1 \cup \theta_2}\ \text{R3}$$

$$\frac{(f(p_1, \ldots, p_n),\ \ f(q_1, \ldots, q_n)) \vdash \theta_1;\quad (h^a,\ \ h^b) \vdash \theta_2}{(h^a = f(p_1, \ldots, p_n),\ \ h^b = f(q_1, \ldots, q_n)) \vdash \theta_1 \cup \theta_2}\ \text{R4}$$

$$\frac{(h_1^a,\ \ h_1^b) \vdash \theta_1 \quad (h_2^a,\ \ h_2^b) \vdash \theta_2}{(h_1^a \diamond h_2^a,\ \ h_1^b \diamond h_2^b) \vdash \theta_1 \cup \theta_2}\ \text{R5} \qquad \frac{([p_1, \ldots, p_n],\ \ [q_1, \ldots, q_n]) \vdash \theta}{(f(p_1, \ldots, p_n),\ \ f(q_1, \ldots, q_n)) \vdash \theta}\ \text{R6}$$

$$\frac{}{(v_g,\ \ v_g) \vdash [v_g \mapsto v_g]}\ \text{R7} \qquad \frac{}{(\&v_g,\ \ \&v_g) \vdash [v_g \mapsto v_g]}\ \text{R8}$$

$$\frac{}{(*v_g,\ \ *v_g) \vdash [v_g \mapsto v_g]}\ \text{R9} \qquad \frac{}{(\&v^a,\ \ \&v^b) \vdash [v^a \mapsto v^b]}\ \text{R10}$$

$$\frac{}{(*v^a,\ \ *v^b) \vdash [v^a \mapsto v^b]}\ \text{R11} \qquad \frac{}{(v^a,\ \ v^b) \vdash [v^a \mapsto v^b]}\ \text{R12}$$

$$\frac{}{(c,\ \ c) \vdash [\,]}\ \text{R13}$$

Figure 7.4. Rules for inferring syntactic equivalence among residual code of threads

constructed by successfully applying these rules. Otherwise, we treat c_a and c_b as non-syntactic-equivalent.

In these rules, we use m^τ to denote a statement in the residual code of thread τ, l^τ to denote a label in the residual code of thread τ. We use v_g to denote a global variable, v^τ to denote a local variable of thread τ, and c to denote a constant.

To simplify the presentation, we assume that the residual code c_a and c_b are two lists of statements. In practice, the residual code can be presented as a control flow graph or an abstract syntax tree. We can easily extend our rules to handle these structures. In our implementation, we represent the residual code of a thread as a control flow graph.

We start by applying rule R0 to c_a and c_b. R0 first checks whether m_1^a, which is the first statement in c_a, is syntactically equivalent to m_1^b, which is the first statement of c_b, by recursively applying other rules. Next, R0 recursively checks whether the rest of c_a are syntactically equivalent to the rest of c_b. If m_1^a and m_1^b are syntactically equivalent under θ_1, and the rest of residual code are syntactically equivalent under θ_2, we conclude that the two statement lists are syntactically equivalent under $\theta_1 \cup \theta_2$.

Rules R1 - R4 handles different kinds of statements, and rules R5 - R13 handles different forms of expressions. Specifically, rules R7 - R9 guarantees that the bijection we construct only maps a global variable to itself. We can also extend the rules to be more semantic-aware, for instance, by taking commutativity of binary operators into consideration (e.g., $x + y$ versus $y + x$).

Theorem 7.3 Let $c_1 = \mathcal{C}(s_1, a)$ and $c_2 = \mathcal{C}(s_2, b)$ be the residual code of thread a at s_1 and b at s_b, respectively. If a bijection $\theta : VL(c_1) \rightarrow VL(c_2)$ between c_1 and c_2 can be constructed following the rules in Figure 7.4, c_1 and c_2 are syntactically equivalent.

Theorem 7.3 can be easily proven by induction on the length of the statements.

7.2.2 Discovering Symmetric Transitions

Syntactic equivalence between the residual code of threads alone does not imply transition symmetry, as different threads may take different paths depending on the local states of threads. To soundly infer transition symmetry, we also need to examine the local states of threads. The procedure SYMMETRIC in Figure 7.5 shows our algorithm for detecting transition symmetry.

SYMMETRIC accepts three parameters as inputs: a state s and a pair of threads that are enabled in s. It returns TRUE or FALSE. Let a and b be the two threads. To test whether the transitions that enabled by a and b at s are symmetric transitions, we first compute the residual code of a and b (line 2 or Figure 7.5), which are c_1 and c_2, respectively. Then we check whether a bijection between $VL(c_1)$ and $VL(c_2)$ can be constructed following the rules in Figure 7.4. If such a bijection cannot be constructed, SYMMETRIC returns FALSE. Otherwise, let θ be the constructed bijection, we check whether the local states

```
1: SYMMETRIC(s, a, b) {
2:    let c_1 = C(s, a) and c_2 = C(s, b) ;
3:    if ( following the rules in Figure 7.4, there is a bijection θ such that C(s, a) ~θ
        C(s, b) )
4:      return l_a(s)[θ] = l_b(s);
5:    else
6:      return FALSE;
7: }
```

Figure 7.5. Checking whether two transitions enabled at s by a and b are symmetric

of a and b at s are equivalent under θ (line 4 of Figure 7.5). We only conclude transition symmetry if the local states of threads $l_a(s)$ and $l_b(s)$ are equivalent under this bijection.

7.2.3 Soundness

Lemma 7.1 Let $c_a = \mathcal{C}(s_a, a)$ and $c_b = \mathcal{C}(s_b, b)$ be the residual code of thread a at s_a and thread b at s_b, respectively. Let t_a be a transition that is enabled at s_a by thread a such that $s_a \xrightarrow{t_a} s'_a$. Suppose we can construct $\theta : VL(c_a) \rightarrow VL(c_b)$ following the rules in Figure 7.4 such that $c_a \overset{\theta}{\sim} c_b$. Now, if we have $g(s_a) = g(s_b)$, $l_a(s_a)[\theta] = l_b(s_b)$, there must exist a transition t_b that is enabled by thread b in s_b such that $s_b \xrightarrow{t_b} s'_b$, $g(s'_b) = g(s'_a)$, and $l_a(s'_a)[\theta] = l_b(s'_b)$.

Proof. A transition t of thread τ that is enabled at a state s is determined by the global state $g(s)$, the local state $l_\tau(s)$, and the residual code of τ. Let $\eta(s_a, t_a)$, which is the list of statements to be exercised while executing t_a from s_a, be $[m_1, \ldots, m_k]$. As $g(s_a) = g(s_b)$, $l_a(s_a)[\theta] = l_b(s_b)$, and we can construct a bijection θ following Figure 7.4 such that $c_a \overset{\theta}{\sim} c_b$, we have $\eta(s_b, t_b) = \eta(s_a, t_a)[\theta]$ (This can be proven by induction on k. We omit the details here.). Obviously there is s'_b such that $s_b \xrightarrow{t_b} s'_b$.

As $t_a \overset{\theta}{\sim} t_b$ and θ is constructed following the rules in Figure 7.4, $\eta(s_a, t_a)$ and $\eta(s_b, t_b)$ must have the same visible operation. As only the visible operations may update global objects, we have $g(s'_a) = g(s'_b)$.

$l_a(s'_a)$ is determined by $g(s'_a)$, $l_a(s_a)$, and the invisible operations in t_a, and $l_b(s'_b)$ is determined by $g(s'_b)$, $l_b(s_b)$, and the invisible operations in t_b. As $t_a \overset{\theta}{\sim} t_b$, we can use induction to prove that if $l_a(s_a)[\theta] = l_b(s_b)$, $l_a(s'_a)[\theta] = l_b(s'_b)$. □

Lemma 7.2 Let t_a and t_b be two transitions that are enabled at a state s by thread a and b, and let s_a and s_b be two states in M such that $s \xrightarrow{t_a} s_a$ and $s \xrightarrow{t_b} s_b$. If SYMMETRIC(s, a, b) returns TRUE, let $\theta : VL(\mathcal{C}(s,a)) \rightarrow VL(\mathcal{C}(s,b))$ be the bijection that we construct following the rules in Figure 7.4, we have $s_b = s_a[a := l_b(s_a)[\theta^{-1}]; b := l_a(s_a)[\theta]]$.

Proof. As a transition t that is enabled at a state s can only changes the global states of s and the local state of $tid(t)$ in s, t_a does not change the local state of b in s. Hence, $l_b(s_a) = l_b(s)$. Similarly, we have $l_a(s_b) = l_a(s)$. According to Lemma 7.1, we have $g(s_b) = g(s_a)$, and $l_b(s_b) = l_a(s_a)[\theta]$. Based on this, we have

$$\begin{aligned} s_b &= s_a[a := l_a(s); b := l_b(s_b)] \\ &= s_a[a := l_a(s)[\theta][\theta^{-1}]; b := l_b(s_b)] \\ &= s_a[a := l_b(s)[\theta^{-1}]; b := l_b(s_b)] \\ &= s_a[a := l_b(s_a)[\theta^{-1}]; b := l_a(s_a)[\theta]] \end{aligned}$$

□

Lemma 7.3 Let t_a and t_b be two transitions that are enabled at a state s by thread a and b. If SYMMETRIC(s, a, b) returns TRUE, let θ be the bijection from $VL(\mathcal{C}(s,a))$ to $VL(\mathcal{C}(s,b))$ that we construct following the rules in Figure 7.4. Then for any transition sequence $t_1, \ldots, t_k$ in M such that $s_c \xrightarrow{t_a} s_a \xrightarrow{t_1} s_1 \ldots \xrightarrow{t_k} s_k$, there must exist another transition sequence $t'_1, \ldots, t'_k$ in M such that $s_c \xrightarrow{t_b} s_b \xrightarrow{t'_1} s'_1 \ldots \xrightarrow{t'_k} s'_k$ and for all i, $1 \leq i \leq k$, $s'_i = s_i[a := l_b(s_i)[\theta^{-1}]; b := l_a(s_i)[\theta]]$.

Here we provide only a sketch of the proof here. We can prove this lemma by induction on the length of the transition sequence. In the base case, $k = 1$. Let t_1 be a transition such that $s \xrightarrow{t_a} s_a \xrightarrow{t_1} s_1$. Following Lemma 7.2, we have $g(s_a) = g(s_b)$, and $l_b(s_b) = l_a(s_a)[\theta]$. Based on this and Lemma 7.1, we can prove the base case by studying three cases: (i) $tid(t_1) \neq a$ and $tid(t_1) \neq b$; (ii) $tid(t_1) = a$ and (iii) $tid(t_1) = b$. The induction step can be proven similarly.

Theorem 7.4 Let s_m be a state of a transition system $M = (S, R, s_0)$. Let t_a and t_b be two transitions that are enabled at s_m by thread a and b. If SYMMETRIC(s_m, a, b) returns TRUE, t_a and t_b are symmetric transitions.

Proof. Let $c_1 = \mathcal{C}(s_m, a)$ and $c_2 = \mathcal{C}(s_m, b)$ be the residual code of a and b at the state s. Let t_a and t_b be the transitions of a and b at s such that $s_m \xrightarrow{t_a} s_a$ and $s_m \xrightarrow{t_b} s_b$. If

SYMMETRIC(s_m, a, b) returns TRUE, let $\theta : VL(c_1) \rightarrow VL(c_2)$ be the bijection that we construct following the rules in Figure 7.4. We can construct an automorphism σ on M as follows:

$$\sigma(s) = \begin{cases} s & \text{if } s \text{ is not reachable from } s_a \text{ or } s_b \\ s[a := l_b(s)[\theta^{-1}]; b := l_a(s)[\theta]] & \text{otherwise} \end{cases}$$

Obviously, for any state s that is not reachable from s_a or s_b, $(s, s') \in R \iff (\sigma(s), \sigma(s')) \in R$ holds.

For any state s_k that is reachable from s_a or s_b. Let $s_a \xrightarrow{t_1} s_1 \ldots \xrightarrow{t_k} s_k$ be the transition sequence from s_a to reach s_k. Following Lemma 7.3, if $(s_k, s_{k+1}) \in R$, $(\sigma(s_k), \sigma(s_{k+1})) \in R$. If $(\sigma(s_k), \sigma(s_{k+1})) \in R$, with $\theta' = \theta^{-1}$, again according to Lemma 7.3, we have $(s_k, s_{k+1}) \in R$. Hence σ is an automorphism of M. □

7.3 Dynamic Partial Order Reduction with Symmetry Discovery

Dynamic partial order reduction (DPOR) [21] is an effective algorithm for reducing redundant interleavings in dynamic model checking. In DPOR, given a state s, the persistent set of s [24] is not computed immediately after reaching s. Instead, DPOR explores the states that can be reached from s using depth-first search, and dynamically adds backtrack information into the backtrack set of s while exploring the subspace that is reachable from s.

In more detail, let t_i be a transition that is enabled at state s. Suppose the model checker first selects t to execute at s. Let t_j be a transition that can be enabled with a depth-first search (in one or more steps) from s by executing t_i. Then, before t_j is executed, DPOR will check whether t_j and t_i are dependent and can be enabled concurrently. If so, $tid(t_j)$ or the identity of the thread on which t_j is dependent will be added to the backtrack set of s if a transition of $tid(t_j)$ is enabled at s. Later, in the process of backtracking, if the state s is found with nonempty backtrack set, DPOR will select one transition t that is enabled at s and $tid(t)$ is in the backtrack set of s, and explore a new branch of the state space by executing t from s; at the same time, $tid(t)$ will be removed from the backtrack set of s.

As shown in Figure 7.6, combining our transition symmetry discovery method with dynamic partial order reduction is straightforward. In this algorithm, we use $s.enabled$ to denote the set of enabled transitions in a state s, $s.backtrack$ to denote the set of enabled threads that need to be explored at a state s, and $s.done$ to denote the set of enabled threads the transitions of which have been executed at s. Comparing with the original DPOR algorithm, the only place we need to change w.r.t. the original DPOR algorithm is in line 11 of Figure 7.6, where we check whether a symmetric transition has been explored before exploring a transition.

Theorem 7.5 Consider a terminating multithreaded program M. If there exist deadlocks in the state space of M, SYMPOR will visit at least one of them. If there exist data races in the state space of M, SYMPOR will visit at least one of them. If we encode local assertions as part of the residual code, and there exist local assertion violations in the state space of M, SYMPOR will visit at least one of them.

Proof. Following the definition of deadlock, data races, or local assertion violations, and Theorem 7.4, if a state s has a deadlock, each state s' in $[s]$ has a deadlock. Thus, exploring one state per equivalent class $[s]$ shall be sufficient to detect deadlocks in multithreaded programs. Similarly, the theorem holds for data races. As for local assertions, if they are encoded as part of the residual code and there exist local assertion violations in the program, all states that belong to the same equivalent class must violate the assertion. Hence, the theorem holds. □

7.4 Implementation and Evaluation

We have implemented the transition symmetry discovery algorithm on top of our dynamic model checker `Inspect` [97, 47]. With source-level instrumentation, `Inspect` uses an external scheduler to intercept visible operations of multithreaded C programs and systematically explore interleavings of multithreaded C programs under specific inputs. To implement the symmetry discovery algorithm, we added a dynamic analyzer to compute the residual code of threads, and added a prober thread to check the local states of threads.

```
1:  Initially: S is empty; SYMPOR(S, s_0)

2:  SYMPOR(S, s) {
3:      S.push(s);
4:      for each t ∈ s.enabled, UPDATEBACKTRACKSETS(S, t);
5:      let τ ∈ Tid such that ∃t ∈ s.enabled : tid(t) = τ;
6:      s.backtrack ← {τ};
7:      s.done ← ∅;
8:      while (∃t ∈ s.backtrack \ s.done) {
9:        s.done ← s.done ∪ {t};
10:       s.backtrack ← s.backtrack \ {t};
11:       if (∀t' ∈ s.done : ¬SYMMETRIC(s, tid(t), tid(t')))){
12:         let s' ∈ S such that s -t-> s';
13:         SYMPOR(S, s');
14:         S.pop(s);
15: } } }

16: UPDATEBACKTRACKSETS(S, t) {
17:   let T be the sequence of transitions associated with S;
18:   let t_d be the latest transition in T that is dependent and may be co-enabled with t;
19:   if (t_d ≠ null){
20:     let s_d be the state in S from which t_d is executed;
21:     let E be {q ∈ s_d.enabled | tid(q) = tid(t), or q in T, q was after t_d in T,
            and there is a happens-before relation for (q, t)}
22:     if (E ≠ ∅)
23:       choose any q in E, add tid(q) to s_d.backtrack;
24:     else
25:       s_d.backtrack ← s_d.backtrack ∪ {tid(q) | q ∈ s_d.enabled};
26:     }
27: }
```

Figure 7.6. Dynamic partial order reduction with symmetry discovery

To compute the residual code of threads, the scheduler needs to know the execution contexts of threads and shall be able to perform program analysis under the dynamic context. We instrument some code into the program with the help of which the scheduler can learn the execution context of threads. The instrumentation we do for a multithreaded program are (i) inserting code right before function calls to number the function calls; (ii) adding code right before the first statement of each function to capture the new frame

that is pushed into the call stack; (iii) adding a probing thread that can help the scheduler to get the concrete values of local variables of threads in an execution of the program, the pseudocode of the probing thread is shown in Figure 7.7; and (iv) adding struct parsing functions for user-defined structs such that we can probe fields of the structs.

We use CIL [74] as the front end for parsing and instrumenting the multithreaded C programs. We have CIL to output the parsed ASTs to an intermediate file, and have the scheduler parse the intermediate file to reconstruct the ASTs. By learning the execution context under the helper of the prober thread, we can easily compute the residual code of threads at a state. The equivalence of local states under a bijection between local variables of threads (line 6 of Figure 7.5) is checked with the help of the prober thread.

7.5 Experiment Results

In realistic multithread applications, it is quite common that multiple threads are spawned from the same thread routine, and are assigned tasks of the same category to speed up processing [66]. For instance, the master/worker pattern, which is widely used in multithreaded programs, uses multiple workers to speed up processing. The shared queue pattern, which is also frequently used in mulithreaded application, often employs multiple producers for enqueing and consumers for dequeing. Our experiments show that our algorithm can successfully reveal symmetries in these applications and significantly speed up the dynamic mode checking.

We conducted experiments on two realistic multithreaded benchmarks, `aget` and `pfscan`. `aget` [43] is an ftp client that uses multiple worker threads to download different segments of a large file concurrently. `pfscan` [44] is a multithreaded file scanner that uses multiple threads to search in parallel through directories. `aget-buggy` and

```
1: while (TRUE) {
2:     wait query request from the scheduler;
3:     probing value or address according to the request;
4:     send result to the scheduler;
5: }
```

Figure 7.7. The procedure of the probing thread

`pfscan-buggy` are buggy versions of `aget` and `pfscan` with inserted data race and deadlock bugs. All benchmarks are accompanied by test cases to facilitate the concrete execution. Our experiments were conducted on a workstation with 2.8 GHz Pentium D processor and 2GB memory running Fedora 5.

Table 7.1 shows the experimental results. The first three columns show the statistics of the test cases, including the name, the line of code, and the number of threads. Columns 4-5 compare the two methods in terms of their total runtime in seconds. Columns 6-7 compare the number of explored executions before they produce verification results. Columns 8-9 compare the number of explored transitions.

As shown in Table 7.1, our symmetry discovery scheme can help to significantly reduce the checking time, with only modest overheads. For instance, the symmetry checking adds 15%-40% overhead on the time per execution of `pfscan`. However, this overhead is well compensated by the checking time we save with symmetry reduction. Furthermore, the symmetry discovery step can be made more efficient by precomputing the bijections between residual code of threads and storing them in a hash table along with the bijections.

Table 7.1. Experimental results on symmetry

Test programs		Runtime		Executions		Transitions	
Benchmark	thrds	DPOR	Sympor	DPOR	Sympor	DPOR	Sympor
aget-bug	4	8h5m	16s	1009k	420	30233k	19k
aget-bug	5	>24h	37s	-	926	-	32k
aget	4	> 24h	18s	-	462	-	16k
aget	5	> 24h	4m18s	-	6k	-	211k
aget	6	> 24h	43m	-	88k	-	3117k
pfscan-bug	3	2s	1s	120	71	2k	1.2k
pfscan-bug	4	6m29s	50s	28079	3148	428k	50k
pfscan	3	21s	3s	1096	136	18006	2334
pfscan	4	17h48m	2m31s	4185k	7k	64465k	85k
pfscan	5	>24h	51m47s	-	323k	-	5374k

7.6 Related work

There has been a lot of research on automatic symmetry discovery. In solving boolean satisfiability, a typical approach is to convert the problem into a graph and employ graph symmetry discovery tools to uncover symmetry [2]. Another approach for discovering symmetry is boolean matching [7], which converts the boolean constraints into a canonization form to reveal symmetries. In domains such as microprocessor verification, the graph often has a large number of vertices; however, the average number of neighbors of a vertex is usually small. Several algorithms based on exploiting this fact [14, 53] are proposed to efficiently handle these graphs. More recent effort on discovery symmetry using sparsity [15] significantly reduced the discovery time by exploiting the sparsity in both the input and the output of the system.

In explicit state software model checking, state canonicalization has been the primary method to reveal symmetry. Efficient canonization functions [61, 50] have been proposed to handle heap symmetry in Java programs that create objects in dynamic area.

In dynamic model checking of concurrent programs, transition symmetry [26] has been the main method for exploiting symmetry at the whole process level. However, in [26], the user is required to come up with a permutation function, which is then used by the algorithm to check whether two transitions are symmetric. In practice, it is often difficult to manually specify such a permutation function. By employing dynamic analysis, our approach automates symmetry discovery. To the best of our knowledge, our algorithm is the first effort in automating symmetry discovery for dynamic model checking.

7.7 Summary

We propose a new algorithm that uses dynamic program analysis to discover symmetry in mulithreaded programs. The new algorithm can be easily combined with partial order reduction algorithms and significantly reduce the runtime of dynamic model checking. In future work, we would like to further improve the symmetry discovery algorithm with a more semantic-aware dynamic analysis. Since dynamic analysis can

be a helpful technique for testing and verification in many contexts, we are investigating several possibilities in this direction.

CHAPTER 8

CONCLUSION AND FUTURE WORK

In this dissertation, we focused on combining program analysis and model checking for efficient dynamic verification of concurrent programs. We developed several algorithms that use program analysis to help prune the search space for efficient systematic testing of multithreaded programs. We designed `Inspect`, a framework for systematic testing of multithreaded C programs, and implemented those algorithms in `Inspect`.

`Inspect` is a framework for systematic testing of multithreaded C programs under specific test harnesses. It is aimed to reveal concurrency related errors (data races, deadlocks, local assertion violation, etc.). It is primarily composed of three parts: (i) a program analyzer that analyzes the program; (ii) a program instrumentor that does source-level instrumentation; and (iii) an external scheduler that monitors the concrete execution of the instrumented multithreaded program. `Inspect` repeatedly executes the program under given test harnesses and uses the external scheduler to guide the program to systematically explore different interleavings.

To deal with the interleaving-explosion problem that we face when the size of the multithreaded program increases, we proposed four algorithms that leverage program analysis to prune the search spaces. Our experiments confirm the effectiveness of our approach on several multithreaded benchmarks in C, including some practical programs.

8.1 Future Research Directions

I believe that program analysis will be essential in the next generation of development environments. Tools similar to `Inspect`, which combine program analysis and formal methods for efficient testing of programs, will be an important part of the future development environment. `Inspect` is an initial attempt in exploring this direction. In the near future, I would like to explore the following lines:

8.1.1 Using Dynamic Program Analysis to Improve Test Case Generation

We can use dynamic program analysis for more efficient test case generation of sequential programs. One immediate integration is to combine this technique with concolic testing tools for property-specific concolic testing. I also plan to investigate other scenarios in which dynamic program analysis can help to improve testing efficiency.

8.1.2 Program Analysis at the Binary Level

Currently `Inspect` does static analysis at the source level. It may require significant engineering effort to deal with front ends and language-specific features while porting the algorithms that are implemented in `Inspect` to other programming languages. One possible approach to avoid the engineering effort is to do static analysis and instrumentation at the binary level. Doing analysis at the binary level, we can also avoid the potential inconsistency problem that is introduced by compiler optimization.

8.1.3 Testing Concurrent Programs under Open Environments

Systematic testing tools such as `Inspect` require a closed environment to replay the program for backtracking. However, a large number of concurrent programs deal with open environments. I plan to develop algorithms and tools that can test the concurrent programs with a better coverage under the open environments.

REFERENCES

[1] http://manju.cs.berkeley.edu/cil/.

[2] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult sat instances in the presence of symmetry. In *DAC '02: Proceedings of the 39th Conference on Design Automation, New Orleans, Louisiana, USA, June 10-14, 2002*, pages 731–736. ACM, 2002.

[3] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 484–487. Springer, 2004.

[4] H.-J. Boehm. Threads cannot be implemented as a library. In V. Sarkar and M. W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 261–268. ACM, 2005.

[5] L. Brim, I. Cerná, P. M. 0002, and J. Simsa. Distributed partial order reduction of state spaces. *Electr. Notes Theor. Comput. Sci.*, 128(3):63–74, 2005.

[6] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science, 4-7 June 1990, Philadelphia, Pennsylvania, USA*, pages 428–439. IEEE Computer Society, 1990.

[7] D. Chai and A. Kuehlmann. Building a better boolean matcher and symmetry detector. In *DATE '06: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1079–1084, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[8] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. In *ICSE*, pages 385–395. IEEE Computer Society, 2003.

[9] F. Chen and G. Rosu. Parametric and sliced causality. In *Computer Aided Verification*, pages 240–253. Springer, 2007. LNCS 4590.

[10] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Form. Methods Syst. Des.*, 9(1-2):77–104, 1996.

[11] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[12] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[13] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Satabs: Sat-based predicate abstraction for ansi-c. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer, 2005.

[14] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for cnf. In *DAC '04: Proceedings of the 41st Annual Conference on Design Automation*, pages 530–534, New York, NY, USA, 2004. ACM.

[15] P. T. Darga, K. A. Sakallah, and I. L. Markov. Faster symmetry discovery using sparsity of symmetries. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 149–154, New York, NY, USA, 2008. ACM.

[16] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[17] B. Dutertre and L. M. de Moura. A fast linear-arithmetic solver for dpll(t). In T. Ball and R. B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2006.

[18] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[19] E. A. Emerson and A. P. Sistla. Symmetry and model checking. *Form. Methods Syst. Des.*, 9(1-2):105–131, 1996.

[20] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, New York, NY, USA, 2003. ACM Press.

[21] C. Flanagan and P. Godefroid. Dynamic Partial-order Reduction for Model Checking Software. In J. Palsberg and M. Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 110–121. ACM, 2005.

[22] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, New York, NY, USA, 2002. ACM Press.

[23] M. Ganai, S. Kundu, and R. Gupta. Partial order reduction for scalable testing of SystemC TLM designs. In *Design Automation Conference*, 2008.

[24] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Springer, 1996. LNCS 1032.

[25] P. Godefroid. Model Checking for Programming Languages using Verisoft. In *POPL*, pages 174–186, 1997.

[26] P. Godefroid. Exploiting symmetry when model-checking software. In J. Wu, S. T. Chanson, and Q. Gao, editors, *FORTE*, volume 156 of *IFIP Conference Proceedings*, pages 257–275. Kluwer, 1999.

[27] P. Godefroid. Software model checking: The VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, 2005.

[28] P. Godefroid and N. Nagappan. Concurrency at Microsoft - An Exploratory Survey. In R. Alur, G. Gopalakrishnan, V. Kahlon, and S. Siegel, editors, *International Workshop on Exploiting Concurrency Efficiently and Correctly, Princeton, NJ, USA, July 6-7, 2008, Proceedings*, pages 17–20.

[29] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). In C. Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 438–449. Springer, 1993.

[30] P. Godefroid and P. Wolper. A partial approach to model checking. *Inf. Comput.*, 110(2):305–326, 1994.

[31] G. Gopalakrishnan, Y. Yang, and H. Sivaraj. Qb or not qb: An efficient execution verification tool for memory orderings. In R. Alur and D. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 401–413. Springer, 2004.

[32] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In O. Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[33] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian partial-order reduction. In D. Bosnacki and S. Edelkamp, editors, *SPIN*, volume 4595 of *Lecture Notes in Computer Science*, pages 95–112. Springer, 2007.

[34] S. Hallem, B. Chelf, Y. Xie, and D. R. Engler. A system and language for building system-specific, static analyses. In *PLDI*, pages 69–82, 2002.

[35] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, 2000.

[36] C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy. Automatic generation of schedulings for improving the test coverage of systems-on-a-chip. In *Formal Methods in Computer-Aided Design, 6th International Conference, FMCAD 2006, San Jose, California, USA, November 12-16, 2006, Proceedings*, pages 171–178. IEEE Computer Society, 2006.

[37] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 1–13, New York, NY, USA, 2004. ACM Press.

[38] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. In W. Pugh and C. Chambers, editors, *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pages 1–13. ACM, 2004.

[39] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual* . Addison-Wesley, 2004.

[40] G. J. Holzmann and D. Bosnacki. The design of a multicore extension of the spin model checker. *IEEE Trans. Software Eng.*, 33(10):659–674, 2007.

[41] http://bzip2smp.sourceforge.net/.

[42] http://en.wikipedia.org/wiki/2003_North_America_blackout.

[43] http://freshmeat.net/projects/aget/.

[44] http://freshmeat.net/projects/pfscan.

[45] http://research.microsoft.com/en us/um/people/mbj/marspathfinder/.

[46] http://www.cert.org/advisories/.

[47] http://www.cs.utah.edu/∼yuyang/inspect.

[48] http://www.lam mpi.org/.

[49] http://www.mpi forum.org/docs/docs.html.

[50] R. Iosif. Symmetry reductions for model checking of concurrent dynamic software. *STTT*, 6(4):302–319, 2004.

[51] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

[52] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-soft: Software verification platform. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 301–306. Springer, 2005.

[53] T. Junttila and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *SIMA Workshop on Algorithm Engineering and Experiments*, 2007.

[54] V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In R. Gupta and S. P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 249–259. ACM, 2008.

[55] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In W. Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 226–239. Springer, 2007.

[56] S. Katz and D. Peled. Defining conditional independence using collapses. *Theor. Comput. Sci.*, 101(2):337–359, 1992.

[57] R. Kumar and E. G. Mercer. Load Balancing Parallel Explicit State Model Checking. *Electr. Notes Theor. Comput. Sci.*, 128(3):19–34, 2005.

[58] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[59] E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[60] Y. Lei and R. H. Carver. Reachability testing of concurrent programs. *IEEE Trans. Software Eng.*, 32(6):382–403, 2006.

[61] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In M. B. Dwyer, editor, *SPIN*, volume 2057 of *Lecture Notes in Computer Science*, pages 80–102. Springer, 2001.

[62] N. Leveson and C. Turner. Investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[63] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. Muvi: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In T. C. Bressoud and M. F. Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 103–116. ACM, 2007.

[64] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In J. P. Shen and M. Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 37–48. ACM, 2006.

[65] E. Marcus and H. Stern. *Blueprints for High Availability*. Wiley, 2000.

[66] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2004.

[67] A. W. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, pages 279–324. Springer, 1986. LNCS 255.

[68] K. L. McMillan. Interpolation and sat-based model checking. In W. A. H. Jr. and F. Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2003.

[69] I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. M. Kirby, and G. Gopalakrishnan. Parallel and Distributed Model Checking in Eddy. In *SPIN*, pages 108–125, 2006.

[70] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001.

[71] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. Cmc: A pragmatic approach to model checking real code. In *5th Symposium on Operating System Design and Implementation, Boston, MA, USA, December 9-11, 2002, Proceedings*, pages 75–88, New York, NY, USA, 2002. USENIX Association.

[72] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In J. Ferrante and K. S. McKinley, editors, *PLDI*, pages 446–455. ACM, 2007.

[73] National Institute of Standards and Technology (NIST). Software Errors Cost U.S. Economy $59.5 Billion Annually. http://www.nist.gov/public_affairs/releases/n02-10.htm. June, 2002.

[74] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[75] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[76] R. Palmer and G. Gopalakrishnan. A distributed partial order reduction algorithm. In *FORTE '02: Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*, page 370, London, UK, 2002. Springer-Verlag.

[77] R. Palmer and G. Gopalakrishnan. Partial Order Reduction Assisted Parallel Model Checking. *PDMC*, 2002.

[78] R. L. Palmer. *Formal Analysis for MPI-based High Performance Computing Software, Ph.D. Dissertation, University of Utah*. 2007.

[79] D. Peled. All from one, one for all: on model checking using representatives. In *CAV '93: Proceedings of the 5th International Conference on Computer Aided Verification*, pages 409–423, London, UK, 1993. Springer-Verlag.

[80] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 320–331, New York, NY, USA, 2006. ACM Press.

[81] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/SIGSOFT FSE*, pages 267–276, 2003.

[82] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Principles and Practices of Parallel Programming*, pages 12–23. ACM Press, 2001.

[83] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[84] D. Scott. Assessing the costs of application downtime. Technical report, Gartner Group, 1998.

[85] K. Sen, G. Rosu, and G. Agha. Detecting errors in multithreaded programs by generalized predictive analysis of executions. In *Formal Methods for Open Object-Based Distributed Systems*, pages 211–226. Springer, 2005. LNCS 3535.

[86] H. Sivaraj and G. Gopalakrishnan. Random Walk Based Heuristic Algorithms for Distributed Memory Model Checking. *Electr. Notes Theor. Comput. Sci.*, 89(1), 2003.

[87] M. Snir and S. Otto. *MPI-The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[88] U. Stern and D. L. Dill. Parallelizing the Mur*hi* Verifier. In O. Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–278. Springer, 1997.

[89] S. S. Vakkalanka, G. Gopalakrishnan, and R. M. Kirby. Dynamic verification of mpi programs with reductions in presence of split operations and relaxed orderings. In A. Gupta and S. Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 66–79. Springer, 2008.

[90] A. Valmari. Stubborn sets for reduced state generation. In *APN 90: Proceedings on Advances in Petri nets 1990*, pages 491–515, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[91] W. Visser, K. Havelund, G. P. Brat, and S. Park. Model Checking Programs. In *ASE*, pages 3–12, 2000.

[92] A. Vo, S. Vakkalanka, M. Delisi, G. Gopalakrishnan, R. M. Kirby, and R. Thakur. Formal verification of practical mpi programs. In *14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), February 14-18, 2009, Raleigh, North Carolina, Proceedings*. ACM, 2009.

[93] J. Voung, R. Jhala, and S. Lerner. RELAY: static race detection on millions of lines of code. In *Foundations of Software Engineering*, pages 205–214. ACM, 2007.

[94] C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. In *6th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, volume 5311 of *Lecture Notes in Computer Science*, pages 126–140. Springer, 2008.

[95] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In V. Sarkar and M. W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 1–14. ACM, 2005.

[96] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. Modist: Transparent model checking of unmodified distributed systems. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, USA, 2009.

[97] Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A Runtime Model Checker for Multithreaded C Programs. Technical Report UUCS-08-004, University of Utah, 2008.

[98] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In *14th International SPIN Workshop on Model Checking Software*, volume 4595 of *Lecture Notes in Computer Science*, pages 58–75. Springer, 2007.

[99] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient stateful dynamic partial order reduction. In *15th International SPIN Workshop on Model Checking*

Software, volume 5156 of *Lecture Notes in Computer Science*, pages 288–305, 2008.

[100] Y. Yang, X. Chen, G. Gopalakrishnan, and C. Wang. Automatic Discovery of Transition Symmetry Using Dynamic Analysis, in submission. 2009.

[101] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. Memory-model-sensitive data race analysis. In J. Davies, W. Schulte, and M. Barnett, editors, *Formal Methods and Software Engineering, 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004, Proceedings*, volume 3308 of *Lecture Notes in Computer Science*, pages 30–45. Springer, 2004.

[102] X. Yi, J. Wang, and X. Yang. Stateful Dynamic Partial-Order Reduction. In Z. Liu and J. He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 149–167. Springer, 2006.