The Dissertation Committee for Katherine Elizabeth Coons
certifies that this is the approved version of the following dissertation:

Fast Error Detection with Coverage Guarantees for Concurrent Software

Committee:

Kathryn S. McKinley, Supervisor

James C. Browne

William R. Cook

Calvin Lin

Madan Musuvathi

Fast Error Detection with Coverage Guarantees for Concurrent Software

by

Katherine Elizabeth Coons, B.S.;M.S.C.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2013

In loving memory of my grandmother, Jean Schlager.

Acknowledgments

I wish to thank my committee members: Kathryn McKinley, Madan Musuvathi, J.C. Browne, William Cook, and Calvin Lin for their valuable feedback on my research. Their insights improved this dissertation immensely.

I am particularly indebted to my advisor, Kathryn McKinley, for her guidance, support, and constant encouragement. She is an ideal role model and I am honored to have her as an advisor, mentor, and friend. I will look to her for inspiration as I move forward with my career.

I thank Madan Musuvathi for teaching me the value of combining theory and practice. When they come together I find it beautiful, and I thank Madan for this. I would like to additionally thank Doug Burger for motivating me to push my limits, and Jim Larus for taking interest in my career and opening doors for me. Doug and Jim have inspired me as researchers, and as people. I built on Patrice Godefroid's work extensively and I thank him for providing such a sound foundation on which to build, and for being exceedingly kind in answering my questions.

I would like to thank the friends I made in graduate school for their support and for the pleasure of their company. Graduate school would have been unbearable without you. In particular, I would like to thank Simha Sethumadhavan for always believing I was capable of more than I realized. Childhood friends Elizabeth Rothschild and Dawn Hamilton supported me through the best and the worst moments, and gave me confidence when I needed it most. I will be forever indebted to them.

I would like to thank my parents, Carol and T.A. Coons, for their unconditional love and support. They taught me to value education, gave me great freedom, and trusted me to use that freedom wisely. The values they instilled in me have made me a better researcher, and a better person.

Finally, I would like to thank my husband, Bert Maher, for his unwavering belief in me. Without his encouragement and his shoulder to cry on, I would never have made it through.

Katherine Elizabeth Coons

The University of Texas at Austin
May 2013

Fast Error Detection with Coverage Guarantees for Concurrent Software

Publication No. _____

Katherine Elizabeth Coons, Ph.D.
The University of Texas at Austin, 2013

Supervisor: Kathryn S. McKinley

Concurrency errors are notoriously difficult to debug because they may occur only under unexpected thread interleavings that are difficult to identify and reproduce. These errors are increasingly important as recent hardware trends compel developers to write more concurrent software and to provide more concurrent abstractions. This thesis presents algorithms that dynamically and systematically explore a program's thread interleavings to manifest concurrency bugs quickly and reproducibly, and to provide precise incremental coverage guarantees.

Dynamic concurrency testing tools should provide (1) *fast response* – bugs should manifest quickly if they exist, (2) *reproducibility* – bugs should be easy to reproduce and (3) *coverage* – precise correctness guarantees when no bugs manifest.

In practice, most tools provide either fast response or coverage, but not both. These goals conflict because a program's thread interleavings exhibit exponential *state-space explosion*, which inhibits fast response.

Two approaches from prior work alleviate state-space explosion. (1) *Partial-order reduction* provides *full coverage* by exploring only one interleaving of independent transitions. (2) *Bounded search* provides *bounded coverage* by enumerating only interleavings that do not exceed a bound. Bounded search can additionally provide guarantees for cyclic state spaces for which dynamic partial-order reduction provides no guarantees. Without partial-order reduction, however, bounded search wastes most of its time exploring executions that reorder only independent transitions. Fast response with coverage guarantees requires both approaches, but prior work failed to combine them soundly.

We combine bounded search with partial-order reduction and extensively analyze the space of dynamic, bounded partial-order reduction strategies. We first prioritize with best-first search and show that heuristics that combine these approaches find bugs quickly. We then restrict partial-order reduction to combine these approaches while maintaining bounded coverage. We specialize this approach for several bound functions, prove that these algorithms guarantee bounded coverage, and leverage dynamic information to further reduce the state space.

Finally, we bound the *partial order* on a program's transitions, rather than the total order on those transitions, to combine these approaches without sacrificing partial-order reduction. This algorithm provides fast response, incremental coverage guarantees, and reproducibility. We manifest bugs an order of magnitude more quickly than previous approaches and guarantee incremental coverage in minutes or hours rather than weeks, helping developers find and reproduce concurrency errors. This thesis makes bounded stateless model checking for concurrent programs substantially more efficient and practical.

Contents

Chapter 1

Introduction

Concurrent software is notoriously difficult to debug because errors may occur only with unexpected thread interleavings that are difficult to identify and reproduce. This problem is increasingly important because power limitations have compelled hardware developers to abandon single-thread performance in favor of parallel hardware. In response, developers must write more concurrent code and develop concurrent abstractions.

To debug and test concurrent software, dynamic tools execute the program with different thread interleavings, searching for one that manifests a bug. A bug manifests when the program crashes or a deadlock or user-specified assertion failure occurs. Experience with developers and testers suggests that dynamic concurrency testing tools should provide:

1. *Fast Response:* If a bug exists, it should manifest quickly.

2. *Reproducibility:* After a bug manifests, it should be easy to reproduce.

3. *Coverage:* The search should complete with a precise coverage guarantee.

Fast response, reproducibility, and coverage are all important to developers, and dynamic concurrency testing tools currently fall short. Three classes of tools meet

only one or two of these requirements:

1. **Stress testing** repeatedly runs the test under heavy load to increase the probability that a rare schedule will occur. Stress testing provides fast response, particularly during early stages of testing, but it does not guarantee reproducibility or coverage.

2. **Heuristic-based fuzzing** directs the thread scheduler towards interleavings likely to manifest a bug based on heuristics [Edelstein et al., 2003, Sen, 2008, Park et al., 2009, Joshi et al., 2009, Burckhardt et al., 2010, Nagarakatte et al., 2012]. Fuzzing provides fast response, but it does not guarantee coverage.

3. **Stateless model checking** systematically enumerates a program's thread interleavings [Godefroid, 1997, Musuvathi and Qadeer, 2007a]. Model checkers guarantee coverage and reproducibility, but they do not provide fast response.

None of these approaches provide fast response and guarantee coverage because exploring the state space suffers from exponential *state-space explosion* – the number of thread interleavings for a concurrent program is exponential in the length of the program and the number of threads. Two approaches from prior work alleviate this problem. (1) *Partial-order reduction* explores the entire state space but selects only one interleaving of independent transitions and thus provides *full coverage* [Godefroid, 1997, Godefroid, 1996, Flanagan and Godefroid, 2005]. (2) *Bounded search* enumerates all thread interleavings that do not exceed a bound, and thus provides *bounded coverage* [Musuvathi and Qadeer, 2007a, Emmi et al., 2011].

We consider *coverage* with respect to a particular input, initial program state, and safety property. An algorithm provides full coverage if it guarantees that no state reachable from the initial program state with the given input violates a particular safety property. An algorithm provides bounded coverage if it guarantees that no state reachable *within the bound* from the initial program state with the

given input violates the safety property. Safety properties include the absence of deadlocks and the absence of user-specified assertion failures.

Unit tests for concurrent abstractions are particularly well-suited to software model checking. Unit tests are typically small programs in which multiple threads access a shared concurrent data structure, which makes exploring their entire state space feasible. These tests often contain *cycles* in the state space, where a thread returns to the same state repeatedly until another thread's action breaks the cycle.

Enumerating *all* thread interleavings for such a program is impossible because the search may unroll the cycle indefinitely. Finding principled ways to break these cycles is an important component of prior work [Aggarwal et al., 1990, Peled, 1993, Peled, 1994, Godefroid and Wolper, 1994, Musuvathi and Qadeer, 2008]. Bounded search offers one solution to this problem for stateless model checkers.

Partial-order methods explore all states required to verify a given safety property. A unique behavior of a concurrent program corresponds to a *partial order* on the actions of its threads, whereas a thread interleaving is a *total order* on those actions. Thread interleavings with the same partial-order are equivalent with respect to the safety property: any bug that manifests under one such interleaving manifests under all such interleavings.

Partial-order methods explore at least one interleaving per partial order and thus explore all states that might violate the safety property without exploring all thread interleavings. Partial-order methods do not provide incremental coverage, however. If the tester terminates the search prematurely due to time constraints, then it provides no meaningful coverage guarantee. If the search arbitrarily prunes cycles in the state space, then the search sacrifices coverage.

Bounded search, in contrast, explores only states that are reachable within a bound. Within the bound, bounded search explores *all* thread interleavings. When the state space is large, this incremental coverage guarantee is useful. If thread

interleavings with small bounds are representative of those seen in practice, then bounded search prioritizes bugs likely to manifest in practice. Additionally, some bounds systematically prune cycles in the state space. Bounded search can thus provide incremental guarantees for cyclic state spaces. Without partial-order reduction, however, bounded search wastes most of its time exploring redundant thread interleavings. As the bound increases, this wasted time increases exponentially.

We compare bounded search with partial-order reduction and show that in practice, bounded search is useful only when the bound is very small. Time wasted exploring multiple interleavings of independent transitions quickly overwhelms the benefit of the bound. Partial-order reduction with no bound is both more effective and less time-consuming, if the search terminates acceptably quickly. If the search does not terminate quickly enough, practical bounded search requires partial-order reduction. This thesis extensively analyzes the space of possible dynamic, bounded partial-order reduction strategies. We show that naïvely combining bounded search with dynamic partial-order reduction sacrifices bounded coverage. We then show how to combine these approaches in three ways that preserve bounded coverage.

First, we develop an algorithm, compressed data structures, and heuristics to prioritize the state space using best-first search [Coons et al., 2010]. The best two heuristics (1) explore the reduced state space and prioritize executions that contain fewer preemptive context switches [Musuvathi and Qadeer, 2007a], and (2) explore the preemption-bounded state space and prioritize executions that modify the partial order. These heuristics provide fast response – bugs manifest quickly. Because they prioritize rather than prune the state space, the search eventually guarantees coverage. As the bound or the size of the state space increases, however, best-first search becomes impractical due to memory and time constraints. These results motivate using partial-order reduction to reduce, rather than prioritize, the bounded state space. Prior work suggests that combining these approaches while preserv-

ing bounded coverage is impractical for certain useful classes of bound functions, however [Musuvathi and Qadeer, 2007b].

We next reduce the bounded state space with partial-order reduction and preserve bounded coverage by exploiting dynamic information to make this combination practical. Prior work shows that *dynamic* partial-order reduction reduces the state space more effectively than *static* partial-order reduction does [Flanagan and Godefroid, 2005]. Static partial-order reduction relies on static analysis to predict whether accesses to shared data may be dependent, so these predictions must be conservative. Dynamic partial-order reduction determines whether accesses to shared data are dependent with one another explicitly at runtime. We exploit this dynamic information to identify dependences imposed by the bound function and show that we can reduce the size of the state space significantly by compensating for these dependences. Bounded partial-order reduction (BPOR) provides incremental coverage guarantees with a significantly reduced state space. With certain bound functions, this search also incrementally prunes cycles in the state space.

We develop a bounded partial-order reduction algorithm and specialize it for several bound functions including the number of context switches, the number of preemptive context switches [Musuvathi and Qadeer, 2007a], the number of deltas from an initial execution [Emmi et al., 2011], and a fairness bound that systematically prunes cycles in the state space [Musuvathi and Qadeer, 2008]. We prove that each algorithm guarantees bounded coverage. These algorithms incur significant overhead when compared to unbounded partial-order reduction because the bound introduces dependences between otherwise independent transitions.

We identify properties of bound functions that enable partial-order reduction. Prior work bounds properties of an execution – a total order on a program's transitions [Musuvathi and Qadeer, 2007a, Burckhardt et al., 2010, Emmi et al., 2011]. In Chapter 7, we observe that partial-order reduction relies upon the commutativity of

independent transitions. A bound on the total order sacrifices this commutativity. We solve this problem with bounds on the partial order. A bound on the partial order guarantees that independent transitions still commute. We describe a general approach to apply a bound to a partial order and use this technique to implement depth and context-bounded search without sacrificing any partial-order reduction.

Chapter 2 relates this work to methods for testing concurrent software including model checking and heuristic-based search. Chapter 2 then provides a formal language and semantics to describe concurrent programs as a system of transitions. The behavior of a program is determined by a partial order on its dependent transitions. We review the dependence relation and describe how it enables partial-order reduction. We then review bounded search including depth, context, and preemption bounds from prior work [Musuvathi and Qadeer, 2007a]. Because we build extensively on prior work, we discuss relevant related work throughout the thesis.

Chapter 3 empirically compares dynamic partial-order reduction to bounded search to motivate combining them. We additionally show why naïvely combining them sacrifices bounded coverage. We then describe an algorithm, compressed data structures, and heuristics to prioritize the state enumeration with best-first search. This chapter includes results for best-first search because they motivate bounded partial-order reduction – the heuristics that combine intuitions from bounded search and partial-order reduction are most effective at finding bugs quickly.

Chapter 4 presents conditions under which a set of transitions is sufficient to explore all local and deadlock states reachable within a bound. We define conditions for several bound functions and prove that they are sufficient. Chapter 5 presents an algorithm that computes these sufficient sets. We specialize this algorithm for several bound functions and prove that each algorithm computes a sufficient set of transitions in each state. In Chapter 6, we optimize bounded partial-order reduction to further reduce the state space. Together, these chapters provide a framework for

applying dynamic partial-order reduction to a bounded state space.

Chapter 7 proposes a general technique for bounding a partial order, rather than a total order on a program's transitions. We apply this technique to depth and context-bounded search. We prove that the resulting algorithms explore all states reachable within the bound without sacrificing partial-order reduction. These examples provide a framework for designing bounds that combine well with partial-order reduction.

Chapter 8 empirically compares different bound functions with and without partial-order reduction, and shows that the optimizations in Chapter 6 are effective. We compare the partial-order bounds introduced in Chapter 7 with their total order counterparts, and show that bounded partial-order reduction is significantly more effective with partial-order bounds. Chapter 9 discusses avenues for future work and finally, Chapter 10 discusses this work's results and their impact.

The state of the art dynamic partial-order reduction algorithm reduces the state space significantly but does not provide incremental guarantees or guarantees for cyclic state spaces [Flanagan and Godefroid, 2005]. State of the art fuzzing-based approaches often find bugs quickly but provide probabilistic coverage guarantees at best [Nagarakatte et al., 2012]. This thesis introduces algorithms, proves their coverage guarantees, and empirically demonstrates the following,

> *Bounded search can be combined with dynamic partial-order reduction to find bugs quickly and reproducibly with systematic, incremental coverage guarantees.*

This thesis makes bounded stateless model checking for concurrent programs substantially more efficient and practical.

Chapter 2

Background and Related Work

This section relates this work to strategies for testing concurrent programs including model checking and heuristic-based search. We formalize semantics to describe multithreaded programs, and review partial-order reduction and bounded search. We review dependence relations and show how persistent sets use them to reduce the state space while providing coverage guarantees. We review Dynamic Partial-Order Reduction (DPOR), an algorithm that dynamically computes persistent sets [Flanagan and Godefroid, 2005]. We then review bounded search. We define several bounds from prior work and discuss their advantages and disadvantages.

2.1 Testing Concurrent Programs

In this section, we relate our work to other methods for testing concurrent programs. Our work focuses heavily on coverage, so we first discuss model checkers for concurrent programs. We use heuristics to guide and bound the search, so we describe other heuristic-based approaches as well.

2.1.1 Model Checking

A model checker exhaustively verifies whether a system meets a specification [Clarke and Emerson, 1981]. We broadly divide model checkers into two categories: *symbolic* and *explicit* model checkers. Symbolic model checkers express sets of states and their transition relations as formulas. Successful symbolic model checkers have used binary decision diagrams (BDDs) to efficiently encode sets of states or used SAT solvers for exploration [McMillan, 1992, Burch et al., 1990, Yang and Dill, 1998, Cimatti et al., 2002, Biere et al., 1999]. We focus on explicit model checking, in which the search explicitly and exhaustively enumerates states.

Explicit model checkers exhaustively enumerate states and verify that a given property holds in each state. Many explicit model checkers explore an abstract model of the system, often expressed in a modeling language such as SPIN's Promela [Holzmann, 1997, Edelkamp et al., 2004, Aggarwal et al., 1990]. We focus on model checkers that explore the system's implementation directly [Godefroid, 1997, Havelund and Pressburger, 2000, Musuvathi et al., 2002, Leven et al., 2004, Visser et al., 2000b, Visser et al., 2000a]. Exploring the implementation directly eliminates potential errors in translating the system to the modeling language.

While exhaustively exploring the state space, some explicit model checkers keep track of the states they have visited [Havelund and Pressburger, 2000, Holzmann, 1997, Edelkamp et al., 2004, Musuvathi et al., 2002, Leven et al., 2004, Visser et al., 2000b, Visser et al., 2000a]. Stateful model checkers must encode and store states concisely. Inadequate storage space is a primary concern when exploring an exponential state space.

We focus on *stateless* model checking, which is better-suited to model checking large-scale software in which encoding states concisely may be difficult, and the state space is intractably large to store [Godefroid, 1997]. VeriSoft [Godefroid, 1997] and CHESS [Musuvathi and Qadeer, 2007a] both use stateless model checking to find

bugs in large software systems. Stateless model checkers are more practical for large software systems, but they suffer from an unacceptable blow up of the state space.

Explicit model checkers exhaustively explore the state space and thus encounter exponential *state space explosion*. Many model checkers use partial-order reduction to reduce the state space. Partial-order methods avoid exploring redundant states by exploring only one interleaving of independent transitions. Most partial-order reduction strategies from prior work use static analysis to identify transitions that may be dependent with one another [Godefroid and Wolper, 1992,Peled, 1994, Valmari, 1990, Bosnacki et al., 2006]. Partial-order methods apply to both stateful and stateless search, and to model checkers that check abstractions as well as model checkers that check implementations directly.

We focus on *dynamic* partial-order reduction, where the model checker detects dependences at runtime. We choose dynamic partial-order reduction because prior work shows it is far more effective at reducing the state space than static partial-order reduction is [Flanagan and Godefroid, 2005]. Dynamic partial-order reduction detects dependences more accurately than static partial-order reduction does because dynamic partial-order reduction exploits information available only at runtime. Dynamic partial-order reduction can thus identify more independent transitions and reduce the state space more aggressively.

The results of this work are clearly applicable to other stateless, explicit-state model checkers. These results may be less useful for stateful model checkers because their priorities are quite different. Reducing the overhead of tracking state is often the most important concern for stateful model checking algorithms. Still, the bounds we discuss and the heuristics we explore may be useful in that context. Next, we discuss heuristic-based search in more detail.

2.1.2 Heuristic-Based Search

Prior work uses heuristics to guide both model checking and fuzzing algorithms towards error states more quickly. Heuristics also guide the search towards shorter, simpler error states that are more useful to the tester. We first relate this work to heuristics for fuzzing algorithms, then to heuristics for model checking.

Heuristic-based fuzzing perturbs a program's thread schedule to manifest bugs [Edelstein et al., 2003, Sen, 2008, Sen, 2007, Park et al., 2009, Joshi et al., 2009]. Fuzzing is not systematic and does not provide coverage guarantees, but it often finds bugs quickly. We focus on coverage guarantees so that testers will have a useful guarantee when no bugs manifest. Fuzzing heuristics are still related, however, because they may provide good bound functions if they lead to bugs quickly. For example, CTrigger's bug triggering interleavings, designed to manifest atomicity violations, are similar to the preemption and context bounds that we describe in Section 2.7 [Park et al., 2009, Musuvathi and Qadeer, 2007a].

Many heuristic-based fuzzing strategies exploit random search because it finds bugs very effectively [Burckhardt et al., 2010, Joshi et al., 2009, Edelstein et al., 2003, Dwyer et al., 2007, Sen, 2008, Sen, 2007]. Random search does not naturally lend itself to systematic state space exploration with coverage guarantees. Parallel Randomized State-Space Search in Java PathFinder, however, shows that searching the state space in parallel from random locations within the state space is beneficial [Dwyer et al., 2007, Visser et al., 2000a]. This work could similarly benefit from randomized parallelization, as discussed in Chapter 9, but we leave that optimization for future work. In Chapter 3, we use a random priority function to guide the search randomly through the entire state space.

There is a long history of using heuristics in artificial intelligence and planning [Russell and Norvig, 2003, Pearl, 1984]. Our work is closely related and, in part, motivated by the success of heuristics in model checking [Yang and Dill,

1998, Edelkamp et al., 2001, Edelkamp and Jabbar, 2006, Rungta and Mercer, 2009]. These *directed* model checking algorithms use sophisticated analysis of the input program model to generate heuristics for guided search. These techniques have been studied in an explicit-state or symbolic-state setting. Translating these heuristics from stateful search to a stateless setting is not straightforward. For example, Yang and Dill [Yang and Dill, 1998] use Hamming distance to prioritize states that are closer to error states. Such heuristics are not applicable to stateless search. In Chapter 3, however, we use user-provided heuristics similar to *GuidePosts* in *SpotLight* [Yang and Dill, 1998].

Groce and Visser investigate guided model-checking for stateful search in Java PathFinder, and use several heuristics that are more applicable in stateless search [Groce and Visser, 2002]. For example, they include a thread interleaving heuristic that favors *more* preemptions to increase the variation in thread schedules, which allows their search to scale to more threads. We favor fewer preemptions because we find these executions are most useful to developers, as the preemptions often indicate the root cause of the bug. The state space is much smaller with fewer preemptions, which makes it possible to find errors more quickly. Preliminary work has been done in exploring heuristics in stateless search [Godefroid and Khurshid, 2002], but the genetic algorithms used in this work do not fare well when combined with partial-order reduction techniques nor do they provide the same soundness and progress guarantees.

These model checking strategies and concurrency testing heuristics from prior work motivate both best-first search and bounded partial-order reduction. Best-first search leverages these heuristics to guide the search towards new states quickly while preserving correctness guarantees. Bounded partial-order reduction reduces the state space while still guiding the search via the bound. In the next section, we formalize the terms we use to describe concurrent programs and their executions.

2.2 Multithreaded Programs and Semantics

This section formalizes the terms we use to describe concurrent programs and their executions. We use dynamic, stateless model checking to systematically explore the state space of multithreaded programs. A runtime scheduler systematically forces concurrent programs down different thread interleavings. The search stores only its current state as a stack of explored transitions.

A concurrent program contains a fixed set $\mathcal{T}id$ of thread identifiers and a set $\mathcal{T}$ of transitions. A *transition* $t \in \mathcal{T}$ is a tuple, $\langle tid, var, op \rangle$, that transfers the program from one state to another state. Thread $t.tid \in \mathcal{T}id$ performs transition t by executing operation $t.op$ on variable(s) $t.var$. Examples of operations include fork, join, lock acquire, lock release, and load/store operations. A transition may access multiple variables. For example, a wait operation may wait for a signal from multiple threads before it proceeds. For simplicity, we associate only one thread u and one operation op with each transition.

Intuitively, each program state is a graph whose edges indicate dependences between transitions. A *state* is a finite graph $\langle T, H \rangle$ where $T \subseteq \mathcal{T}$ is a set of transitions and H is an irreflexive partial order on T such that for each thread u, H is a total order on the set $\{t \in T \mid t.tid = u\}$. There exists a unique initial state $\langle T_0, H_0 \rangle$ where T_0 is the empty set and H_0 is the empty relation. A transition t transfers the program from a state $\langle T, H \rangle$ to a successor state, $\langle T', H' \rangle$, where $T' = T \cup \{t\}$, and H' is the partial order H with any additional orderings required by t. Definition 2.1 defines the *local state* for a thread u in a state $\langle T, H \rangle$.

Definition 2.1. Local state.

The *local state* for a thread u in a state $\langle T, H \rangle$ is the graph $\langle T_L, H_L \rangle$, defined as

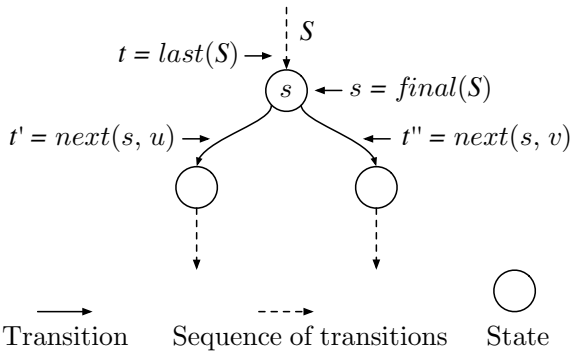


Figure 2.1: S is a sequence of transitions such that $s = \mathit{final}(S)$. Threads $u, v \in \mathit{enabled}(s)$.

follows where t is the last transition by u, if any:

$$T_L = \begin{cases} \emptyset & \text{if } t \text{ does not exist} \\ \{t' \in T \mid (t', t) \in H\} \cup \{t\} & \text{otherwise} \end{cases}$$

$$H_L = \begin{cases} \emptyset & \text{if } t \text{ does not exist} \\ \{(t'', t') \in H \mid t'' \in T_L \text{ and } t' \in T_L\} & \text{otherwise} \end{cases}$$

Intuitively, thread u's local state in state $s = \langle T, H \rangle$ is the subgraph of s that contains only those transitions that are ordered by s with respect to the most recent transition by u. The algorithms we present guarantee reachability of local states for finite, acyclic state spaces. For clarity, we use the term s as shorthand for an arbitrary state $\langle T, H \rangle$. The function $\mathit{local}(s, u)$ returns the local state for thread u in state s.

Figure 2.1 illustrates terms defined in this section. The term $\mathit{next}(s, u) \in \mathcal{T}$ denotes the transition that thread u will execute next from state s. We assume that the next transition for each thread from a given state is unique. A transition is *enabled* in s if it can execute from s. A thread u is enabled in s if $\mathit{next}(s, u)$ is enabled in s. The function $\mathit{enabled}(s)$ returns the set of all threads enabled in s. A state s in which $\mathit{enabled}(s) = \emptyset$ is a *terminal state*.

The expression $s \xrightarrow{t} s'$ indicates that transition t leads from state s to state s'. Using Flanagan and Godefroid's notation [Flanagan and Godefroid, 2005], a *transition sequence* S is a finite sequence of transitions $t_1.t_2.\ldots t_n$ such that there exist states $s_1, \ldots s_{n+1}$ where s_1 is the initial state s_0, and

$$s_1 \xrightarrow{t_1} \ldots \xrightarrow{t_n} s_{n+1}$$

The function $dom(S)$ returns the *domain* of S, the set $\{1 \ldots n\}$, and the length of S is $len(S) = n$. The term $final(S)$ refers to s_{n+1}, the final state reached after executing all transitions in S. Transition S_i is the ith transition in S, $i \in dom(S)$. Greek symbols ω, α, β, and γ, represent arbitrary-length sequences of transitions. The term $S.t$ denotes the sequence of transitions that results when transition t executes from $final(S)$, and $S.\alpha$ is the sequence that results when arbitrary-length sequence of transitions α executes from $final(S)$. An *execution* is a sequence of transitions where $s_0 = \langle T_0, H_0 \rangle$ and $enabled(s_{n+1}) = \emptyset$.

The rest of our definitions exactly follow Flanagan and Godefroid [Flanagan and Godefroid, 2005]. The behavior of a concurrent system is a transition system $A_G = (State, \Delta, s_0)$ where $State$ is the set of all possible states, $\Delta \subseteq State \times State$ is the *transition relation* defined by

$$(s, s') \in \Delta \text{ iff } \exists t \in \mathcal{T} : s \xrightarrow{t} s'$$

and s_0 is the system's unique initial state. Bounded search and partial-order methods each explore only a subset of A_G [Flanagan and Godefroid, 2005].

2.3 Traces

A *trace* is an equivalence class of sequences of transitions that can be obtained from one another by reordering adjacent independent transitions [Mazurkiewicz, 1986].

Algorithm 1 Basic search that explores A_G [Godefroid, 1996].

```
1: Initially, Explore(ε) from s_0
2: procedure Explore(S) begin
       # Recursively explore all enabled transitions.
3:     for all u ∈ enabled(final(S)) do
4:         Explore(S.next(final(S), u))
5:     end for
6: end
```

We use $[\omega]$ to denote the trace that contains the sequence of transitions ω. Any sequence of transitions in a trace uniquely denotes the trace.

Formally, using Godefroid's definition of traces [Godefroid, 1996], the concurrent alphabet for a system is the pair $\Lambda = (\mathcal{T}, D)$ where $\mathcal{T}$ is the finite set of transitions in the system, and D is the dependence relation. The relation $I_\Lambda = \mathcal{T}^2 \backslash D$ is the independency in Λ. Let ϵ denote the empty word. The relation $\equiv_\Lambda$ is the least congruence in the monoid $[\mathcal{T}^*; ., \epsilon]$ such that

$$(t, t') \in I_\Lambda \implies t.t' \equiv_\Lambda t'.t$$

We define a trace as follows,

Definition 2.2. Traces [Godefroid, 1996].
Equivalence classes of $\equiv_\Lambda$ are called *traces* over Λ. The term $[\omega]$ denotes the trace that contains the sequence of transitions ω.

We use the following theorem, proved by Godefroid, to reason about traces:

Theorem 1. *Let* final(S) *be a state in* A_G*. If* final$(S.\omega) = s$ *and* final$(S.\omega') = s'$ *in* A_G *and* $[\omega] = [\omega']$*, then* $s = s'$ *[Godefroid, 1996].*

2.4 Systematic Search

Algorithm 1 contains a basic search that explores A_G, the entire state space reach-

able from initial state s_0 for acyclic programs. We assume throughout this thesis that the state space is finite. In each state s, Algorithm 1 recursively explores the next transition by each thread enabled in s. All of the algorithms in this thesis build off of this basic algorithm for systematically exploring the state space.

The number of states that Algorithm 1 explores is exponential in the number of threads, and the length of the program. To check correctness properties for concurrent programs despite this exponential growth, prior work explores only a subset of the state space. In the next section we review partial-order methods that search a reduced yet sufficient state space by exploring only one interleaving of independent transitions.

2.5 Partial-Order Reduction

We introduce two classes of partial-order methods from prior work, which we combine with bounded search: *persistent sets* and *sleep sets* [Godefroid and Pirottin, 1993,Godefroid, 1990]. Persistent sets and sleep sets allow the search to explore only a subset of the enabled transitions in each state. By identifying transitions whose orderings may affect the program's behavior, the search provides safety guarantees despite exploring only a reduced state space.

Persistent sets reason about dependent transitions that may occur in the future, as the search explores the subsequent state space. Sleep sets, in contrast, reason about the search's past and guarantee that it does not return to a small subset of previously visited local states. Persistent sets store transitions that the search *will* explore because new states may be reachable via those transitions. Sleep sets determine transitions that the search *will not* explore because they lead to local states that the search has already explored.

Both persistent sets and sleep sets exploit *independent transitions* – transitions that may be reordered without affecting the program's behavior. A search that

identifies more independent transitions reduces the state space more effectively than a search that identifies fewer independent transitions. Thus, detecting independence has been an important focus of prior work [Godefroid, 1996].

Given a safety property and a set of assumptions about the state space, there exists a set of *valid* dependence relations that identify transitions whose interactions may affect that safety property. In this work, we present general techniques that search a subset of the state space, and are applicable to any valid dependence relation. In the next section, we define valid dependence relations, place restrictions on valid dependence relations for bounded search with partial-order reduction, and describe the particular dependence relation that we implement.

2.5.1 Dependence relation

A *dependence relation* identifies transitions whose interleavings may affect a multi-threaded program's behavior. The following definition characterizes any *valid dependence relation* for the transitions of a concurrent system:

Definition 2.3. Valid dependence relation [Flanagan and Godefroid, 2005] adapted from [Katz and Peled, 1992].
Let $\mathcal{T}$ be the set of transitions in a concurrent system and let $D \subseteq \mathcal{T} \times \mathcal{T}$ be a binary, reflexive, and symmetric relation. The relation D is a *valid dependence relation* for the system if and only if for all $t, t' \in \mathcal{T}, (t, t') \notin D$ (t and t' are independent) implies that the following properties hold for all states s of the system:

1. if $t \in \mathit{enabled}(s)$ and $s \xrightarrow{t} s'$, then $t' \in \mathit{enabled}(s)$ if and only if $t' \in \mathit{enabled}(s')$
2. if $t, t' \in \mathit{enabled}(s)$, then there is a unique state s' such that $s \xrightarrow{t.t'} s'$ and $s \xrightarrow{t'.t} s'$

Intuitively, the first property requires that independent transitions neither enable nor disable one another, and the second property requires that enabled independent

transitions commute. Any valid dependence relation must meet these criteria. We also assume that the dependence relation is *constant*, and do not consider conditional dependence relations [Godefroid and Pirottin, 1993]. Applying the algorithms we present to conditional dependence relations is beyond the scope of this work.

We implement the dependence relation in Definition 2.4. Definition 2.4 is valid because it does not permit any communication among independent transitions under our assumptions. We assume that if transitions do not communicate then they can neither enable nor disable one another, and they must commute.

Definition 2.4. Dependence relation.
Transitions $t, t' \in \mathcal{T}$ are *dependent*, $t \not\leftrightarrow t'$, if and only if

1. $t.tid = t'.tid$, or
2. $t.var \cap t'.var \neq \emptyset \wedge (IsWrite(t.op) \vee IsWrite(t'.op))$

If transition t is independent with transition t' then we write $t \leftrightarrow t'$. If t is dependent with t' then we write $t \not\leftrightarrow t'$. We apply the same notation to sequences of transitions. If all transitions in the sequence of transitions α are independent with transition t, for example, then we write $\alpha \leftrightarrow t$.

2.5.2 Persistent Sets

A *persistent set* T in each state s is a sufficient set of transitions to execute from s while preserving certain safety guarantees [Godefroid and Pirottin, 1993]. Persistent sets divide the enabled transitions in s into two groups: those in T and those not in T. All transitions reachable via transitions not in T, without executing any transitions in T, must be independent with all transitions in T. Formally, Godefroid defines persistent sets as follows:

Definition 2.5. Persistent sets [Godefroid and Pirottin, 1993].
A set $T \subseteq \mathcal{T}$ of transitions enabled in a state s is *persistent in* s if and only if for all

Algorithm 2 Selective search of A_R using persistent sets [Godefroid, 1996].

```
1: Initially, Explore(ε) from S_0
2: procedure Explore(S) begin
       # Recursively explore transitions in a nonempty persistent set.
3:     Let T = Persistent_Set(final(S))
4:     for all t ∈ T do
5:         Explore(S.t)
6:     end for
7: end
```

nonempty sequences α of transitions from s in A_G such that $\forall i \in dom(\alpha) : \alpha_i \notin T$ and for all $t \in T$, $t \leftrightarrow \text{last}(\alpha)$.

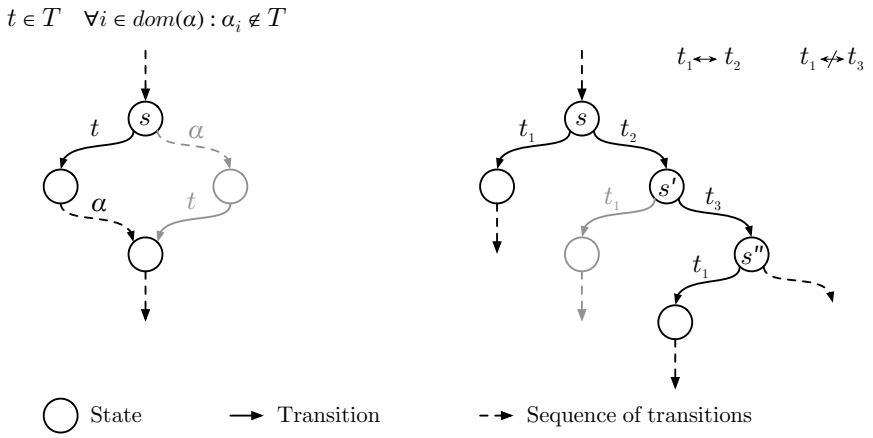
A *selective search* explores only a subset of the enabled transitions in each state. Algorithm 2 implements a selective search that explores a nonempty persistent set of transitions in each state [Godefroid, 1996]. This search explores a reduced state space $A_R \subseteq A_G$ that includes all local and deadlock states, provided that the state space is acyclic [Godefroid, 1996].

Figure 2.2(a) illustrates a transition t in the persistent set T in a state s. The sequence of transitions α contains only transitions that are not in T. The interleaving in gray need not be explored because the interleaving in black is equivalent and will be explored if needed.

2.5.3 Sleep Sets

Sleep sets are form of limited state caching [Godefroid and Wolper, 1992]. Sleep sets store visited transitions from prior states and prohibit them from executing again until the search explores a dependent transition. Assume that the search explores a transition t from a state $s = \mathit{final}(S)$, backtracks t, then explores t' from s instead. If $t \leftrightarrow t'$, then

$$\mathit{local}(\mathit{final}(S.t'.t), t.tid) = \mathit{local}(\mathit{final}(S.t), t.tid)$$



(a) T is persistent in s.

(b) t_1 is in the sleep set in s'.

Figure 2.2: Persistent sets and sleep sets; transitions in gray may be pruned.

This equation holds until the search explores a transition that is dependent with t. Thus, until the search explores a transition that is dependent with t, exploring t will lead to a previously visited local state. To prevent the search from exploring these redundant states, t "sleeps" until the search explores a dependent transition.

Figure 2.2(b) illustrates sleep sets. After the search explores t_1 and all states reachable via t_1 from s, it places t_1 in the sleep set for s. No new states become reachable via t_1 until the search performs a transition that is dependent with t_1. Thus, t_1 propagates to the sleep set in state s', because $t_1 \leftrightarrow t_2$. When the search explores t_3, however, it cannot propagate t_1 to the sleep set in s'' because $t_1 \not\leftrightarrow t_3$. New states may be reachable via t_1 from s'', so the search must explore t_1 from s''.

A family of algorithms reduce the size of the state space using some variation of persistent sets or sleep sets [Godefroid, 1996, Overman, 1981, Valmari, 1990]. Most of these algorithms are static; they use static analysis to determine which transitions may be dependent with one another. As a result, these algorithms must be conservative. Unless two transitions are known to always be independent, the search must assume that they may be dependent. In the next section, we present

an alternative approach that exploits dynamic information to reduce the size of the state space more aggressively.

2.6 Dynamic Partial-Order Reduction

Flanagan and Godefroid introduce Dynamic Partial-Order Reduction (DPOR), which dynamically computes persistent sets [Flanagan and Godefroid, 2005]. Flanagan and Godefroid show that DPOR computes smaller persistent sets than the best static methods, and thus achieves more state space reduction. Because DPOR is dynamic, it builds its persistent sets based on known dependences, while static partial-order reduction methods must make conservative assumptions about dependences based on static analysis of the source code. We provide intuition for DPOR here and provide greater detail in Chapter 5, where we build on this algorithm.

DPOR performs a depth-first search of the state space, always exploring the deepest unexplored transition. DPOR keeps track of the most recent access to each shared variable as it explores new transitions. In each state s, DPOR adds a *backtrack point* for the dependence between each thread's next transition t, and the most recently explored transition that is dependent with t, if such a dependence exists. This backtrack point allows the search to explore the dependent transitions in the opposite order in a future execution. Flanagan and Godefroid prove that DPOR explores a persistent set of transitions in each state [Flanagan and Godefroid, 2005].

Figure 2.3 illustrates two executions of DPOR. First, DPOR explores the transitions in Execution 1 until it reaches a deadlock state, where no transitions are enabled. During this initial execution, DPOR adds the backtrack points shown in Figure 2.3 under Execution 1. These backtrack points force DPOR to reorder the dependent transitions in a future execution.

After reaching a deadlock state in Execution 1, DPOR backtracks to the deepest backtrack point. Execution 2 shows the new execution that results. In

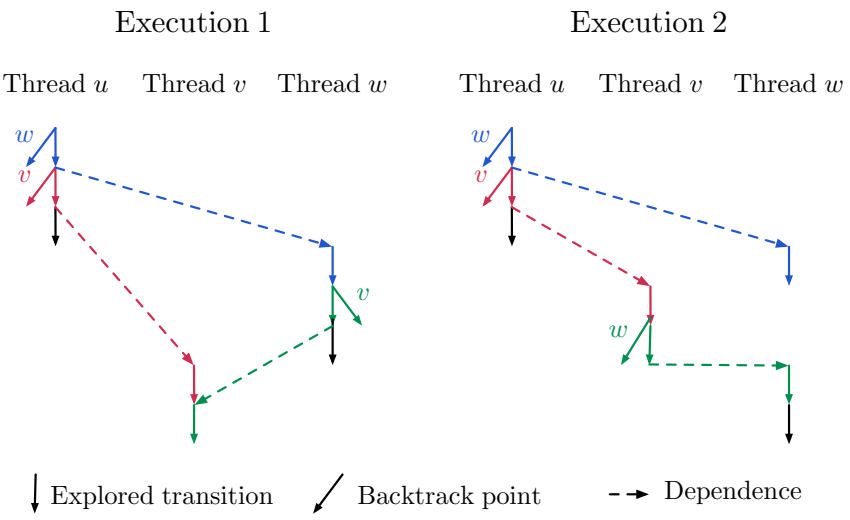


Figure 2.3: Two executions DPOR. Backtracking points appear prior to the first in each pair of dependent transitions.

Execution 2, DPOR executes Thread v prior to the dependent transition by Thread w, reversing the order of the dependent transitions. During Execution 2, DPOR adds a new backtrack point for Thread w, prior to the dependent transition by Thread v. This new backtrack point is redundant – it will swap the dependent transitions back to their original order. Sleep sets will prevent this redundant backtrack point because Thread w is in the sleep set prior to the dependent transition by Thread v.

DPOR reduces the state space more aggressively than static partial-order reduction does while still preserving persistent sets' safety guarantees. In contrast, bounded search reduces the state space, yet provides only bounded coverage. Bounded search explores only states that are reachable within the bound. The next section introduces bounded search and defines several bound functions. We describe their advantages and disadvantages, including how much partial-order reduction they permit.

Algorithm 3 Bounded search that explores $A_{G(Bv,c)}$.

```
1: Initially, Explore(ε) from s_0
2: procedure Explore(S) begin
     # Recursively explore all transitions that do not exceed the bound.
3:   for all u ∈ enabled(final(S)) do
4:     if Bv(S.next(final(S), u)) ≤ c then
5:       Explore(S.next(final(S), u))
6:     end if
7:   end for
8: end
```

2.7 Bounded Search

In contrast to partial-order methods, which provide full coverage, bounded search provides *bounded coverage* for acyclic state spaces. Bounded search limits the size of the state space because it does not explore transitions that exceed a bound. This bound may be any property of a state, or of the sequence of transitions that led to that state. The bound is computed by a *bound evaluation function*, Bv. The function $Bv(S)$ returns the *bounded value* for the transition sequence S. Bounded search requires two parameters: the bound evaluation function Bv, and the bound c, which may be any nonnegative, comparable value.

Algorithm 3 performs bounded search of $A_{G(Bv,c)}$, the full state space reachable with bound function Bv within bound c from initial state s_0. This search requires that the state space be acyclic. Algorithm 3 explores all enabled transitions in each state s that do not exceed the bound from s. Algorithm 3 is similar to Algorithm 1, but at Line 4, Algorithm 3 explores $\mathit{next}(\mathit{final}(S), u)$ from $\mathit{final}(S)$ only if $\mathit{next}(\mathit{final}(S), u)$ does not exceed the bound from $\mathit{final}(S)$. The *bounded state space*, $A_{G(Bv,c)} \subseteq A_G$, is an automaton

$$A_{G(Bv,c)} = (State_{(Bv,c)}, s_0, \Delta_{(Bv,c)})$$

where $State_{(Bv,c)}$ is the set of all states reachable from initial state s_0 via a sequence S of transitions such that $Bv(S) \leq c$. The relation $\Delta_{(Bv,c)} \subseteq State \times State$ is the *bounded transition relation* defined by

$$(s, s') \in \Delta_{(Bv,c)} \text{ iff } \exists t \in \mathcal{T} : s = \mathit{final}(S) \wedge s' = \mathit{final}(S.t) \wedge Bv(S.t) \leq c$$

We focus on *monotonic* bound evaluation functions, which we define as follows:

Definition 2.6. Monotonic bound functions [Musuvathi and Qadeer, 2007b]. A bound evaluation function Bv is *monotonic* if and only if for all sequences of transitions S and for all transitions $t \in \mathit{enabled}(\mathit{final}(S))$, $Bv(S) \leq Bv(S.t)$.

Musuvathi and Qadeer show that monotonic bound evaluation functions provide bounded coverage. With monotonic bound evaluation function Bv and bound c, bounded search visits all states s such that there exists a sequence S of transitions from s_0 such that $s = \mathit{final}(S)$ and $Bv(S) \leq c$, i.e., all states reachable in $A_{G(Bv,c)}$ [Musuvathi and Qadeer, 2007b]. Monotonicity is a sufficient condition for Algorithm 3 to provide bounded coverage, provided that the state space is acyclic.

The size of the bounded state space grows exponentially with the number of threads and with the bound. The bound is thus a useful parameter to control the search's state-space explosion. Because the size of the state space grows exponentially with the bound, ideally most bugs should manifest with a small bound. The bound also provides a useful coverage metric; the search guarantees its safety property for all states reachable within the bound. Next, we describe several monotonic bound evaluation functions and discuss their strengths and weaknesses.

2.7.1 Depth-Bounded Search

The *depth bound* limits the depth of each execution that the search explores. The depth bound is defined recursively as follows:

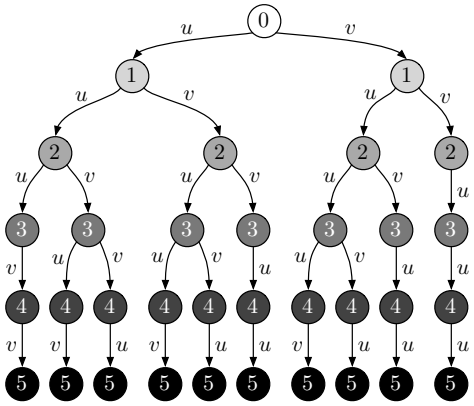


Figure 2.4: Depth-bounded state space exploration.

Definition 2.7. Depth bound (*Df*).

$Df(t) = 1$

$Df(S.t) = Df(S) + 1$

Figure 2.4 illustrates depth-bounded search for a small program with two threads, u and v. Circles represent states, arrows represent transitions, and the number in each state is the bounded value for the sequence that led to that state. Light colored states can be reached with smaller bounded values than dark colored states can.

Depth-bounded search is simple to implement and easy to understand. Any two paths to the same global state have the same bounded value in depth-bounded search, and partial-order reduction algorithms can exploit this property. Depth-bounded search also provides a workaround for cyclic state spaces by pruning executions after a given number of transitions.

Depth-bounded search also has several disadvantages. As shown in Figure 2.4, depth-bounded search arbitrarily biases the search towards early portions of the state space. This bias is often undesirable, particularly for shared-memory systems. Shared-memory systems often contain many accesses to shared data and bugs are no more likely to manifest in early portions of an execution than they are

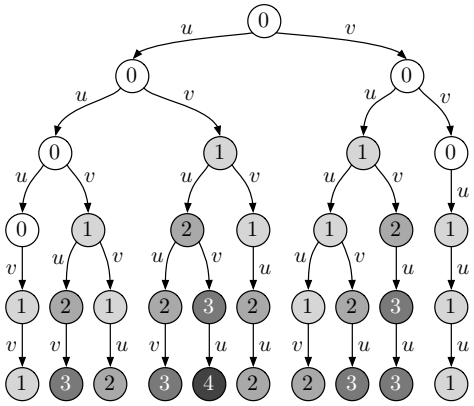


Figure 2.5: Context-bounded state space exploration.

in any other part of an execution. Depth-bounded search also inhibits local state reachability. Two sequences of transitions that lead to the same local state may not have the same depth.

2.7.2 Context-Bounded Search

Context-bounded search explores sequences of transitions that contain up to c context switches [Musuvathi and Qadeer, 2007a]. Given a sequence of transitions S from initial state s_0, the context bound is defined recursively as follows:

Definition 2.8. Context bound (Cs).

$$Cs(t) = 0$$

$$Cs(S.t) = \begin{cases} Cs(S) + 1 & \text{if } t.tid \neq \text{last}(S).tid \\ Cs(S) & \text{otherwise} \end{cases}$$

Figure 2.5 illustrates context-bounded search. The context-bounded state space becomes reachable within the bound from the top of the state space downward, much like the depth-bounded state space does. The context bound increases more slowly than the depth bound does, however, provided that the same thread executes

repeatedly. More of the state space is reachable with a small context bound than is reachable with a small depth bound. This property is desirable because the size of the state space grows exponentially with the bound. Both context and depth-bounded search may reach states in which all threads exceed the bound, however, systematically leaving deeper portions of the state space unreachable.

The size of the state space grows exponentially with the number of context switches in context-bounded search, rather than the length of each execution. Thus, context-bounded search may explore deep into the state space with a relatively small bound. With a small bound, the search experiences less state space explosion. Thus, context-bounded search can often explore more unique states than depth-bounded search can before state-space explosion significantly slows progress.

Two paths to the same deadlock or local state may contain different numbers of context switches. Thus, partial-order reduction algorithms do not combine easily with context-bounded search. Because the context bound does not increase when the same thread executes repeatedly, the context bound does not prune cycles in the state space. Thus, cyclic state spaces pose a problem for context-bounded search.

2.7.3 Preemption-Bounded Search

Preemption-bounded search limits the number of *preemptive* context switches in an execution [Musuvathi and Qadeer, 2007a]. The preemption bound is defined recursively as follows by Musuvathi and Qadeer [Musuvathi and Qadeer, 2007b]:

Definition 2.9. Preemption bound (*Pb*).

$$Pb(t) = 0$$

$$Pb(S.t) = \begin{cases} Pb(S) + 1 & \text{if } t.tid \neq \text{last}(S).tid \text{ and last}(S).tid \in \mathit{enabled}(\mathit{final}(S)) \\ Pb(S) & \text{otherwise} \end{cases}$$

In contrast to depth-bounded and context-bounded search, preemption-bounded

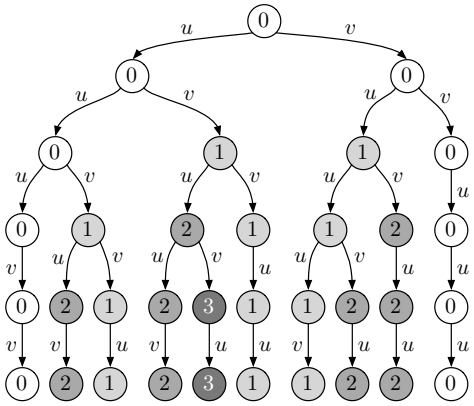


Figure 2.6: Preemption-bounded state space exploration.

search, as shown in Figure 2.6, does not bias the search towards early portions of the state space. Instead, preemption-bounded search explores the state space from zero-preemption executions outwards.

Preemption-bounded search has several advantages over depth-bounded and context-bounded search. The preemption bound *always* provides a zero-cost path to a terminal state. As a result, more states are reachable within a small bound in preemption-bounded search than in depth-bounded or context-bounded search. The preemption bound often provides a more useful coverage metric than these bounds do because the preemption bound does not arbitrarily bias the search towards early portions of the state space. In addition, if few preemptions occur in practice, then the preemption bound may bias the search towards executions that are likely to occur in practice.

Preemption-bounded search has several disadvantages. Because different paths to the same deadlock or local state may contain different numbers of preemptions, preemption-bounded does not combine easily with partial-order reduction algorithms. Additionally, the cost of a transition in preemption-bounded search varies with the enabledness of the prior transition. Threads that enable or disable other

threads may therefore introduce dependences, which inhibit partial-order reduction. Finally, preemption-bounded search does not prune cycles in the state space, and thus cannot handle cyclic state spaces.

2.7.4 Delta-Bounded Search

Delta-bounded search limits the number of deltas from an initial, deterministic execution. Delta-bounded search is similar to delay-bounded search, which limits the number of delays that an otherwise-deterministic scheduler is allowed [Emmi et al., 2011]. Emmi et al. find that delay-bounded search manifests bugs quickly, requiring fewer executions than preemption-bounded search requires. The delta bound we use is a specific type of delay bound.

Delta-bounded search increments the bounded value once for each difference from an initial, deterministic execution. We implement delta-bounded search with a round-robin scheduler. Initially, the search associates an arbitrary, unique priority with each thread. In each state, the search explores the highest priority enabled thread. To explore a thread u from a state s, the search rotates the thread priorities in s such that u has the highest priority. The cost to make u the highest priority thread in s is equal to the number of enabled threads in s with higher priority than u. We define the delta bound recursively as follows:

Definition 2.10. Delta bound (De).

$De(t) = \textbf{PriorityDiff}(s_0, t.tid)$

$De(S.t) = De(S) + \textbf{PriorityDiff}(\mathit{final}(S), t.tid)$

The function **PriorityDiff**($\mathit{final}(S), u$) for state $\mathit{final}(S)$ and thread u returns the number of enabled threads in $\mathit{final}(S)$ that have higher priority than u. Figure 2.7 illustrates delta-bounded search. The delta bound provides a single zero-cost path to a terminal state. Delta-bounded and preemption-bounded search explore the

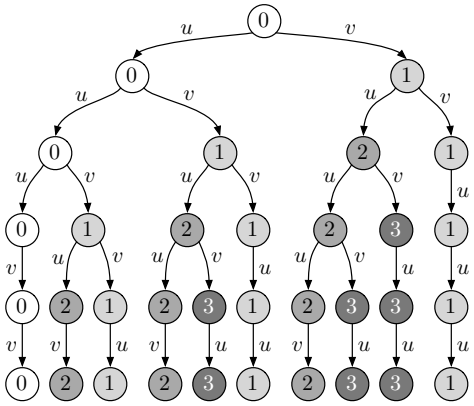


Figure 2.7: Delta-bounded state space exploration.

state space in a similar manner to one another. With the delta bound, however, each enabled transition has a unique cost in each state.

We choose to increment the bound by the number of higher priority threads rather than incrementing it by one to ensure that each thread's cost is unique in each state with the delta bound. We hypothesize that a unique cost for each transition may make exploring the cheapest transition first a good heuristic for guiding delta-bounded search to states via the cheapest path first. In Chapter 8, we reach mixed conclusions about this decision. The unique cost is helpful for some state spaces, but the search must significantly sacrifice partial-order reduction for others.

Delta-bounded search always provides a zero-cost path to a terminal state, so it does not bias the search towards early portions of the state space. Delta-bounded search does, however, bias the search towards executions that are similar to the initial execution. If the default scheduler is similar to schedulers encountered in practice, this bias may be an advantage. Additionally, the search may be parallelized by searching from distinct initial schedules. The delta bound does not prune cycles in the state space. Next, we introduce a bound that prunes unfair executions to limit the number of times the search may unroll a cycle in a cyclic state space.

2.7.5 Fair-Bounded Search

Fair-bounded search limits the number of times that the search may unroll a cycle in a cyclic state space. We adapt a fairness criteria from prior work to identify cycles in stateless search given two assumptions [Musuvathi and Qadeer, 2008]:

1. a thread always yields the processor if it is not making progress
2. a thread never yields the processor if it is making progress

In each state s, fair-bounded search tracks the number of yield operations that each thread u has performed, $Yc(s, u)$. The fair bounded value in a state $s = \mathit{final}(S)$ is equal to the maximum observed difference between the executing thread's yield count and each other enabled thread's yield count in each state reached by S. Formally, we define the fair bound recursively as follows:

Definition 2.11. Fair bound (*Fb*).
Let $Yc(S, u)$ return Thread u's yield count in $\mathit{final}(S)$.
$Fb(t) = 0$

$Fb(S.t) = max(Fb(S), \max_{u \in enabled(\mathit{final}(S))}(Yc(S, u) - Yc(S, t.tid)))$

Figure 2.8 illustrates fair-bounded search in a cyclic state space. Given our assumptions about yield operations, the set of states reachable via the transitions with bound zero is not meaningfully different from the set of states reachable with bound one, etc. The search explores a cycle in the state space, returns to a previously visited state, and begins to explore the entire state space reachable from that state again. In Figure 2.8, Thread u repeatedly performs an operation that is dependent with an operation by Thread v. For example, Thread u reads a shared variable repeatedly until Thread v changes the result. A non-preemptive scheduler will allow Thread u to execute indefinitely, as shown by the leftmost path in Figure 2.8.

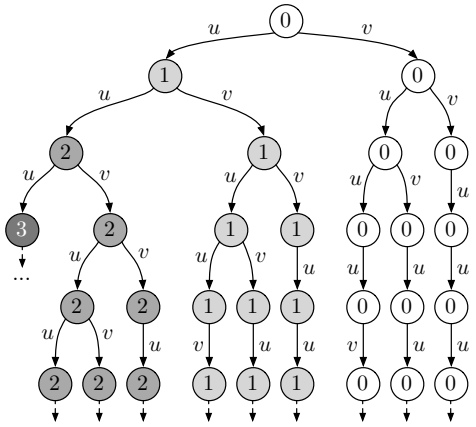


Figure 2.8: Fair-bounded state space exploration.

Given the assumptions listed above, Thread u must perform a yield operation during each cycle. Thus, eventually, Thread u will have a yield count that is enough greater than Thread v's yield count to exceed the fair bound. Fair-bounded search allows Thread v to execute, breaking the cycle. Note that if Thread u does not yield the processor, then the search continues to execute Thread u until it reaches a depth bound and reports a livelock. This user-supplied depth bound must be very large such that it constrains the search only when the search enters a cycle in the state space that the fair bound cannot break.

When Thread v executes, it performs an action that breaks Thread u's cycle. The reachable state space in Figure 2.8 does not meaningfully change as Thread u unrolls its cycle repeatedly. Given our assumptions regarding when threads yield the processor, any bug that manifests after $n+1$ cycles will also manifest after n cycles. Fair-bounded search prunes executions that unroll a cycle more than n times.

Two sequences of transitions that lead to the same local state may contain different values for $Yc(S, u)$, so the fair bound does not easily combine with partial-order reduction. Additionally, a transition's cost varies with the number of *enabled* transitions with a lower yield count. Thus, the fair bound may introduce depen-

dences between otherwise independent transitions if they enable or disable threads with a lower yield count.

There exist many fairness criteria that produce similar results. We selected this fairness criterion primarily for its simplicity, but other fairness criteria can be expressed as bounds, as well. We show that a fairness criterion can be implemented as a bound function and that it combines with partial-order reduction similarly to other bound functions. We restrict partial-order reduction to accommodate dependences that the fairness criterion imposes on otherwise independent transitions.

Prior work uses fairness criteria for cyclic state spaces in stateful search. For example, Holzmann and Peled include provisos to close cycles in the state space only from states in which the search explores all enabled threads [Holzmann et al., 1992, Peled, 1993, Peled, 1994]. We exploit similar insights to combine fair-bounded search with DPOR. Dynamic search offers more opportunities to reduce the state space, but stateless search makes cycle detection more challenging.

2.8 Discussion

Prior work evaluates the depth, context, and preemption bounds, and existing systems use them in practice [Musuvathi and Qadeer, 2007a, Holzmann, 1997, Musuvathi and Qadeer, 2008, Musuvathi et al., 2009]. All of these bounds prune portions of the state space and may therefore fail to detect bugs in programs. Partial-order methods, in contrast, explore the entire relevant state space and detect all bugs that violate the correctness guarantee.

If the state space is very large, however, and the search does not terminate within a reasonable time period, then partial-order methods do not provide any guarantees. Ideally, combining these techniques would provide incremental coverage guarantees for a reduced state space. The next section compares these techniques, highlights their advantages and disadvantages, and shows why combining them is

not trivial. Then, we determine whether combining them will find bugs quickly by exploiting best-first search.

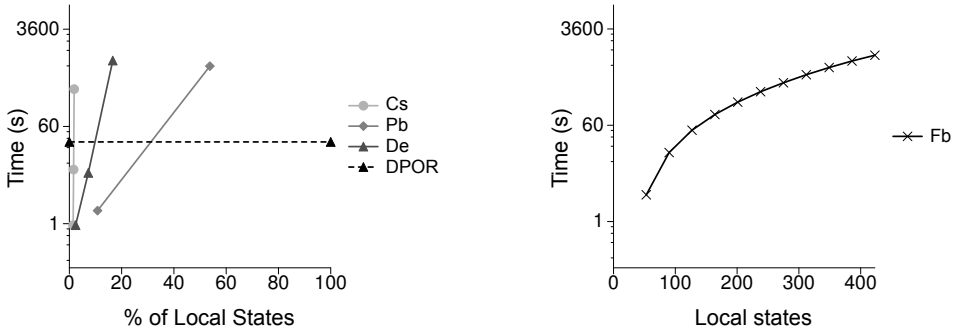
Chapter 3

Best-First Search

This section uses best-first search to combine intuitions from bounded search and dynamic partial-order reduction (DPOR). We first compare bounded search with DPOR and show why naïvely combining these techniques sacrifices bounded coverage. We show that bounded search scales poorly without partial-order reduction, and that DPOR is a better choice when the state space is small enough to be explored in its entirety. If the state space is intractably large, however, then DPOR provides no guarantees. These results motivate combining DPOR with bounded search to provide incremental guarantees. To further motivate this approach, we prioritize the state space for both bounded search and DPOR using best-first search, and implement heuristics to detect bugs quickly with a tool called Gambit.

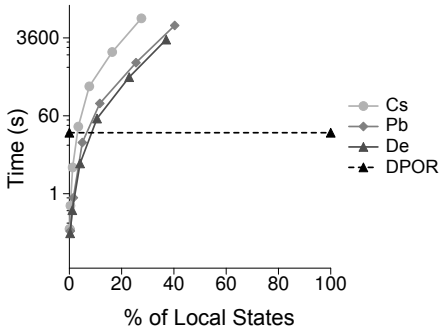
3.1 Partial-Order Reduction for Bounded Search

We compare bounded search with DPOR and show that bounded search requires more time and explores less of the state space than DPOR does. We implement DPOR, as described in Section 2.6, in Chess, a model checker for concurrent programs that also performs bounded search [Flanagan and Godefroid, 2005,Musuvathi



(a) FFT

(b) Fair



(c) MRSE

Figure 3.1: Coverage vs. time for bounded search without partial-order reduction.

and Qadeer, 2007a]. We compare state space coverage over time for each technique on small programs and show that bounded search provides little benefit without partial-order reduction, except when the bound is very small.

Figure 3.1 compares bounded search to DPOR. The x-axis shows the percent of all local states that the search visits, and the y-axis shows the time in seconds that the search requires to explore them. Each data point represents an invocation of CHESS with a particular value for the bound, which we iteratively increase. These results show the limitations of bounded search. Without partial-order reduction,

bounded search provides little benefit. DPOR explores the entire state space in less time than bounded search requires to explore the small subset of the state space reachable with a bound of one or two. The fair bound in Figure 3.1(b) offers an exception to this rule. Without the fair bound, the cyclic state space that we test in Figure 3.1(b) never terminates with DPOR, and thus we provide no results for DPOR for this test.

Each test in Figure 3.1 is very small. A larger state space will benefit from bounded search because the entire state space using DPOR becomes intractably large. Without searching the entire space, DPOR provides no guarantees. Bounded search, in contrast, may explore only a fraction of the state space, but it provides an incremental guarantee that programmers may find useful.

Context and preemption-bounded search make large state spaces tractable, but only with a bound of zero or possibly one. With higher bounds, the bounded state space becomes too large, leaving most of the state space unexplored. DPOR finds new states far more quickly than bounded search does. Without a bound, however, DPOR provides an all-or-nothing guarantee. If the state space is intractably large, then DPOR runs for a very long time without providing any guarantees.

Practical bounded search requires DPOR's aggressive state space reduction. Unfortunately, the bounds introduced in Section 2.7 are unsound when naïvely combined with DPOR. Sequences of transitions that lead to the same local or deadlock state may have different bounded values, and the search therefore cannot guarantee that it has taken the cheapest path to a given state. DPOR may prune transitions that make new states reachable within the bound, sacrificing coverage.

Figure 3.2 illustrates a scenario in which preemption-bounded search with DPOR is unsound. Although s' is reachable within the bound via the sequence of transitions $t_3.t_1.t_2$, the search never reaches s'. DPOR never adds t_3 to the backtrack set in s because t_3 is not dependent with t_1; it is dependent with t_2.

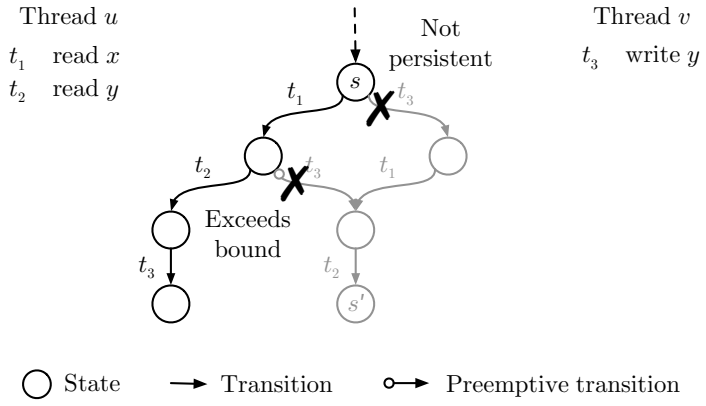


Figure 3.2: Preemption-bounded search with naïve DPOR Although s' is reachable within the bound, the search never reaches it.

DPOR instead adds t_3 to the backtrack set in $\mathit{final}(S.t_1)$. From there, however, t_3 requires a preemption, so it exceeds the bound and the search does not explore it.

DPOR sacrifices bounded coverage because the bound introduces dependences between instructions that are otherwise independent. If a transition t exceeds the bound in a state s then the search cannot explore t from s and t is, in some sense, "disabled" in s. Any transition that alters t's bounded value in s is thus dependent with it, by Definition 2.3 of valid dependence relations.

The dependences that the bound introduces are not the same as dependences within the program under test, however. Bound dependences are artificial. Programmers do not generally care whether their programs are capable of exceeding the bound or not, they care whether or not executions within the bound adhere to the safety property.

If the search treats bound dependences equivalently to dependences in the tested program, then the search must explore each state with *all* possible bounded values, which is an enormous waste. Thus, we differentiate these dependences. Bounded search need not explore both orders of bound dependent transitions if those

transitions are otherwise independent and all states reachable within the bound via one are also reachable within the bound via the other. We therefore keep these dependences separate from one another. A transition that exceeds the bound is not disabled in the same way that a transition waiting for another thread to release a lock is disabled. Still, the search must compensate for these bound dependences when there exists a cheaper path to a given state.

We use best-first search to determine whether combining bounded search with DPOR is beneficial. Best-first search guides DPOR towards executions with a lower bounded value, and guides bounded search towards executions likely to contain new partial orders. We evaluate various best-first heuristics to guide the search, and we find that the heuristics that combine partial-order reduction with bounded search are effective.

3.2 Best-First Search

We prioritize the state space for bounded search and DPOR using best-first search. We implement heuristics to detect bugs quickly in a tool called Gambit and apply them to both DPOR and bounded search. As shown in Section 3.1, bounded search sacrifices bounded coverage with DPOR. Musuvathi and Qadeer show that determining the context bound of a state – the fewest context switches with which it is possible to reach that state – is NP-complete [Musuvathi and Qadeer, 2007b].

Rather than compute the cheapest path to each state, DPOR with best-first search *prioritizes* executions that do not exceed the bound. We choose the preemption bound for these tests because prior work shows that it finds bugs quickly if they are reachable with a small bound [Musuvathi and Qadeer, 2007a]. We also use DPOR as a priority function. Rather than pruning transitions not in the backtrack set, we use the backtrack set to guide best-first search toward executions likely to contain new partial orders. Eventually, the search explores the entire full or bounded

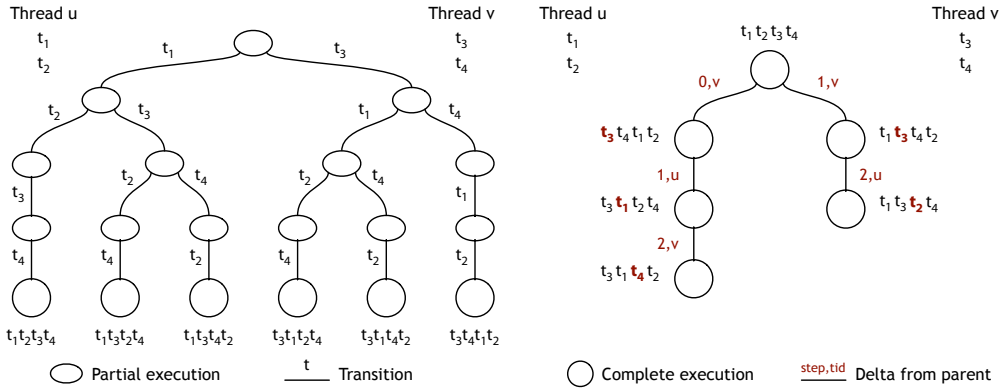
state space and thus preserves coverage.

Best-first search maintains an open list and a closed list of nodes [Korf et al., 2005]. We refer to the open list as the *fringe*. Nodes in the fringe have not yet been explored, and their successors have not yet been generated. Closed nodes have been explored and their successors have already been generated. Best-first search selects the highest priority node from the fringe in each iteration, generates all of its successors, and adds it to the closed list. Each newly generated successor that is not already closed or in the fringe is added to the fringe. The search terminates when all nodes are closed. Because best-first search's storage overhead scales with the size of the graph, we introduce a compressed representation of the state space that exploits properties of bounded search and DPOR.

3.3 Execution Trees

An *execution tree* exploits the fact that DPOR and bounded search explore *sparse* trees. We choose trees rather than graphs because the search is stateless and the model checker is thus unaware when the search returns to a previously visited state. The search tree is sparse in bounded search because bounded search cannot explore any transitions that exceed the bound. The search tree is sparse with DPOR because DPOR prunes many of its edges. Because the state space is a tree we do not need to track closed nodes – we close nodes by deleting them and rely on the fair bound for cyclic state spaces.

The execution tree's storage overhead scales with the number of unique program *executions*, rather than the total number of transitions. Figure 3.3 compares the uncompressed state space for a simple program to its execution tree. The execution tree leverages systematic search by storing only deltas from prior executions. The sequence of transitions next to each node is the execution that node represents. Bold transitions highlight the step at which the delta occurs.



(a) Uncompressed state space.

(b) Execution tree.

Figure 3.3: Compressed and uncompressed state space for two threads each performing two operations.

We assume that the model checker systematically varies a *default behavior*. By default, we use a non-preemptive round robin scheduler. This scheduler never preempts the executing thread, and it always schedules the next enabled thread, in order by thread identifier, when the executing thread blocks. The root node in Figure 3.3(b) represents this unique default behavior. The two edges to its successors indicate steps at which the search must explore a different transition: at steps 0 and 1, the search must explore a transition by thread v. The root node's successor nodes represent executions that always perform the default behavior except at these steps.

The model checker can replay any execution by performing the default behavior except at the steps indicated along the edges leading to that node. Each node in Figure 3.3(b) is equivalent to a leaf node in Figure 3.3(a). The execution tree compresses away steps at which the model checker performs its default behavior. In a sparse search like DPOR or bounded search, this compression provides significant benefit. We perform a best-first search on this execution tree.

Algorithm 4 Best-first search.

```
 1: procedure BestFirstSearch() begin
 2:   Add root to fringe
 3:   while node := fringe.Next() do
 4:     Let S := Execute(node)
 5:     for i ∈ dom(S) do
 6:       for u ∈ enabled(pre(S, i)) do
 7:         if DoBacktrack(pre(S, i), u) then
 8:           succ := node.CreateSuccessor(i, u)
 9:           fringe.Insert(succ, GetPriority(S, i, u))
10:         end if
11:       end for
12:     end for
13:     if node.numSuccessors == 0 then
14:       node.Detatch()
15:     end if
16:   end while
17: end
```

3.4 Best-First Search Algorithm

Algorithm 4 performs best-first search using execution trees. Each *node* stores a link to its parent, its successor count, the step at which it diverges from its parent, and the thread whose next transition it must explore at that step. The **BestFirstSearch** procedure iterates through all nodes in the fringe beginning with the root node. The **Next** method removes and returns the highest priority node in the fringe at Line 3. The **Execute** procedure at Line 4 runs the program as indicated by *node*, and returns the sequence of transitions that results.

Lines 4-10 iterate over each step $i \in dom(S)$ and Lines 6-11 consider each thread $u \in enabled(pre(S, i))$. Line 7 checks whether u must execute from $pre(S, i)$. DPOR's **DoBacktrack** procedure returns true at Line 7 if u is in the backtrack set in $pre(S, i)$. Bounded search's **DoBacktrack** procedure returns true at Line 7 if u's next transition does not exceed the bound from $pre(S, i)$. If Line 7 returns true, then Line 8 creates a successor node for u at $pre(S, i)$ and Line 9 adds it to the fringe.

After generating new nodes, Line 14 deletes *node* if it has no successors. If a node has no successors, then all of the state space reachable from it has been explored and *node* is closed. The **Detatch** method at Line 14 removes *node* from the execution tree by decreasing its parent's successor count. If, as a result, its parent has no successors, then *node* recursively detaches its parent. Thus, nodes delete themselves as soon as they become unnecessary. When the search terminates, the tree is empty aside from the root node. The tree discovers itself dynamically and deletes itself dynamically, so the entire tree is never in memory. Every leaf node is in the fringe and must still execute, and every interior node has already executed and has at least one descendent leaf node in the fringe.

The space required by the execution tree scales with the number of open nodes. GAMBIT can run for days using best-first search without running out of memory. GAMBIT could write low priority nodes to disk and expand its state space further – nodes in the fringe are not accessed until they are ready to execute, so writing low priority ones to disk should have low performance impact. We do not explore this option, however, because we find that when the size of the state space becomes too large the time required to search the entire state space is also too large, so the test does not terminate. We next explore priority functions that guide the search toward bugs.

3.5 Priority Functions

A priority function dictates the order in which GAMBIT explores new executions. Any priority function is admissible, but the best priority functions manifest bugs quickly. A priority function can use any approach to target the search, for example:

1. Find new local states at a faster rate

2. Randomize the search with progress guarantees

3. Prioritize the search based on tester input

4. Target known bug patterns

We implement priority functions using the first three approaches. Heuristics from prior work target known bug patterns [Park et al., 2009, Joshi et al., 2009], and GAMBIT could use those heuristics as well, but we do not evaluate them here.

3.5.1 Prioritizing New Local States

We prioritize new local states in two ways. When searching the bounded state space, bounded search does not reduce the state space with DPOR because doing so might sacrifice bounded coverage. The priority function instead *prioritizes* transitions that lead to new partial orders. Similarly, when searching the reduced state space with DPOR, the search does not bound the number of preemptions in each execution because doing so sacrifices coverage. The priority function instead *prioritizes* executions with fewer preemptions. We use the following priority functions to explore these options:

BF(*Pb*) prioritize transitions that require fewer preemptions

BF(*Dpor*) prioritize transitions that are in their state's DPOR backtrack set

BF(*Ss*) prioritize transitions that are not in their state's sleep set

where **BF** stands for "best-first". These priority functions combine reduction techniques to prioritize the search without sacrificing coverage. Note that BF(*Dpor*) returns transitions that are not in their state's DPOR *backtrack set*. In bounded search, this set is not a persistent set, and thus we refer to the set of transitions that DPOR selects as its backtrack set.

3.5.2 Random Search

Random search is often very effective at exploring large state spaces [Dwyer et al., 2007]. A random stateless search provides no progress guarantees, however, and thus sacrifices coverage. GAMBIT randomizes the search while guaranteeing progress and coverage by randomly walking a program's execution tree:

BF(*Random*) assign a random priority to each execution

This simple priority function ensures that the search does not linger in uninteresting parts of the state-space, yet still guarantees progress because it randomizes only the *order* of the search. The search is not entirely random; it is biased towards the initial execution. Still, we show that it finds many bugs quickly.

3.5.3 Tester Input

Unit tests typically test a specific function, behavior, or data structure. Unit tests for concurrent data structures may test how specific methods interact. For example, a unit test may simultaneously **Enqueue** and **Dequeue** from a concurrent queue. Regression testing often targets changes to specific methods or variables, as well. In these scenarios, the tester can guide the search by specifying methods or variables at the command line and using the following priority functions:

BF(*Method*) prioritize transitions within specified methods

BF(*Var*) prioritize transitions that access specified variables

BF(*Var*) may also help target known data races, because the tester can specifically re-order the racy accesses. We provide these priority functions as examples, but adding new priority functions is simple. We also make these individual priority functions more powerful by combining them hierarchically.

3.5.4 Hierarchical Priority Functions

We combine individual priority functions hierarchically into a single value. The priority function listed first receives the highest priority. When the first priority value is equivalent, ties are broken by the second priority function, etc. Commas indicate hierarchical priority functions. For example, the priority function BF(*Method*,*Pb*) prioritizes transitions first by favoring those in specified methods. When two transitions both occur in a specified method, the search explores the one that requires fewer preemptions first. The tester can combine any number of priority functions.

When the priorities for two transitions are equal, the fringe always returns the more recently added transition. This behavior mimics depth-first search and helps keep space requirements reasonable. If an entire portion of the state space has equal priority, then GAMBIT explores it in depth-first order and it consumes little additional space.

3.6 Results

We evaluate GAMBIT on the unit tests in Table 8.1, which contain known bugs. Column 2 contains the test name and the maximum number of transitions in a single execution, which correlates with the size of the state space. Testers at Microsoft developed these tests to test small components of large production software systems.

Figure 3.4 compares the time required to manifest the bugs in Table 8.1 using DPOR with various search strategies. We terminated any search that failed to manifest the bug within one hour, which we indicate with a "+" at the top of the bar in Figure 3.4. We present raw time on a logarithmic y-axis; we did not normalize the results because the baseline often failed to manifest the bug in time.

The light-grey bars represent depth-first DPOR, and the dark grey bars show DPOR with the fewer preemptions first heuristic, BF(*Pb*). Prioritizing executions

Program	Unit test (size)	Description
CCR	`Iterator` (165) `Causality` (2615) `ScatterGather` (12156) `TaskCoverage` (203) `GatherPost1` (142) `GatherPost2` (164)	Concurrent programming model based on message-passing with orchestration primitives that coordinate data and work.
Region Ownership	`RegOwn` (277)	Ownership-based separation of the heap for parallel programs.
SYN	`Barrier1` (124) `Barrier2` (102) `ManualResetEvent` (93) `Semaphore` (120)	Low-level synchronization primitives.
CDS	`ConcBag1` (944) `ConcBag2` (336) `BlockingColl` (936)	Parallel data structures for .NET 4.0.
TPL	`NQueens` (1079)	Imperative task-parallelism for .NET 4.0
PLINQ	`NQueens` (972) `ParallelDo` (2721)	Declarative data-parallelism in .NET 4.0.

Table 3.1: Programs and corresponding unit tests with the maximum number of transitions in a single execution in parenthesis.

with fewer preemptions manifests most bugs an order of magnitude more quickly by preventing depth-first search from lingering in uninteresting parts of the state space. Note that we do not combine DPOR with sleep sets in these results.

The black bars in Figure 3.4 randomly prioritize executions. We seed the random number generator with the current time and run each random test ten times, reporting the average of those ten tests in Figure 3.4. One data point is missing – `Barrier1` required anywhere from one second to over an hour depending on the random seed, so we could not compute an average. `Barrier1` demonstrates an important shortcoming of random search – inconsistency. Without progress guarantees, random search may never manifest a bug even though states that manifest it are reachable. Gambit harnesses the benefit of random search without sacrificing coverage by incorporating it into a search with progress guarantees.

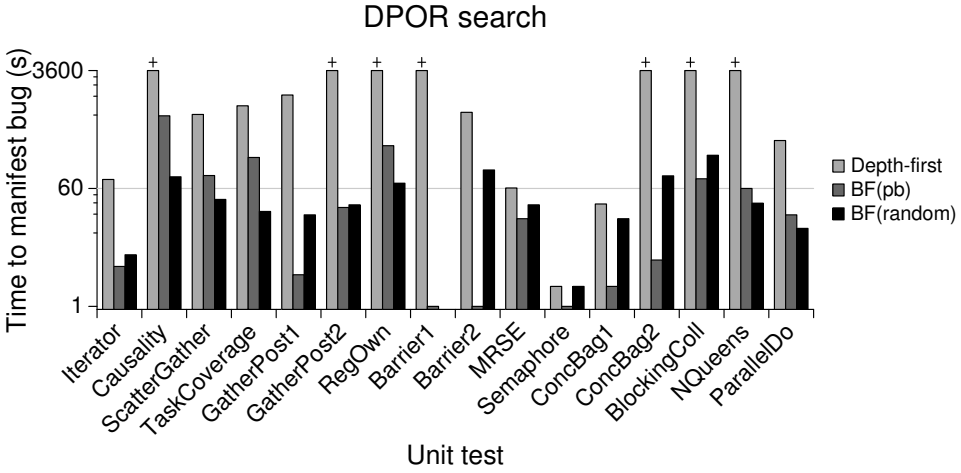


Figure 3.4: Time required to find bugs using depth-first DPOR and DPOR with the preemption bound heuristic. A "+" indicates that the depth-first search required longer than one hour.

Figure 3.5 provides similar results for preemption-bounded search with bound two. We use sleep sets and DPOR as best-first search heuristics. Some bugs, such as those in `GatherPost1`, `Barrier1`, and `NQueens`, manifest more quickly with bounded search than with DPOR, while others such as `Semaphore` and `ConcBag` manifest more quickly with DPOR than with bounded search. Neither approach is clearly more effective. Soundly combining bounded search with partial-order reduction, however, shows great promise.

Execution trees require, on average, about 10x less space than the uncompressed representation. We assume that the uncompressed trees also clean themselves up as efficiently as possible when nodes become unnecessary. Prior work includes results for other priority functions [Coons et al., 2010]. The results in Figures 3.4 and 3.5 motivate soundly combining bounded search with DPOR to reduce the state space.

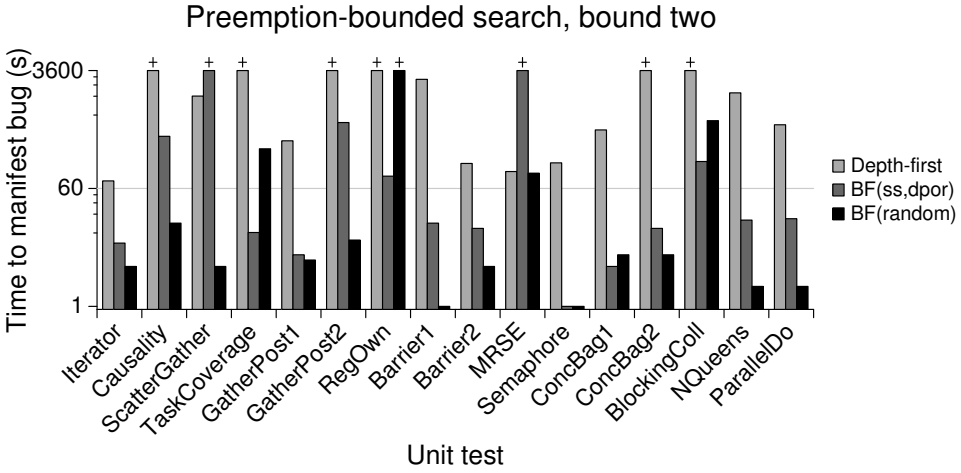


Figure 3.5: Time required to find bugs using preemption-bounded search with bound two using depth-first search, and using a combined DPOR and sleep sets heuristic. A "+" indicates that the depth-first search required longer than one hour.

3.7 Discussion

This chapter compared DPOR with bounded search and showed that bounded search does not scale well as the bound increases. Bounded search could benefit from partial-order reduction, yet combining DPOR with bounded search is difficult, as shown in Section 3.1. To determine whether combining DPOR with bounded search will help testers find bugs more quickly, we combine intuitions from these approaches using best-first search.

We find that algorithms that combine intuitions from DPOR with best-first search are very effective at finding bugs quickly while preserving coverage guarantees. As the bound or the size of the state space increase, however, preserving coverage guarantees requires too much time and space. To find bugs quickly and provide coverage guarantees in a reasonable time period, bounded search requires partial-order reduction. In the next chapter, we show that by conservatively adding backtrack points to compensate for dependences introduced by the bound, we can combine bounded search with partial-order reduction and guarantee bounded coverage.

Chapter 4

Bound Persistent Sets

In this chapter we combine bounded search with partial-order reduction by reasoning about dependences that the bound introduces between otherwise independent transitions. To handle these dependences we introduce *bound persistent sets*. Bound persistent sets reduce the size of the bounded state space while guaranteeing bounded coverage. In the previous chapter we *prioritized* the bounded state space using intuitions from partial-order reduction, but we did not reduce the state space. In this chapter, we reduce the state space while preserving bounded coverage.

First, we establish sufficient conditions to provide two safety guarantees for acyclic state spaces among executions that do not exceed the bound: absence of deadlocks, and absence of local assertion failures. We show that the bound introduces dependences between otherwise independent transitions, and these dependences severely limit partial-order reduction with persistent sets.

These dependences suggest two properties of bound functions that allow them to combine with partial-order reduction without sacrificing coverage. Bound functions from prior work do not have these properties, and thus interact poorly with partial-order reduction. We use these properties to identify cases where bounded search must sacrifice partial-order reduction to maintain bounded coverage.

As we show in Section 3.1, dependences introduced by the bound are different from dependences on shared data. While bound dependences can disable threads, they cannot otherwise change the program's state. We define *bound persistent sets* for each bound function described in Section 2.7 to exploit this distinction and reduce the state space while preserving bounded coverage. We define bound persistent sets for each of the following bound functions:

1. **Depth bound**: bound the number of transitions
2. **Context bound**: bound the number of context switches
3. **Preemption bound**: bound the number of preemptive context switches
4. **Delta bound**: bound the number deltas from a deterministic execution
5. **Fair bound**: bound the difference in yield operations performed by the explored thread and each other enabled thread

Each bound function interacts differently with partial-order reduction. The depth bound has the property that any two deadlock states have the same depth, but it increments the bound at each step and thus leaves transitions unreachable within the bound. The context bound permits more partial-order reduction than the depth bound does. Transitions increment the bound less often in context-bounded search and thus introduce fewer dependences. The preemption, delta, and fair bounds permit partial-order reduction to varying degrees. We prove that selective search with bound persistent sets reaches all deadlock and local states in an acyclic state space for each bound function.

4.1 Sufficient Sets

A set of transitions is sufficient in a state s if any relevant state reachable via an enabled transition from s is also reachable from s via at least one of the transitions

Algorithm 5 Generic selective search [Godefroid, 1996].

```
1: Initially, Explore(ε) from s_0
2: procedure Explore(S) begin
3:    T = Sufficient_set(final(S))
4:    for all t ∈ T do
5:       Explore(S.t)
6:    end for
7: end
```

Algorithm 6 Bounded selective search for bound function Bv with bound c.

```
1: Initially, Explore(ε) from s_0
2: procedure Explore(S) begin
3:    T = Sufficient_set(final(S))
4:    for all t ∈ T do
5:       if Bv(S.t) ≤ c then
6:          Explore(S.t)
7:       end if
8:    end for
9: end
```

in the sufficient set. The search can thus explore only the transitions in the sufficient set from s because all relevant states still remain reachable. The set containing all enabled threads is trivially sufficient in s, but smaller sufficient sets often reduce the state space further and enable more efficient search.

Selective search [Godefroid, 1996] explores only a sufficient set of transitions from each state and thus explores a reduced state space that preserves coverage guarantees. Algorithm 5 performs selective search. Line 3 returns a nonempty sufficient set of transitions in each state *final*(S), and Lines 4-6 recursively explore only the transitions in that sufficient set.

Algorithm 6 performs bounded selective search. Like Algorithm 5, the **Sufficient_set** procedure returns a nonempty sufficient set of enabled transitions in each state *final*(S). Algorithm 6 explores each transition, however, only if that transition does not exceed the bound from *final*(S). Requirements for this sufficient set vary with the bound evaluation function and with the desired safety guarantee.

Note that enabled transitions that exceed the bound in Algorithm 6 are *not* disabled. The bound blocks threads and affects what portions of the state space are reachable, but it cannot change the behavior of the tested program. The bound is not part of the program under test; it is an artificial construct imposed by the thread scheduler. Thus, we do not consider a thread blocked by the bound to be disabled in the same sense that a thread waiting to acquire a lock is disabled.

A deadlock that constitutes a program error or terminal state is different from a deadlock that is artificially imposed by the bound. The former concept is useful to the tester whereas the latter concept most likely is not. Thus, a thread that exceeds the bound is not disabled, and we account for dependences it introduces differently than we account for dependences in the program under test.

The search explores co-enabled dependent transitions in the program under test in both orders to guarantee coverage. The search need not explore dependences introduced by the bound in both orders, however. Provided that the search reaches each reachable state required by the safety guarantee, the search is sufficient. Thus, we conservatively add backtrack points only when the taken transitions leave states unreachable due to the bound. Dynamic search makes this information more accurate and more easily accessible than it would be with static partial-order reduction.

We identify constraints on a sufficient set such that Algorithm 6 guarantees absence of deadlocks and absence of local assertion failures for acyclic state spaces. We use $A_{G(Bv,c)}$ to refer to a generic global state space that may or may not be bounded – unbounded search is equivalent to bounded search with a nonnegative bound and a bound function that always returns zero.

Definition 4.1. Deadlock sufficient sets.
A nonempty set $T \subseteq \mathcal{T}$ of transitions enabled in a state s in $A_{G(Bv,c)}$ is *deadlock sufficient* in s if and only if for all deadlock states d reachable from s via a nonempty sequence ω of transitions in $A_{G(Bv,c)}$, there exists a sequence ω' of transitions from

s in $A_{G(Bv,c)}$ such that $\omega' \in [\omega]$, and $\omega'_1 \in T$.

Let $A_{R(Bv,c)}$ be the reduced state space that Algorithm 6 explores if Line 3 returns a nonempty deadlock sufficient set in each state. We prove that all deadlock states reachable in $A_{G(Bv,c)}$ are also reachable in $A_{R(Bv,c)}$. This theorem and its proof are similar to one that proves that persistent sets provide deadlock state reachability [Godefroid, 1996]. We generalize this concept and apply it to bounded search.

Theorem 2. *Let s be a state in $A_{R(\mathrm{Bv},c)}$, and let d be a deadlock state reachable from s in $A_{G(\mathrm{Bv},c)}$ by a sequence ω of transitions. Then, d is also reachable from s in $A_{R(\mathrm{Bv},c)}$.*

Proof. The proof is by induction on the length of ω.

Case 2.1. Base Case.

For $len(\omega) = 0$ the result is immediate.

Case 2.2. Inductive case.

Assume that the theorem holds for all sequences of transitions of length $n > 0$, and show that it holds for sequences ω of length $n + 1$.

Assume that d is reachable from s by a sequence ω of transitions of length $n + 1$ in $A_{G(Bv,c)}$. Let T be the nonempty deadlock sufficient set selected in s by Algorithm 6, i.e., the set of transitions explored from s in $A_{R(Bv,c)}$. By Definition 4.1 of deadlock sufficient sets, there exists a sequence ω' of transitions from s in $A_{G(Bv,c)}$ such that $\omega' \in [\omega]$ and $\omega'_1 \in T$. By Theorem 1, ω' also leads to d. Because $\omega'_1 \in T$, ω'_1 is explored from s and the state $final(S.\omega'_1)$ is reachable in $A_{R(Bv,c)}$. From $final(S.\omega'_1)$, d is reachable via a path of length n in $A_{G(Bv,c)}$. Thus, by the inductive hypothesis, d is also reachable from s in $A_{R(Bv,c)}$.

□

Thus, Algorithm 6 explores all deadlock states reachable in the bounded state space

if it explores a nonempty deadlock sufficient set in each state.

To guarantee that a program will not violate any local assertions with the given inputs, the search must additionally provide local state reachability. Local state reachability places greater constraints on bounded search than deadlock state reachability does. When the search requires only deadlock state reachability it must ensure that any two sequences of transitions that lead to the same deadlock state have the same bounded value. Thus, if two sequences of transitions lead to the same deadlock state, the search may explore either sequence of transitions.

In contrast, to guarantee local state reachability the search must ensure that any two sequences of transitions that lead to the same local state have the same bounded value. Two paths to the same local state may not contain the same transitions or the same number of transitions. Thus, simple bounds like the depth bound hinder local-state reachability. We define *local sufficient sets* to ensure that the search reaches all local states reachable within the bound in an acyclic state space.

Definition 4.2. ***Pref***$([\omega])$ [Godefroid, 1996].
Pref$([\omega])$ returns the set containing all prefixes of all sequences in the Mazurkiewicz trace defined by ω.

Definition 4.3. Local sufficient.
A nonempty set $T \subseteq \mathcal{T}$ of transitions enabled in a state s in $A_{G(Bv,c)}$ is *local sufficient* in s if and only if for all sequences ω of transitions from s in $A_{G(Bv,c)}$, there exists a sequence ω' of transitions from s in $A_{G(Bv,c)}$ such that $\omega \in Pref([\omega'])$ and $\omega'_1 \in T$.

Let $A_{R(Bv,c)}$ be the reduced state space that Algorithm 6 explores if Line 3 returns a nonempty local sufficient set in each state.

Theorem 3. *Let s be a state in $A_{R(\mathrm{Bv},c)}$, and let l be a local state reachable from s in $A_{G(\mathrm{Bv},c)}$ by a sequence ω of transitions. Then, l is also reachable from s in $A_{R(\mathrm{Bv},c)}$.*

Proof. The proof is by induction on the length of the longest sequence ω of transitions from s that leads to l in $A_{G(Bv,c)}$.

Case 3.1. Base Case.

For $\mathit{len}(\omega) = 0$ the result is immediate.

Case 3.2. Inductive case.

Let l be a local state such that the longest sequence of transitions ω from s to l has length $n+1$. Let $l = \mathit{local}(\mathit{final}(S.\omega), u)$ for some thread u. Let T be the nonempty local sufficient set selected in s by Algorithm 6, i.e., the set of transitions explored from s in $A_{R(Bv,c)}$.

By Definition 4.3 of local sufficient sets, there exists a sequence ω' of transitions from s in $A_{G(Bv,c)}$ such that $\omega'_1 \in T$ and $\omega \in \mathit{Pref}([\omega'])$. Thus, by Definition 4.2 of the prefix function, there exists a sequence β of transitions from $\mathit{final}(S.\omega)$ such that $\omega.\beta \in [\omega']$. Assume that none of the transitions in ω are by u. Then, by definition of local states,

$$\mathit{local}(\mathit{final}(S.\omega), u) = \mathit{local}(\mathit{final}(S), u)$$

and the result is immediate.

Assume that a transition in ω is by u. Let $i \in \mathit{dom}(\omega)$ be the maximum value of i such that $\omega_i.\mathit{tid} = u$. Because $\omega.\beta \in [\omega']$, there must exist $j \in \mathit{dom}(\omega')$ such that $\omega'_j = \omega_i$. Let $\omega' = \alpha.t.\gamma$ such that $t = \omega'_j$. Because $\omega.\beta \in [\omega']$,

$$\mathit{local}(\mathit{final}(S.\omega), u) = \mathit{local}(\mathit{final}(S.\alpha.t), u)$$

Thus, ω' leads to l. Because ω'_1 is in T, it is explored from s and the state $\mathit{final}(S.\omega'_1)$ is reachable in $A_{R(Bv,c)}$. Because ω is the longest sequence of transitions that leads

to l in $A_{G(Bv,c)}$, $len(S.\alpha.t) \leq len(\omega)$. Thus, from $final(S.\omega'_1)$, l is reachable via a sequence of transitions of length n. By the inductive hypothesis, l is also reachable from s in $A_{R(Bv,c)}$.

□

Thus, if Algorithm 6 returns a local sufficient set of transitions in each state at Line 3, then Algorithm 6 explores all local states reachable within the bound. Next, we show that any local sufficient set in a state s is also deadlock sufficient in s.

Theorem 4. *If T is a nonempty local sufficient set in a state s in $A_{R(\mathrm{Bv,c})}$, then T is deadlock sufficient in s.*

Proof. Let s be a state in $A_{R(Bv,c)}$ and let d be a deadlock state reachable from s in $A_{G(Bv,c)}$ via a nonempty sequence ω of transitions. By Definition 4.3 of local sufficient sets, there exists a sequence ω' from s in $A_{G(Bv,c)}$ such that $\omega \in Pref([\omega'])$ and $\omega'_1 \in T$. Because d is a deadlock state there can be no transitions enabled after ω, and thus $\omega \in Pref([\omega'])$ implies $\omega \in [\omega']$. Thus, by Definition 2.2 of a trace, $\omega' \in [\omega]$ and T is deadlock sufficient in s.

□

Thus, any local sufficient set in a state s is also deadlock sufficient in s, and any algorithm that provides local state reachability within a bound also provides deadlock state reachability within that bound.

We derive Definition 4.3 of local sufficient sets from Godefroid's definition of a trace automaton [Godefroid, 1990]. A trace automaton provides local state reachability from the initial state, s_0. We first consider systems with a more stringent requirement – local state reachability from every visited state. In Chapter 5, we ease this requirement when we introduce sleep sets. For brevity we do not always specify that the state space is acyclic, but we always assume that it is unless we explicitly state otherwise. Similarly, we always assume that the state space is finite.

Next, we show that a nonempty persistent set in a state s is local sufficient in s. Let A_G be the global state space for a system. We derive this theorem and its proof from the correctness proof for persistent sets [Godefroid, 1996], but we include them here to clarify how they fit into this modified framework.

Theorem 5. *A nonempty persistent set T in a state s is local sufficient in s.*

Proof. Let l be a local state reachable from s in A_G via a nonempty sequence ω of transitions. Assume that $\forall i \in dom(\omega) : \omega_i \notin T$. Let t be any transition in T. By Definition 2.5 of persistent sets, t is independent with ω, $t \leftrightarrow \omega$. Consider the sequence $\omega' = t.\omega$. Because $t \leftrightarrow \omega$, by Definition 2.2 of a trace $\omega.t \in [\omega']$. Thus, $\omega \in \mathit{Pref}([\omega'])$, and T is local sufficient in s.

Assume that ω contains a transition $t \in T$. Let $\omega = \alpha.t.\gamma$ such that $\forall i \in dom(\alpha) : \alpha_i \notin T$. By Definition 2.5 of persistent sets, $t \leftrightarrow \alpha$. Consider the sequence $\omega' = t.\alpha.\gamma$, i.e., ω with t moved to the first position. Because $t \leftrightarrow \alpha$, by Definition 2.2 of a trace $\omega' \in [\omega]$. Thus, $\omega \in \mathit{Pref}([\omega'])$, and T is local sufficient in s.

□

Thus, persistent sets are local sufficient and by Theorem 4, persistent sets are deadlock sufficient. Persistent sets can thus verify the absence of deadlocks and local assertion failures in an acyclic state space. Persistent sets severely limit partial-order reduction in bounded search, however, because the bound introduces dependences between otherwise independent transitions. In the next section, we modify persistent sets to accommodate bounded search while preserving partial-order reduction.

4.2 Bound Sufficient Sets

We introduce *bound sufficient sets* to reduce the bounded state space and guarantee bounded coverage. While persistent sets are effective for unbounded search, they inhibit partial-order reduction in bounded search because the bound introduces

dependences between otherwise independent transitions, as shown in Section 3.1. Bound sufficient sets compensate for these dependences to guarantee bounded coverage while preserving partial-order reduction.

We define sufficient sets for depth, context, preemption, delta, and fair-bounded search, introduced in Section 2.7, and prove that these sets are sufficient. Note that the criteria we propose for each bound function are *sufficient* to provide local or deadlock state reachability within the bound, but these criteria may be optimized to further reduce the state space. Chapter 5 optimizes these sufficient sets to permit additional partial-order reduction and preserve bounded coverage. First, we describe two properties of bound functions that enable partial-order reduction.

4.2.1 Properties of Bound Functions

We identify two properties of bound functions that enable bounded partial-order reduction. Unlike Definition 2.6 of monotonicity, bounded search does not require these properties for correctness. These properties do affect the degree to which partial-order methods may reduce the state space, however. The first property enables deadlock state reachability, and the second property enables local state reachability. We define each property for a generic bound function, Bv.

Definition 4.4. Stable bound functions.
Bound function Bv is *stable* if and only if for all sequences ω and ω' in $A_{G(Bv,c)}$

$$\omega \in [\omega'] \implies Bv(\omega) = Bv(\omega')$$

Intuitively, a stable bound function requires that sequences of transitions that lead to the same global state have the same cost. This property ensures that the bound does not interfere with deadlock state reachability, as shown in Figure 4.1. This property enables partial-order reduction because it preserves the commutativity of

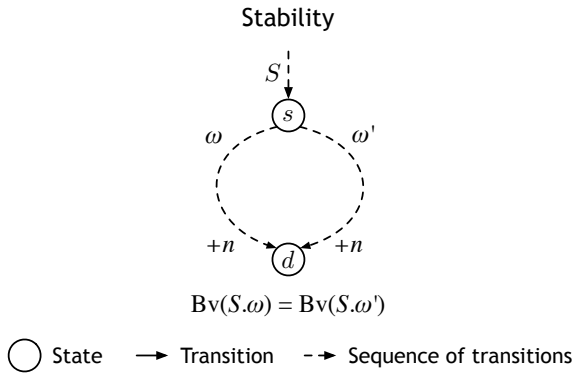


Figure 4.1: Stable bound functions ensure that the bound does not interfere with deadlock state reachability.

independent transitions with respect to the bound. Partial-order reduction leverages this commutativity to reduce the state space.

When the bound function is not stable, two sequences of transitions that lead to the same deadlock state may have different costs. If a portion of the state space is unreachable within the bound via the path that the search explores first, then the search must also explore the other path. Although additional states may be reachable via the other path, many of the states reachable via that path will be redundant, and this redundancy sacrifices partial-order reduction.

The search can prune the state space most effectively if it explores the cheapest sequence of transitions to each state first. Bounded partial-order reduction guides the search toward this cheapest path to limit the overhead of exploring alternative paths. We construct deadlock sufficient sets to guide the search toward the cheapest path to each state first, and to ensure that the search explores alternative paths when necessary. First, we define a second property of bound functions that enables bounded partial-order reduction.

Definition 4.5. Extensible bound functions.

Bound function Bv is *extensible* if and only if for all sequences of transitions S in

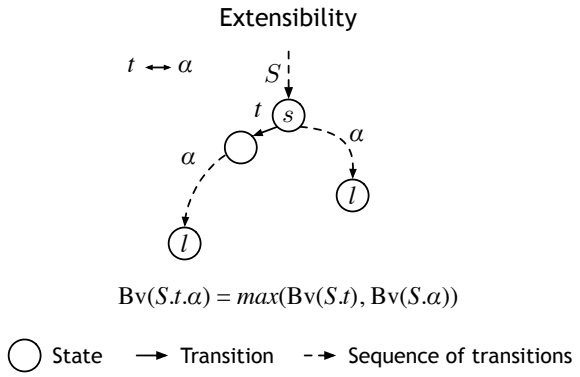


Figure 4.2: Extensible bound functions ensure that the bound does not interfere with local state reachability.

$A_{G(Bv,c)}$, for all transitions t such that $t.tid \in \mathit{enabled}(\mathit{final}(S))$ and for all sequences of transitions α from $\mathit{final}(S)$ such that $t \leftrightarrow \alpha$,

$$Bv(S.t.\alpha) = max(Bv(S.t), Bv(S.\alpha))$$

Extensible bound functions enable local state reachability, as shown in Figure 4.2.

If the bound is not extensible, then exploring independent transitions may make local states that were previously reachable within the bound unreachable. Thus, to ensure local state reachability within the bound, the search must explore otherwise independent transitions. These independent transitions sacrifice partial-order reduction because they lead to many redundant states.

One trivial bound is both stable and extensible – the bound function that always returns zero. This bound function is equivalent to unbounded search and permits full partial-order reduction. In Chapter 7, we introduce other stable, extensible bound functions. Bounds from prior work are not extensible, and many are not stable – they introduce artificial dependences among otherwise independent transitions. We define deadlock sufficient and local sufficient sets for different bound functions to compensate for these bound dependences.

4.2.2 Depth-Bounded Search

Depth-bounded search, introduced in Section 2.7.1, limits the depth – the number of transitions – in each execution. The depth bound is stable but not extensible. Because the depth bound is stable, persistent sets are deadlock sufficient in depth-bounded search without accounting for the bound. We define *depth-bound persistent sets* similarly to persistent sets (Definition 2.5), except that depth-bound persistent sets require independence only from sequences reachable within the bound.

Definition 4.6. Depth-bound persistent sets.
A set $T \subseteq \mathcal{T}$ of transitions enabled in a state $s = \mathit{final}(S)$ is *depth-bound persistent in* s if and only if for all nonempty sequences α of transitions from s in $A_{G(Df,c)}$ such that $\forall i \in \mathit{dom}(\alpha), \alpha_i \notin T$ and for all $t \in T$, $t \leftrightarrow \mathrm{last}(\alpha)$.

Let $A_{R(Df,c)}$ be the reduced state space explored by Algorithm 6 with bound function *Df* and bound c. Assume that Line 3 returns a depth-bound persistent set T in each state. We prove that T is deadlock sufficient.

Theorem 6. *If T is a nonempty depth-bound persistent set in a state s in $A_{R(\mathrm{Df},c)}$, then T is deadlock sufficient in s.*

Proof. Let s be a state in $A_{R(Df,c)}$ and let d be a deadlock state reachable from s in $A_{G(Df,c)}$ via a sequence ω of transitions. Assume that $\forall i \in \mathit{dom}(\omega) : \omega_i \notin T$. Then, by Definition 4.6 of depth-bound persistent sets, $\forall t \in T : t \leftrightarrow \omega$. Thus, by Definition 2.3 of valid dependence relations, all transitions in T remain enabled in d. Thus, d is not a deadlock state and we have a contradiction.

Assume that at least one transition in ω is in T. Let $\omega = \alpha.t.\gamma$ such that $\forall i \in \mathit{dom}(\alpha) : \alpha_i \notin T$ and $t \in T$. Consider the sequence $\omega' = t.\alpha.\gamma$, i.e., ω with t moved to the first position. By Definition 4.6 of depth-bound persistent sets, $t \leftrightarrow \alpha$. Thus, by Definition 2.2 of a trace, $\omega' \in [\omega]$, and by Theorem 1, ω' also leads to d. By Definition 2.7 of the depth bound, $\mathit{Df}(S.t.\alpha.\gamma) = \mathit{Df}(S.\alpha.t.\gamma) \leq c$, and T is thus

deadlock sufficient in s.

□

Thus, if T is a nonempty depth-bound persistent set in a state s in $A_{R(Df,c)}$, then T is deadlock sufficient in s. In contrast, T may *not* be local sufficient in s because the depth bound is not extensible. All transitions increment the bound and may therefore leave local states unreachable within the bound. Thus, the search sacrifices *all* partial-order reduction to guarantee local state reachability. We next define sufficient sets for context-bounded search.

4.2.3 Context-Bounded Search

Context-bounded search, introduced in Section 2.7.2, limits the number of context switches in each execution [Musuvathi and Qadeer, 2007a]. The context bound is neither stable nor extensible. We show how to compensate for dependences that the context bound introduces and provide both deadlock-state reachability and local-state reachability. Deadlock-state reachability permits more partial-order reduction than local-state reachability permits for context-bounded search. Unlike depth-bounded search, however, context-bounded search provides local-state reachability without sacrificing *all* partial-order reduction.

By guiding the search to each state via the cheapest path, context-bounded search can prune portions of the state space that leave no additional states reachable. We show that exploring the executing thread first in each state, if it is enabled in that state, is effective for reaching new states via the cheapest path first. We exploit this property of the context bound to provide limited partial-order reduction for context-bounded search.

To provide deadlock state reachability for context-bounded search, we introduce *deadlock sufficient context-bound persistent sets*. Note that deadlock sufficient context-bound persistent sets do not provide local state reachability. The context

bound is not stable because different paths that lead to the same global state may contain different numbers of context switches. To alleviate this problem, the search can sometimes guarantee that it has reached a state via the cheapest path to that state. To reach states via the cheapest path first, the search explores the transition that does not require a context switch first in each state, if possible.

Intuitively, the following proofs treat sequences of transitions by the same thread that do not contain any context switches as a unit. The search can reduce the state space when *all* of those transitions are independent with reachable transitions not in the deadlock sufficient context-bound persistent set. Otherwise, the search must insert conservative backtrack points because a cheaper path may exist, and new deadlock states may be reachable within the bound via that path.

Definition 4.7. ***ext***(s, t)**.**
Given a state $s = \mathit{final}(S)$ and a transition $t \in \mathit{enabled}(s)$, $\mathit{ext}(s, t)$ returns the unique sequence of transitions β from s such that

1. $\forall i \in \mathit{dom}(\beta) : \beta_i.\mathit{tid} = t.\mathit{tid}$
2. $t.\mathit{tid} \notin \mathit{enabled}(\mathit{final}(S.\beta))$

Intuitively, $\mathit{ext}(s, t)$ returns the sequence of transitions that results if $t.\mathit{tid}$ executes from s until it blocks.

Definition 4.8. Deadlock sufficient context-bound persistent sets.
A set $T \subseteq \mathcal{T}$ of transitions enabled in a state $s = \mathit{final}(S)$ is *deadlock sufficient context-bound persistent* in s if and only if for all nonempty sequences α of transitions from s in $A_{G(Cs,c)}$ such that $\forall i \in \mathit{dom}(\alpha), \alpha_i \notin T$ and for all $t \in T$,

1. $\mathit{Cs}(S.t) \leq \mathit{Cs}(S.\alpha_1)$
2. if $\mathit{Cs}(S.t) < \mathit{Cs}(S.\alpha_1)$, then $t \leftrightarrow \mathrm{last}(\alpha)$
3. if $\mathit{Cs}(S.t) = \mathit{Cs}(S.\alpha_1)$, then $\mathit{ext}(s, t) \leftrightarrow \mathrm{last}(\alpha)$

Note that Requirement 2 would also be correct if it read "$Cs(S.t) \leq Cs(S.\alpha_1)$", because Requirement 3 implies the equal case. We chose this definition because the two definitions are equivalent, but this one simplifies the proofs.

Let $A_{R(Cs,c)}$ be the reduced state space explored by Algorithm 6 with bound function Cs and bound c. Assume that in each state, Line 3 of Algorithm 6 returns a nonempty deadlock sufficient context-bound persistent set, T. We prove a lemma to manage the bound, then show that T is deadlock sufficient.

Lemma 7. *Let $s = \text{final}(S)$ be a state in $A_{R(\text{Cs},c)}$ and let $\omega = \alpha.\beta.\gamma$ be a sequence of transitions from s in $A_{G(\text{Cs},c)}$ such that α and β are nonempty and*

1. $\text{Cs}(S.\beta_1) \leq \text{Cs}(S.\alpha_1)$
2. $\beta \leftrightarrow \alpha$
3. $\forall i \in \text{dom}(\beta) : \beta_i.tid = \beta_1.tid$
4. *if* $\text{Cs}(S.\beta_1) = \text{Cs}(S.\alpha_1)$ *and* γ *is nonempty, then* $\gamma_1.tid \neq \beta_1.tid$

Then, $\beta.\alpha.\gamma$ is a sequence of transitions from s in $A_{G(\text{Cs},c)}$.

Proof. By Assumption 2, $\beta.\alpha.\gamma$ is a sequence of transitions from s in A_G. For each context switch in $S.\beta.\alpha.\gamma$, from left to right, show that there exists a unique context switch in $S.\alpha.\beta.\gamma$.

Assume that β_1 requires a context switch from *final*(S). By Assumption 1, α_1 also requires a context switch from *final*(S). By Assumption 3, no transition in β after β_1 requires a context switch. Assume that α_1 requires a context switch from *final*$(S.\beta)$. By Assumption 2, $\text{last}(\alpha).tid \neq \beta_1.tid$, and thus β_1 requires a context switch from *final*$(S.\alpha)$. Assume that a transition α_i, $2 \leq i \leq len(\alpha)$, requires a context switch in $S.\beta.\alpha.\gamma$. By Definition 2.8 of the context bound, α_i also requires a context switch in $S.\alpha.\beta.\gamma$.

Assume that γ is nonempty and that γ_1 requires a context switch from $\mathit{final}(S.\beta.\alpha)$. If $\mathit{Cs}(S.\beta_1) < \mathit{Cs}(S.\alpha_1)$, then α_1 requires a context switch from $\mathit{final}(S)$ and β_1 does not, so this context switch is unique. Otherwise, by Assumption 1, $\mathit{Cs}(S.\beta_1) = \mathit{Cs}(S.\alpha_1)$, and by Assumption 4, $\gamma_1.\mathit{tid} \neq \text{last}(\beta).\mathit{tid}$. Thus, γ_1 requires a context switch from $\mathit{final}(S.\alpha.\beta)$. Assume that a transition γ_i, $2 \leq i \leq \mathit{len}(\gamma)$, requires a context switch in $S.\beta.\alpha.\gamma$. By Definition 2.8 of the context bound, γ_i also requires a context switch in $S.\alpha.\beta.\gamma$. Thus, for each context switch in $S.\beta.\alpha.\gamma$ there exists a unique context switch in $S.\alpha.\beta.\gamma$ and

$$\mathit{Cs}(S.\beta.\alpha.\gamma) \leq \mathit{Cs}(S.\alpha.\beta.\gamma) \leq c$$

Thus, $\beta.\alpha.\gamma$ is a sequence of transitions from s in $A_{G(\mathit{Cs},c)}$.

□

Theorem 8. *If T is a nonempty deadlock sufficient context-bound persistent set in a state s in $A_{R(\mathrm{Cs},c)}$, then T is deadlock sufficient in s.*

Proof. Let s be a state in $A_{R(\mathit{Cs},c)}$ and let d be a deadlock state reachable from s in $A_{G(\mathit{Cs},c)}$ via a nonempty sequence ω of transitions. Assume that $\forall i \in \mathit{dom}(\omega) : \omega_i \notin T$. Then, by Requirements 2 and 3 of Definition 4.8 of deadlock sufficient context-bound persistent sets, $\forall t \in T : t \leftrightarrow \omega$. Thus, by Definition 2.3 of valid dependence relations, all transitions in T remain enabled in d. Thus, d is not a deadlock state and we have a contradiction. Therefore, at least one transition in ω is in T.

Let $\omega = \alpha.\beta.\gamma$ such that

1. $\forall i \in \mathit{dom}(\alpha) : \alpha_i \notin T$
2. $\beta_1 \in T$
3. $\forall i \in \mathit{dom}(\beta) : \beta_i.\mathit{tid} = \beta_1.\mathit{tid}$
4. if $\mathit{Cs}(S.\beta_1) < \mathit{Cs}(S.\alpha_1)$, then $\mathit{len}(\beta) = 1$

5. if $Cs(S.\beta_1) = Cs(S.\alpha_1)$ and γ is nonempty, then $\gamma_1.tid \neq t.tid$

Assume that α is empty. Then, T is deadlock sufficient in s because $\omega_1 \in T$ and d is reachable via ω. Assume that α is nonempty. Consider the sequence $\omega' = \beta.\alpha.\gamma$, i.e., ω with β moved to the beginning. By Definition 4.8 of deadlock sufficient context-bound persistent sets, $Cs(S.\beta_1) \leq Cs(S.\alpha_1)$ and $\beta \leftrightarrow \alpha$. Thus, by Definition 2.2 of a trace, $\omega' \in [\omega]$ and by Theorem 1, ω' also leads to d. By Lemma 7, ω' is a sequence of transitions from s in $A_{G(Cs,c)}$, and T is deadlock sufficient in s.

□

By Theorems 2 and 8, Algorithm 6 reaches all deadlock states reachable in $A_{G(Cs,c)}$ if it explores a nonempty deadlock sufficient context-bound persistent set in each state. The search may not reach all local states in $A_{G(Cs,c)}$, however. The context switch bound is not extensible, and independent transitions may therefore leave local states unreachable within the bound. In particular, if the search explores a transition that requires a context switch from a state s, then the search must explore *all* enabled threads from s to preserve local state reachability.

We introduce *context-bound persistent sets* to provide local state reachability for context-bounded search. Context-bound persistent sets prune the state space only in states where the executing thread is a valid single-transition persistent set. In all other states, the search conservatively explores all enabled threads to preserve local state reachability. With this requirement Algorithm 6 provides local state reachability, but reduces the state space less aggressively than it does with deadlock sufficient context-bound persistent sets.

Definition 4.9. Context-bound persistent sets.
A set $T \subseteq \mathcal{T}$ of transitions enabled in a state $s = final(S)$ is *context-bound persistent* in s if and only if for all nonempty sequences α of transitions from s in $A_{G(Cs,c)}$ such that $\forall i \in dom(\alpha), \alpha_i \notin T$ and for all $t \in T$,

1. $Cs(S.t) < Cs(S.\alpha_1)$

2. $t \leftrightarrow \text{last}(\alpha)$

Let $A_{R(Cs,c)}$ be the reduced state space that Algorithm 6 explores if Line 3 returns a context-bound persistent set T in each state. We prove a lemma to manage the bound, then prove that T is local sufficient.

Lemma 9. *Let α be a nonempty sequence of transitions from $s = \text{final}(S)$ in $A_{G(\text{Cs},c)}$ and let t be a transition enabled in s such that*

1. $\text{Cs}(S.t) < \text{Cs}(S.\alpha_1)$

2. $t \leftrightarrow \alpha$

Then, $t.\alpha$ is a sequence of transitions from s in $A_{G(\text{Cs},c)}$.

Proof. By Assumption 2, $t.\alpha$ is a sequence of transitions from s in A_G. For each context switch in $S.t.\alpha$, from left to right, show that there exists a unique context switch in $S.\alpha$.

By Assumption 1, t does not require a context switch from *final*(S). Assume that α_1 requires a context switch from *final*$(S.t)$. By Assumption 1, α_1 requires a context switch from *final*(S). Assume that a transition α_i, $2 \leq i \leq len(\alpha)$, requires a context switch in $S.t.\alpha$. By Definition 2.8 of the context switch bound, α_i also requires a context switch in $S.\alpha$. Thus, for each context switch in $S.t.\alpha$ there exists a unique context switch in $S.\alpha$ and

$$Cs(S.t.\alpha) \leq Cs(S.\alpha) \leq c$$

Thus, $t.\alpha$ is a sequence of transitions from s in $A_{G(Cs,c)}$.

□

Theorem 10. *If T is a nonempty context-bound persistent set in a state s in $A_{R(\mathrm{Cs},c)}$, then T is local sufficient in s.*

Proof. Let s be a state in $A_{R(Cs,c)}$ and let l be a local state reachable from s in $A_{G(Cs,c)}$ via a sequence ω of transitions. Assume that $\forall i \in dom(\omega) : \omega_i \notin T$. Let t be any transition in T. Consider the sequence $\omega' = t.\omega$. By Definition 4.9 of context-bound persistent sets, $Cs(S.t) < Cs(S.\omega_1)$ and $t \leftrightarrow \omega$. Thus, by Lemma 9, ω' is a sequence of transitions from s in $A_{G(Cs,c)}$, and by Definition 2.2 of a trace, $\omega.t \in [\omega']$. By Definition 4.2 of the prefix function, $\omega \in Pref([\omega'])$. Thus, T is local sufficient in s.

Assume that a transition in ω is in T. Let $\omega = \alpha.t.\gamma$ such that $\forall i \in dom(\alpha) : \alpha_i \notin T$ and $t \in T$. Consider the sequence $\omega' = t.\alpha.\gamma$, i.e., ω with t moved to the first position. By Definition 4.9 of context-bound persistent sets, $Cs(S.t) < Cs(S.\alpha_1)$ and $t \leftrightarrow \alpha$. Thus, by Lemma 7, ω' is a sequence of transitions from s in $A_{G(Cs,c)}$, and by Definition 2.2 of a trace, $\omega' \in [\omega]$. By Definition 4.2 of the prefix function, $\omega \in Pref([\omega'])$. Thus, T is local sufficient in s.

□

By Theorems 10 and 3, if Algorithm 6 explores a nonempty context-bound persistent set in each state, then it explores all local states reachable in $A_{G(Cs,c)}$. By Theorem 4, Algorithm 6 also explores all deadlock states reachable in $A_{G(Cs,c)}$. To provide local state reachability, however, the search sacrifices partial-order reduction. In the next section, we show that preemption-bounded search can provide local and deadlock state reachability within the bound while reducing the state space more aggressively than context-bounded search can.

4.2.4 Preemption-Bounded Search

Preemption-bounded search, introduced in Section 2.7.3, limits the number of *preemptive* context switches in each execution [Musuvathi and Qadeer, 2008]. The

preemption bound is neither stable nor extensible. Like context-bounded search, exploring the cheapest transition first in each state is a good heuristic for reaching new states as cheaply as possible. If the set containing only the executing thread is persistent in a state s, then the search may explore only the executing thread from s. The executing thread can reach its subsequent states at least as cheaply as any other thread enabled in s.

If the executing thread is blocked in a state s or if the search incurs a preemption from s, then the search must be conservative in s. The preemption bound is neither stable nor extensible. If the executing thread executes until it blocks, however, then the bound does not increase and *any* transition may execute without incurring a preemption. Thus, we treat sequences of transitions by the same thread, until that thread blocks, as a unit so that the bound behaves as though it were extensible. We introduce *preemption-bound persistent sets* to exploit this property of the preemption bound and permit limited partial-order reduction with local state reachability within the bound.

The preemption bound is unstable in one additional way – a transition's cost varies with the enabledness of the thread that performed the previous transition. This property introduces dependences between otherwise independent transitions. To compensate for these dependences, transitions in the preemption-bound persistent set must be independent with the *next* transition by each thread that is not persistent. This requirement ensures that transitions in the preemption-bound persistent set cannot affect the cost of sequences of transitions that are not persistent, leaving their local states unreachable.

Definition 4.10. Preemption-bound persistent sets.
A set $T \subseteq \mathcal{T}$ of transitions enabled in a state $s = \mathit{final}(S)$ is *preemption-bound persistent* in s if and only if for all nonempty sequences α of transitions from s in $A_{G(Pb,c)}$ such that $\forall i \in \mathit{dom}(\alpha), \alpha_i \notin T$ and for all $t \in T$,

1. $Pb(S.t) \leq Pb(S.\alpha_1)$

2. if $Pb(S.t) < Pb(S.\alpha_1)$, then $t \leftrightarrow \text{last}(\alpha)$ and $t \leftrightarrow \mathit{next}(\mathit{final}(S.\alpha), \text{last}(\alpha).\mathit{tid})$

3. if $Pb(S.t) = Pb(S.\alpha_1)$, then $\mathit{ext}(s,t) \leftrightarrow \text{last}(\alpha)$ and
   $\mathit{ext}(s,t) \leftrightarrow \mathit{next}(\mathit{final}(S.\alpha), \text{last}(\alpha).\mathit{tid})$

Let $A_{R(Pb,c)}$ be the reduced state space that Algorithm 6 explores with bound function Pb and bound c. Assume that in each state, Algorithm 6 returns a preemption-bound persistent set. We prove two lemmas to manage the bound, then show that a nonempty preemption-bound persistent set is local sufficient.

Lemma 11. *Let α and β be nonempty sequences of transitions from $s = \text{final}(S)$ in $A_{G(\text{Pb},c)}$ such that*

1. $\beta \leftrightarrow \alpha$

2. $\text{Pb}(S.\beta_1) \leq \text{Pb}(S.\alpha_1)$

3. $\forall i \in \text{dom}(\beta) : \beta_i.\mathit{tid} = \beta_1.\mathit{tid}$

4. $\beta \leftrightarrow \text{next}(\text{final}(S.\alpha_1 \ldots \alpha_i), \alpha_i.\mathit{tid}), 1 \leq i \leq \text{len}(\alpha) - 1$

5. *if* $\text{Pb}(S.\beta_1) = \text{Pb}(S.\alpha_1)$, *then* $\beta_1.\mathit{tid} \notin \text{enabled}(\text{final}(S.\beta))$

Then, $\beta.\alpha$ is a sequence of transitions from s in $A_{G(\text{Pb},c)}$.

Proof. By Assumption 1, $\beta.\alpha$ is a sequence of transitions from s in A_G. For each preemption in $S.\beta.\alpha$, from left to right, show that there exists a unique preemption in $S.\alpha$. Assume that β_1 requires a preemption from $\mathit{final}(S)$. By Assumption 2, α_1 also requires a preemption from $\mathit{final}(S)$. By Assumption 3, no transition in β after β_1 requires a preemption.

Assume that α_1 requires a preemption from $\mathit{final}(S.\beta)$. Then,

$$\beta_1.\mathit{tid} \in \mathit{enabled}(\mathit{final}(S.\beta))$$

and thus by Assumptions 2 and 5, $Pb(S.\beta_1) < Pb(S.\alpha_1)$. Thus, α_1 requires a preemption from *final*(S) and β_1 does not, so this preemption is unique. Assume that a transition α_i, $2 \leq i \leq len(\alpha)$, requires a preemption in $S.\beta.\alpha$. By Assumption 4, α_i also requires a preemption in $S.\alpha$. Thus, for each preemption in $S.\beta.\alpha$ there exists a unique preemption in $S.\alpha$ and

$$Pb(S.\beta.\alpha) \leq Pb(S.\alpha) \leq c$$

Thus, $\beta.\alpha$ is a sequence of transitions from s in $A_{G(Pb,c)}$.

□

Lemma 12. *Let T be a nonempty preemption-bound persistent set in a state $s =$ final(S) in $A_{R(\mathrm{Pb},c)}$ and let $\alpha.\beta.\gamma$ be a sequence of transitions from s in $A_{G(\mathrm{Pb},c)}$ such that α and β are nonempty and*

1. $\forall i \in \mathrm{dom}(\alpha) : \alpha_i \notin T$
2. $\beta_1 \in T$
3. $\forall i \in \mathrm{dom}(\beta) : \beta_i.tid = \beta_1.tid$
4. *if* $\mathrm{Pb}(S.\beta_1) < \mathrm{Pb}(S.\alpha_1)$ *then* $\mathrm{len}(\beta) = 1$
5. *if* $\mathrm{Pb}(S.\beta_1) = \mathrm{Pb}(S.\alpha_1)$ *and γ is empty, then* $\beta_1.tid \notin \mathrm{enabled}(\mathrm{final}(S.\beta))$
6. *if* $\mathrm{Pb}(S.\beta_1) = \mathrm{Pb}(S.\alpha_1)$ *and γ is nonempty, then* $\gamma_1.tid \neq \beta_1.tid$

Then, $\beta.\alpha.\gamma$ is a sequence of transitions from s in $A_{G(\mathrm{Pb},c)}$.

Proof. By Assumptions 1-4 and by Requirements 2 and 3 of Definition 4.10 of preemption-bound persistent sets, $\beta \leftrightarrow \alpha$ and

$$\forall i \in dom(\alpha) : \beta \leftrightarrow next(final(S.\alpha_1 \ldots \alpha_i), \alpha_i.tid) \tag{4.1}$$

Thus, $\beta.\alpha.\gamma$ is a sequence of transitions from s in A_G. For each preemption in $S.\beta.\alpha.\gamma$, from left to right, show that there exists a unique preemption in $S.\alpha.\beta.\gamma$. Assume that β_1 requires a preemption from *final*(S). Then, by Requirement 1 of Definition 4.10 of preemption-bound persistent sets, α_1 also requires a preemption from *final*(S). By Assumption 3, no transition in β after β_1 requires a preemption.

Assume that α_1 requires a preemption from *final*$(S.\beta)$. If $Pb(S.\beta_1) < Pb(S.\alpha_1)$, then α_1 requires a preemption from *final*(S) and β_1 does not, so this preemption is unique. Otherwise, by Requirement 1 of Definition 4.10 of preemption-bound persistent sets, $Pb(S.\beta_1) = Pb(S.\alpha_1)$. Because α_1 requires a preemption from *final*$(S.\beta)$,

$$\beta_1.tid \in enabled(final(S.\beta)) \tag{4.2}$$

By Assumption 5, γ is nonempty, and by Assumption 6 $\gamma_1.tid \neq \beta_1.tid$. By Equation 4.2 and Requirement 3 of Definition 4.10 of preemption-bound persistent sets,

$$\beta_1.tid \in enabled(final(S.\alpha.\beta))$$

Thus, γ_1 requires a preemption from *final*$(S.\alpha.\beta)$. Assume that a transition α_i, $2 \leq i \leq len(\alpha)$, requires a preemption in $S.\beta.\alpha.\gamma$. By Equation 4.1, α_i also requires a preemption in $S.\alpha.\beta.\gamma$.

Assume that γ_1 requires a preemption from *final*$(S.\beta.\alpha)$. Then,

$$\text{last}(\alpha).tid \in enabled(final(S.\beta.\alpha))$$

By Equation 4.1,

$$\text{last}(\alpha).tid \in enabled(final(S.\alpha))$$

Because $\beta \leftrightarrow \alpha$, $\beta_1.tid \neq \text{last}(\alpha).tid$. Thus, β_1 requires a preemption from *final*$(S.\alpha)$. Assume that a transition γ_i, $2 \leq i \leq len(\gamma)$, requires a preemption in $S.\beta.\alpha.\gamma$. Be-

cause $\beta \leftrightarrow \alpha$, $\mathit{final}(S.\alpha.\beta.\gamma_1) = \mathit{final}(S.\beta.\alpha.\gamma_1)$. Thus, by Definition 2.9 of the preemption bound, γ_i also requires a preemption in $S.\alpha.\beta.\gamma$. Thus, for each preemption in $S.\beta.\alpha.\gamma$ there exists a unique preemption in $S.\alpha.\beta.\gamma$ and

$$Pb(S.\beta.\alpha.\gamma) \leq Pb(S.\alpha.\beta.\gamma) \leq c$$

Thus, $\beta.\alpha.\gamma$ is a sequence of transitions from s in $A_{G(Pb,c)}$.

□

Theorem 13. *If T is a nonempty preemption-bound persistent set in a state s in $A_{R(\mathrm{Pb},c)}$, then T is local sufficient in s.*

Proof. Let s be a state in $A_{R(Pb,c)}$ and let l be a local state reachable from s in $A_{G(Pb,c)}$ via a nonempty sequence ω of transitions.

Case 13.1. $\forall i \in \mathbf{dom}(\omega) : \omega_i \notin T$**.**
Let t be any transition in T. By Requirement 1 of Definition 4.10 of preemption-bound persistent sets, $Pb(S.t) \leq Pb(S.\omega_1)$. Let $\beta = t$ if $Pb(S.t) < Pb(S.\omega_1)$, and let $\beta = \mathit{ext}(s,t)$ otherwise. Consider the sequence $\omega' = \beta.\omega$. By Requirements 2 and 3 of Definition 4.10 of preemption-bound persistent sets, $\beta \leftrightarrow \omega$ and $\forall i \in dom(\omega) : \beta \leftrightarrow \mathit{next}(\mathit{final}(S.\omega_1 \ldots \omega_i), \omega_i.tid)$. Thus, by Lemma 11 $\beta.\omega$ is a sequence of transitions from s in $A_{G(Pb,c)}$ and by Definition 2.2 of a trace, $\omega.\beta \in [\omega']$. By Definition 4.2 of the prefix function, $\omega \in \mathit{Pref}([\omega'])$. Thus, T is local sufficient in s.

Case 13.2. $\exists i \in \mathbf{dom}(\omega) : \omega_i \in T$**.**
Let $\omega = \alpha.\beta.\gamma$ such that

1. $\forall i \in dom(\alpha) : \alpha_i \notin T$
2. $\beta_1 \in T$
3. $\forall i \in dom(\beta) : \beta_i.tid = \beta_1.tid$

4. if $Pb(S.\beta_1) < Pb(S.\alpha_1)$ then $len(\beta) = 1$

5. if $Pb(S.\beta_1) = Pb(S.\alpha_1)$ and γ is nonempty, then $\gamma_1.tid \neq \beta_1.tid$

Assume that α is empty. Then, T is local sufficient in s because $\omega_1 \in T$ and l is reachable via ω. Assume that α is nonempty. By Requirement 1 of Definition 4.10 of preemption-bound persistent sets, $Pb(S.\beta_1) \leq Pb(S.\alpha_1)$.

Case 13.2a. γ is nonempty, or γ is empty and $\beta_1.tid \notin$ *enabled*(*final*$(S.\beta)$), or $\boldsymbol{Pb}(S.\beta_1) < \boldsymbol{Pb}(S.\alpha_1)$.
Consider the sequence $\omega' = \beta.\alpha.\gamma$, i.e., ω with β moved to the beginning. By Requirements 2 and 3 of Definition 4.10 of preemption-bound persistent sets, $\beta \leftrightarrow \alpha$ and $\forall i \in dom(\alpha) : \beta \leftrightarrow next(final(S.\alpha_1 \ldots \alpha_i), \alpha_i.tid)$. Thus, by Lemma 12 ω' is a sequence of transitions from s in $A_{G(Pb,c)}$ and by Definition 2.2 of a trace $\omega' \in [\omega]$. By Definition 4.2 of the prefix function $\omega \in Pref([\omega'])$, so T is local sufficient in s.

Case 13.2b. γ is empty, $\beta_1.tid \in$ *enabled*(*final*$(S.\beta)$), and $\boldsymbol{Pb}(S.\beta_1) = \boldsymbol{Pb}(S.\alpha_1)$.
Let $\beta' = ext(s, \beta_1)$. Consider the sequence $\omega' = \beta'.\alpha$. By Requirement 3 of Definition 4.10 of preemption-bound persistent sets, $\beta' \leftrightarrow \alpha$ and $\forall i \in dom(\alpha) : \beta' \leftrightarrow next(final(S.\alpha_1 \ldots \alpha_i), \alpha_i.tid)$. Thus, by Lemma 11 $\beta'.\omega$ is a sequence of transitions from s in $A_{G(Pb,c)}$ and by Definition 2.2 of a trace $\omega.\beta' \in [\omega']$. By Definition 4.2 of the prefix function $\omega \in Pref([\omega'])$, so T is local sufficient in s.

□

By Theorems 13 and 3, if Algorithm 6 explores a nonempty preemption-bound persistent set in each state, then it reaches all local states reachable in $A_{G(Pb,c)}$. By Theorem 4, Algorithm 6 also reaches all deadlock states reachable in $A_{G(Pb,c)}$.

Preemption-bounded search reduces the state space more effectively than context-bounded search does because preemption-bounded search can more frequently prove that it has reached states via the cheapest possible path. In par-

ticular, when all transitions require a context switch in context-bounded search, the search cannot guarantee that any enabled transition is sufficient to reach all local states reachable via any other enabled transition. In preemption-bounded search, however, the search can sometimes guarantee that all local states reachable via one transition are also reachable via another transition, even if those transitions have the same cost. In delta-bounded search, two enabled transitions in a state s *never* have the same cost in s. In the next section we show how this property of delta-bounded search affects partial-order reduction.

4.2.5 Delta-Bounded Search

Delta-bounded search, introduced in Section 2.7.4, limits the number of deltas from an initial execution. The delta bound is neither stable nor extensible. We choose this bound function because the context and preemption bounds sacrifice partial-order reduction when enabled transitions have equivalent costs. When the search explores a transition with equal or greater cost than other transitions, it sacrifices partial-order reduction because that transition may leave local states unreachable within the bound. The delta bound has the property that each enabled transition has a unique cost. We choose the delta bound to test whether this property is useful.

We implement delta-bounded search with a round robin scheduler that stores fixed priorities for each thread at each step. The scheduler modifies the priorities only by rotating them – moving the highest priority thread to the lowest priority thread. If the highest priority thread is disabled, this rotation is free. Otherwise, each rotation increments the bounded value by one. Each transition's cost varies with the number of higher priority enabled threads, so operations that block or enable threads introduce dependences among otherwise independent transitions. The search must compensate for these dependences to maintain local state reachability within the bound.

The delta bound is neither stable nor extensible, but exploring the cheapest transition first from each state is an effective heuristic for reaching subsequent states via the cheapest possible path. Because the cost of each transition is unique in each state, delta-bounded search does not need to compensate for states in which different transitions have the same cost. If the search must explore a transition other than the cheapest transition, however, then it must explore all cheaper transitions because any of them may reach the same local states with lower cost.

Delta-bounded search must additionally account for release operations because, like preemption-bounded search, the enabledness of other threads affects a transition's cost. In particular, release operations may increase the cost of other transitions by enabling higher priority threads. The search may need to explore such transitions prior to the release operation because the release operation may leave states unreachable within the bound. Note that acquire operations may *decrease* the cost of other transitions by disabling higher priority threads, but decreasing the cost of other sequences of transitions does not leave portions of the state space unreachable. To provide local and deadlock state reachability for delta-bounded search, we introduce *delta-bound persistent sets*.

Definition 4.11. Delta-bound persistent sets.
A set $T \subseteq \mathcal{T}$ of transitions enabled in a state $s = \mathit{final}(S)$ is *delta-bound persistent* in s if and only if for all nonempty sequences α of transitions from s in $A_{G(De,c)}$ such that $\forall i \in \mathit{dom}(\alpha) : \alpha_i \notin T$ and for all $t \in T$,

1. $\mathit{De}(S.t) < \mathit{De}(S.\alpha_1)$
2. if t is a release operation, then $\forall u \in \mathit{enabled}(s) : \mathit{next}(s, u) \in T$
3. $t \leftrightarrow \mathrm{last}(\alpha)$

Let $A_{R(De,c)}$ be the reduced state space that Algorithm 6 explores with bound function De and bound c. Assume that in each state, Line 3 of Algorithm 6 returns a

delta-bound persistent set. We prove two lemmas to manage the bound, then show that a nonempty delta-bound persistent set is local sufficient.

Lemma 14. *Let α be a nonempty sequence of transitions from $s = \text{final}(S)$ in $A_{G(\text{De},c)}$ and let t be a transition enabled in s such that*

1. $\text{De}(S.t) < \text{De}(S.\alpha_1)$
2. *t is not a release operation*
3. $t \leftrightarrow \alpha$

Then, $t.\alpha$ is a sequence of transitions from s in $A_{G(\text{De},c)}$.

Proof. Because $t \leftrightarrow \alpha$, $t.\alpha$ is a sequence of transitions from s in A_G. Because t is not a release operation,

$$\forall i \in dom(\alpha) : enabled(final(S.t.\alpha_1 \dots \alpha_i)) \subseteq enabled(final(S.\alpha_1 \dots \alpha_i))$$

Thus, by Definition 2.10 of the delta bound, the transitions in α cost no more in $S.t.\alpha$ than they do in $S.\alpha$. By Assumption 1, t has higher priority than α_1 has in s. Thus, by Definition 2.10 of the delta bound,

$$De(S.t.\alpha) \leq De(S.\alpha) \leq c$$

and $t.\alpha$ is a sequence of transitions from s in $A_{G(De,c)}$.

□

Lemma 15. *Let T be a nonempty delta-bound persistent set in a state $s = \text{final}(S)$ in $A_{R(\text{De},c)}$ and let $\alpha.t.\gamma$ be a sequence of transitions from s in $A_{G(\text{De},c)}$ such that α is nonempty, $\forall i \in \text{dom}(\alpha) : \alpha_i \notin T$, and $t \in T$. Then, $t.\alpha.\gamma$ is a sequence of transitions from s in $A_{G(\text{De},c)}$.*

Proof. By Requirement 3 of Definition 4.11 of delta-bound persistent sets $t \leftrightarrow \alpha$. Thus, $t.\alpha.\gamma$ is a sequence of transitions from s in A_G. By Requirements 1 and 2 of Definition 4.11 of delta-bound persistent sets $De(S.t) < De(S.\alpha_1)$ and t is not a release operation, because if t were a release operation then α_1 would be in T and we would have a contradiction. Thus, by Lemma 14,

$$De(S.t.\alpha) \leq De(S.\alpha)$$

To execute γ_1 from $\mathit{final}(S.t.\alpha)$, the highest priority thread must rotate from $\text{last}(\alpha).tid$ to $\gamma_1.tid$. To execute $t.\gamma_1$ from $\mathit{final}(S.\alpha)$, the highest priority thread must rotate from $\text{last}(\alpha).tid$ to $t.tid$ and then to $\gamma_1.tid$, which requires at least as many rotations. Because t is not a release operation, these rotations cannot cost more in $S.t.\alpha.\gamma$ than they do in $S.\alpha.t.\gamma$. Thus,

$$De(S.t.\alpha.\gamma_1) \leq De(S.\alpha.t.\gamma_1)$$

Assume that a transition γ_i, $2 \leq i \leq \mathit{len}(\gamma)$, requires a priority change in $S.t.\alpha.\gamma$. By Definition 2.10 of the delta bound, γ_i requires a priority change of equal cost in $S.\alpha.t.\gamma$ and thus

$$De(S.t.\alpha.\gamma) \leq De(S.\alpha.t.\gamma) \leq c$$

Thus, $t.\alpha.\gamma$ is a sequence of transitions from s in $A_{G(De,c)}$.

□

Theorem 16. *If T is a nonempty delta-bound persistent set in a state s in $A_{R(\mathrm{De},c)}$, then T is local sufficient in s.*

Proof. Let s be a state in $A_{R(De,c)}$ and let l be a local state reachable from s in $A_{G(De,c)}$ via a nonempty sequence ω of transitions.

Case 16.1. $\forall i \in \boldsymbol{dom}(\omega) : \omega_i \notin T$**.**

Let t be any transition in T. Consider the sequence $\omega' = t.\omega$. By Requirement 1 of Definition 4.11 of delta-bound persistent sets, $De(S.t) < De(S.\omega_1)$. By Requirement 3 of Definition 4.11 of delta-bound persistent sets, $t \leftrightarrow \omega$. By Requirement 2 of Definition 4.11 of delta-bound persistent sets, t is not a release operation because otherwise $\omega_1 \in T$ and we have a contradiction. Thus, by Lemma 14, $t.\omega$ is a sequence of transitions from s in $A_{G(De,c)}$. Because $t \leftrightarrow \omega$, $\omega.t \in [\omega']$ and $\omega \in \mathit{Pref}([\omega'])$. Thus, T is local sufficient in s.

Case 16.2. $\exists i \in \boldsymbol{dom}(\omega) : \omega_i \in T$**.**

Let $\omega = \alpha.t.\gamma$ such that $\forall i \in dom(\alpha) : \alpha_i \notin T$ and $t \in T$. Assume that α is empty. Then, T is local sufficient in s because $\omega_1 \in T$ and l is reachable via ω. Assume that α is nonempty. Consider the sequence $\omega' = t.\alpha.\gamma$, i.e., ω with t moved to the first position. By Requirement 3 of Definition 4.11 of delta-bound persistent sets, $t \leftrightarrow \alpha$. Thus, $\omega' \in [\omega]$ and $\omega \in \mathit{Pref}([\omega'])$. By Lemma 15, $t.\alpha.\gamma$ is a sequence of transitions from s in $A_{G(De,c)}$, and T is local sufficient in s.

□

By Theorems 16 and 3, if Algorithm 6 explores a nonempty delta-bound persistent set in each state then it reaches all local states reachable in $A_{G(De,c)}$. By Theorem 4, Algorithm 6 reaches all deadlock states reachable in $A_{G(De,c)}$, as well.

Delta-bounded search sometimes permits more partial-order reduction than preemption-bounded search permits because in every state the delta bound provides a unique cheapest transition. Cheaper transitions always reach subsequent states with a lower bounded value than more expensive transitions that lead to the same state. If the search must explore higher cost threads, however, then it must explore all cheaper threads as well, so sometimes delta-bounded search achieves less partial-order reduction than preemption-bounded search.

We have defined sufficient sets that achieve local-state reachability for depth,

context, preemption, and delta-bounded search. These bounds do not provide local state reachability for cyclic state spaces, however. To prune cycles in a cyclic state space, we define local sufficient sets for fair-bounded search.

4.2.6 Fair-Bounded Search

Fair-bounded search, introduced in Section 2.7.5, limits the maximum difference between the number of yield operations performed by each thread. In particular, fair-bounded search bounds the difference between the taken thread's yield count and each other enabled thread's yield count in each state. The fair bound is neither stable nor extensible. This bound prunes cycles in cyclic state spaces by pruning executions in which a thread yields the processor repeatedly. We assume that if a thread yields the processor repeatedly, then it is not doing work that will change the program's behavior in a meaningful way. Thus, the state space reachable after it performs additional yield operations is not interesting to the tester.

The fair bound is neither stable nor extensible. Like the preemption and delta bounds, the fair bound is unstable with respect to release operations. A release operation may enable threads with a lower yield count, and thus increase the cost of another enabled transition. Yields and operations that block other threads cannot increase the cost of another transition – they can only decrease it.

To provide local and deadlock state reachability for fair-bounded search, we introduce *fair-bound persistent sets*. Fair-bound persistent sets compensate for the fair bound's instability by conservatively scheduling all threads prior to release operations. We use the term "release operation" below to refer to any transition that may enable another thread, including lock release operations, fork operations, and event set operations.

Definition 4.12. Fair-bound persistent sets.
A set $T \subseteq \mathcal{T}$ of transitions enabled in a state $s = \mathit{final}(S)$ is *fair-bound persistent*

in s if and only if for all sequences α of transitions from s in $A_{G(Fb,c)}$ such that $\forall i \in dom(\alpha) : \alpha_i \notin T$ and for all $t \in T$,

1. $Fb(S.t) \leq c$
2. if t is a release operation, then $\forall u \in enabled(s) : next(s, u) \in T$
3. $t \leftrightarrow \text{last}(\alpha)$

While other bounds require that each transition $t \in T$ cost less than or equal to transitions not in T, the fair bound requires only that t be within the bound. The fair bound (Definition 2.11) tracks the *maximum* value observed, not the cumulative value, as other bounds do. Thus, the search need not explore the cheapest transition from a state s, provided that it explores a transition that is within the bound.

Let $A_{R(Fb,c)}$ be the reduced state space explored by Algorithm 6 with bound function Fb and bound c. Assume that in each state, Algorithm 6 returns a fair-bound persistent set. We prove two lemmas to manage the bound, then show that a nonempty fair-bound persistent set is local sufficient.

Lemma 17. *Let α be a nonempty sequence of transitions from $s = \text{final}(S)$ in $A_{G(\text{Fb},c)}$ and let t be a transition enabled in s such that*

1. $\text{Fb}(S.t) \leq c$
2. *t is not a release operation*
3. $t \leftrightarrow \alpha$

Then, $t.\alpha$ is a sequence of transitions from s in $A_{G(\text{Fb},c)}$.

Proof. Because $t \leftrightarrow \alpha$, $t.\alpha$ is a sequence of transitions from s in A_G. Because t is not a release operation,

$$\forall i \in dom(\alpha) : enabled(final(S.t.\alpha_1 \ldots \alpha_i)) \subseteq enabled(final(S.\alpha_1 \ldots \alpha_i))$$

Thus, by Definition 2.11 of the fair bound, the transitions in α cost no more in $S.t.\alpha$ than they do in $S.\alpha$. By Assumption 1, t is within the bound from s. Thus, by Definition 2.11 of the fair bound,

$$Fb(S.t.\alpha) \leq c$$

and $t.\alpha$ is a sequence of transitions from s in $A_{G(Fb,c)}$.

□

Lemma 18. *Let T be a nonempty fair-bound persistent set in a state $s = \text{final}(S)$ in $A_{R(\text{Fb},c)}$ and let $\alpha.t.\gamma$ be a sequence of transitions from s in $A_{G(\text{Fb},c)}$ such that α is nonempty, $\forall i \in \text{dom}(\alpha) : \alpha_i \notin T$, and $t \in T$. Then, $t.\alpha.\gamma$ is a sequence of transitions from s in $A_{G(\text{Fb},c)}$.*

Proof. By Requirement 3 of Definition 4.12 of fair-bound persistent sets, $t \leftrightarrow \alpha$. Thus, $t.\alpha.\gamma$ is a sequence of transitions from s in A_G. By Requirements 1 and 2 of Definition 4.12 of fair-bound persistent sets, $Fb(S.t) \leq c$ and t is not a release operation. Thus, by Lemma 17,

$$Fb(S.t.\alpha) \leq Fb(S.\alpha)$$

Assume that γ_1 exceeds the bound from *final*$(S.t.\alpha)$, yet t does not exceed the bound from *final*$(S.\alpha)$ and γ_1 does not exceed the bound from *final*$(S.\alpha.t)$. Then, t must be a release operation that enables a transition t' such that $t'.tid$ has a lower yield count than $\gamma_1.tid$ has in *final*$(S.t.\alpha)$, because otherwise γ_1 would also exceed the bound from *final*$(S.\alpha)$. Because t is not a release operation, we have a contradiction. Thus,

$$Fb(S.t.\alpha.\gamma_1) \leq c$$

Because $t \leftrightarrow \alpha$, *final*$(S.t.\alpha.\gamma_1)$ = *final*$(S.\alpha.t.\gamma_1)$ and thus each transition in γ exe-

cutes from exactly the same state in $S.t.\alpha.\gamma$ as it does in $S.\alpha.t.\gamma$. Thus, by Definition 2.11 of the fair bound,

$$Fb(S.t.\alpha.\gamma) \leq c$$

Thus, $t.\alpha.\gamma$ is a sequence of transitions from s in $A_{G(Fb,c)}$.

□

Theorem 19. *If T is a nonempty fair-bound persistent set in a state s in $A_{R(\mathrm{Fb},c)}$, then T is local sufficient in s.*

Proof. Let s be a state in $A_{R(Fb,c)}$ and let l be a local state reachable from s in $A_{G(Fb,c)}$ via a nonempty sequence ω of transitions.

Case 19.1. $\forall i \in \boldsymbol{dom}(\omega) : \omega_i \notin T$**.**
Let t be any transition in T. Consider the sequence $\omega' = t.\omega$. By Requirement 3 of Definition 4.12 of fair-bound persistent sets, $t \leftrightarrow \omega$. Thus, $\omega.t \in [\omega']$, and $\omega \in \mathit{Pref}([\omega'])$. By Requirements 1 and 2 of Definition 4.12 of fair-bound persistent sets, $Fb(S.t) \leq c$ and t is not a release operation. Thus, by Lemma 17, $t.\omega$ is a sequence of transitions from s in $A_{G(Fb,c)}$ and T is local sufficient in s.

Case 19.2. $\exists i \in \boldsymbol{dom}(\omega) : \omega_i \in T$**.**
Let $\omega = \alpha.t.\gamma$ such that $\forall i \in dom(\alpha) : \alpha_i \notin T$ and $t \in T$. Assume that α is empty. Then, T is local sufficient in s because $\omega_1 \in T$ and l is reachable via ω.

Assume that α is nonempty. Consider the sequence $\omega' = t.\alpha.\gamma$, i.e., ω with t moved to the first position. By Requirement 3 of Definition 4.12 of fair-bound persistent sets, $t \leftrightarrow \alpha$. Thus, $\omega' \in [\omega]$ and $\omega \in \mathit{Pref}([\omega'])$. By Lemma 18, $t.\alpha.\gamma$ is a sequence of transitions from s in $A_{G(Fb,c)}$, and T is local sufficient in s.

□

By Theorems 19 and 3, if Algorithm 6 explores a nonempty fair-bound persistent set in each state then it reaches all local states reachable in $A_{G(Fb,c)}$. By Theorem 4,

Algorithm 6 reaches all deadlock states reachable in $A_{G(Fb,c)}$, as well.

The fair bound often permits more partial-order reduction than the context, preemption, or delta bounds do because it bounds a maximum, rather than a cumulative value in each state. Fewer transitions increment the bound, so fewer transitions leave portions of the state space unreachable within the bound. The fair bound prunes only cycles in the state space, however. Thus, the fair bound does not provide a useful incremental guarantee. The fair bound is primarily useful to prune cycles in cyclic state spaces. In Chapter 6, we combine fair-bounded search with other bounds to provide incremental coverage guarantees, as well.

4.3 Discussion

We defined two properties of bound functions that enable partial-order reduction. Stable bound functions permit deadlock-state reachability, and extensible bound functions permit local state reachability. Most bound functions from prior work are neither stable nor extensible and thus interact poorly with partial-order reduction. These bound functions introduce artificial dependences between otherwise independent transitions, and the search must sacrifice partial-order reduction to compensate for these dependences.

We have identified constraints on sufficient sets for depth, context, preemption, delta, and fair-bounded search that permit limited partial-order reduction. These sufficient sets are not optimal. More aggressive partial-order reduction might be possible by more precisely identifying bound dependences. In Chapter 6, we discuss several ways to reduce the state space further. The primary purpose of this chapter is to show that partial-order reduction and bounded search can be combined by identifying dependences that the bound introduces and ensuring that the bound persistent set for that bound function accounts for these dependences.

In the next chapter, we present an algorithm to dynamically compute bound

persistent sets for each bound function at runtime. We present a general algorithm adapted from prior work [Flanagan and Godefroid, 2005] that must be specialized for each bound function. We generalize this dynamic partial-order reduction algorithm to search a bounded state space. Then, we specialize this algorithm for each bound function, and we prove that these specialized algorithms provide bounded coverage.

Chapter 5

Computing Bound Persistent Sets

In this section we present algorithms that explore a bound persistent set of transitions in each state. We present a general algorithm that we adapt from prior work, then specialize that algorithm for each bound function. First, we present a simplified version of Flanagan and Godefroid's dynamic partial-order reduction (DPOR) algorithm [Flanagan and Godefroid, 2005]. We prove that this simplified algorithm explores a persistent set from each state, and thus explores all local and deadlock states reachable in an acyclic state space. Then, we introduce bounded dynamic partial-order reduction (BPOR), a modified version of DPOR that computes a bound persistent set in each state. We specialize BPOR for various bound functions, and prove safety guarantees for each algorithm.

The modified DPOR algorithm that we present is different from the original DPOR algorithm in several ways. First, we omit several optimizations from the original algorithm and introduce them in Chapter 6 instead. In particular, the optimizations in Sections 6.1, 6.2, and 6.3 were all included in the original DPOR algorithm. We present these as optimizations for several reasons. First, our

results differentiate their effects on partial-order reduction so we can assess their importance. Second, the simplified proofs are easier to follow. Finally, we want to highlight how each optimization interacts with the bound function.

The second significant difference between the simplified algorithm and the original DPOR algorithm is that this simplified algorithm backtracks the most recent dependent transition by *each thread*. The original DPOR algorithm, in contrast, backtracks the most recent dependent transition among all threads. We include this change because some of the bounded algorithms require it and it simplifies the algorithm and the proofs. We also differentiate read operations from write operations. The original DPOR paper assumed that all accesses to the same variable conflict, whereas Definition 2.4 assumes that reads of the same variable commute.

5.1 Dynamic Partial-Order Reduction

Algorithm 7 is a simplified version of Flanagan and Godefroid's dynamic partial-order reduction (DPOR) algorithm [Flanagan and Godefroid, 2005]. First, we step through Algorithm 7. Then, we prove that Algorithm 7 computes a persistent set in each state, and thus explores all reachable local and deadlock states for acyclic state spaces by Theorem 5 and Theorem 4, respectively.

The procedure **Explore** in Algorithm 7 recursively explores the state space from a state $s = \mathit{final}(S)$. Lines 4-10 create backtrack points. For each thread u, Line 6 computes the most recent transition in S by each thread v that is dependent with $\mathit{next}(s, u)$. For each such dependence, Line 7 creates a backtrack point to reverse the order of the dependent transitions in a future execution.

Lines 11-16 recursively explore the state space from s. Line 11 initializes the backtrack set with an arbitrary thread enabled in s. Line 15 recursively explores the next transition by each thread in the backtrack set. During this recursive search, additional threads may be added to the backtrack set in s.

Algorithm 7 Dynamic partial-order reduction [Flanagan and Godefroid, 2005].

```
 1: Initially, Explore(ε) from s_0
 2: procedure Explore(S) begin
 3:   Let s = final(S)
      # Add backtracking points for each thread's next transition.
 4:   for all u ∈ Tid do
 5:     for all v ∈ Tid | v ≠ u do
          # Find most recent dependent transition.
 6:       if ∃i = max({i ∈ dom(S) | S_i ↛ next(s,u) and S_i.tid = v}) then
 7:         Backtrack(S, i, u)
 8:       end if
 9:     end for
10:   end for
      # Continue the search by exploring successor states.
11:   choose one thread u ∈ enabled(s) and add to backtrack(s)
12:   Let visited = ∅
13:   while ∃u ∈ (enabled(s) ∩ backtrack(s) \ visited) do
14:     add u to visited
15:     Explore(S.next(s,u))
16:   end while
17: end
18: procedure Backtrack(S, i, u) begin
19:   if u ∈ enabled(pre(S,i)) then
20:     Add u to backtrack(pre(S,i))
21:   else
22:     backtrack(pre(S,i)) = enabled(pre(S,i))
23:   end if
24: end
```

The procedure **Backtrack** creates a backtrack point to reverse the order of the dependent transitions S_i and $next(final(S), u)$. If u is enabled in $pre(S, i)$, the state prior to S_i, then Line 20 adds u to the backtrack set in $pre(S, i)$. Otherwise, Line 22 conservatively adds *all* enabled threads to the backtrack set in $pre(S, i)$.

To prove that Algorithm 7 computes a persistent set in each state, we introduce a postcondition that Algorithm 7 guarantees before leaving each state s. Algorithm 7 explores the next transition by each thread in the backtrack set, then pops the most recent transition off the stack and returns to the previous state. Be-

fore returning to the previous state, Algorithm 7 guarantees postcondition *PC*. This post-condition is adapted from the postcondition that the original DPOR algorithm guarantees in each state [Flanagan and Godefroid, 2005].

Definition 5.1. ***PC* for Explore(S) for DPOR.**

$\forall u \forall \omega : \mathit{Post}(S.\omega, \mathit{len}(S), u)$

Definition 5.2. ***Post*(S, k, u) for DPOR.**

$\forall v$: **if** $i = \mathit{max}(\{i \in \mathit{dom}(S) \mid S_i \not\leftrightarrow \mathit{next}(\mathit{final}(S), u)$ **and** $S_i.\mathit{tid} = v\})$ **and** $i \leq k$ **then**

if $u \in \mathit{enabled}(\mathit{pre}(S, i))$ **then** $u \in \mathit{backtrack}(\mathit{pre}(S, i))$

else $\mathit{backtrack}(\mathit{pre}(S, i)) = \mathit{enabled}(\mathit{pre}(S, i))$

Postcondition *PC* guarantees that for each sequence ω of transitions from s in A_G and for each thread u, condition *Post* holds. Intuitively, $\mathit{Post}(S.\omega, \mathit{len}(S), u)$ requires backtrack points for the dependence between $\mathit{next}(\mathit{final}(S.\omega), u)$ and its most recent dependent transition in S by each thread v, if such a dependence exists.

$\mathit{Post}(S, k, u)$ identifies the most recent transition in S by each thread v that is dependent with $\mathit{next}(\mathit{final}(S), u)$, if any. Note that *PC* sends the argument $S.\omega$ to *Post*, but *Post* refers to it internally as S. If u is enabled in $\mathit{pre}(S, i)$, the state prior to dependent transition S_i, then *Post* requires that u be in the backtrack set in $\mathit{pre}(S, i)$. Otherwise, *Post* conservatively requires that *all* enabled threads be in the backtrack set in $\mathit{pre}(S, i)$. Next, we prove that the set of transitions explored from s is persistent in s, provided that *PC* holds for each recursive call to **Explore**. Lemma 20, Lemma 21, and Theorem 22 are all adapted from the original DPOR algorithm's correctness proof [Flanagan and Godefroid, 2005].

Lemma 20. *Whenever Algorithm 7 backtracks a state $s = \mathrm{final}(S)$, the set T of transitions explored from s is persistent in s, provided that postcondition* PC *holds for every recursive call* ***Explore***$(S.t)$ *for all $t \in T$.*

Proof. Let $T = \mathit{next}(s, u) \mid u \in \mathit{backtrack}(s)$. Proceed by contradiction. Assume that there exists a nonempty sequence α of transitions from s in A_G and a transition $t \in T$ such that

1. $\forall i \in \mathit{dom}(\alpha) : \alpha_i \notin T$
2. t is dependent with $\text{last}(\alpha)$

Let $n = \mathit{len}(\alpha)$ and let $\omega = \alpha_1 \ldots \alpha_{n-1}$, i.e., α with its last transition removed. Let there be no prefixes of α that also meet the criteria above, and thus

3. $t \leftrightarrow \omega$

Let $u = \text{last}(\alpha).\mathit{tid}$. Assume that $t.\mathit{tid} = u$. Because $t \leftrightarrow \omega$,

$$t = \mathit{next}(\mathit{final}(S), u) = \mathit{next}(\mathit{final}(S.\omega), u) = \text{last}(\alpha)$$

Thus, $\text{last}(\alpha) \in T$ and we have a contradiction.

Assume that $t.\mathit{tid} \neq u$. Consider the postcondition

$$\mathit{Post}(S.t.\omega, \mathit{len}(S) + 1, u)$$

for the recursive call **Explore**$(S.t)$. Because $t \leftrightarrow \omega$, t is the most recent transition by $t.\mathit{tid}$ that is dependent with $\mathit{next}(\mathit{final}(S.t.\omega), u)$. Thus, by Definition 5.2 of *Post*, either $u \in \mathit{backtrack}(s)$, or $\mathit{backtrack}(s) = \mathit{enabled}(s)$ and thus $\alpha_1 \in T$. In either case, we have a contradiction.

□

Thus, if postcondition PC holds in each state s explored by Algorithm 7, then the set of transitions explored from s is persistent in s.

We next prove that postcondition PC holds in each state s explored by Algorithm 7. First, we prove a lemma that simplifies the inductive step. This lemma

is more general than needed because DPOR requires only the first of the two cases, where $\exists\beta : \omega.\beta \in [\omega']$ **and** $\beta \leftrightarrow \mathit{next}(\mathit{final}(S.\omega), u)$. We prove the more general case for this lemma because some bounds do require the more general case. This lemma proves that the inductive hypothesis in the subsequent proof, $\mathit{Post}(S.\omega', \mathit{len}(S{+}1), u)$ is sufficient to obtain the desired conclusion, $\mathit{Post}(S.\omega, \mathit{len}(S), u)$.

Lemma 21. *Let $s = \mathrm{final}(S)$ be a state in A_R, let ω and ω' be nonempty sequences of transitions from s in A_G, and let u be a thread such that*

1. $\exists\beta : \omega.\beta \in [\omega']$ ***and*** $\beta \leftrightarrow \mathrm{next}(\mathrm{final}(S.\omega), u)$, *or*
2. $\exists\beta : \omega'.\beta \in [\omega]$ ***and*** $\beta \leftrightarrow \mathrm{next}(\mathrm{final}(S.\omega), u)$

Then, $\mathrm{Post}(S.\omega', \mathrm{len}(S) + 1, u) \implies \mathrm{Post}(S.\omega, \mathrm{len}(S), u)$.

Proof. Because $\beta \leftrightarrow \mathit{next}(\mathit{final}(S.\omega), u)$,

$$\mathit{next}(\mathit{final}(S.\omega), u) = \mathit{next}(\mathit{final}(S.\omega'), u)$$

Assume that for some thread v in Definition 5.2 of $\mathit{Post}(S.\omega, \mathit{len}(S), u)$, $i > k$. Then, *Post* does not require any backtrack points for v.

Assume that for some thread v in Definition 5.2 of $\mathit{Post}(S.\omega, \mathit{len}(S), u)$, $i \leq k$. Then, because $\beta \leftrightarrow \mathit{next}(\mathit{final}(S.\omega), u)$, i is the same for thread v in $\mathit{Post}(S.\omega', \mathit{len}(S), u)$. Thus, by Definition 5.2 of *Post*,

$$\mathit{Post}(S.\omega, \mathit{len}(S), u) \text{ iff } \mathit{Post}(S.\omega', \mathit{len}(S), u) \tag{5.1}$$

Because Definition 5.2 of *Post* requires that i be less than or equal to k,

$$\mathit{Post}(S.\omega', \mathit{len}(S) + 1, u) \implies \mathit{Post}(S.\omega', \mathit{len}(S), u)$$

Thus, by Equation 5.1,

$$Post(S.\omega', len(S) + 1, u) \implies Post(S.\omega, len(S), u)$$

□

Theorem 22. *Whenever Algorithm 7 backtracks a state* $s = \text{final}(S)$ *in an acyclic state space, the postcondition* Post *for* ***Explore***(S) *is satisfied, and the set* T *of transitions explored from* s *is persistent in* s*.*

Proof. The proof is by induction on the order in which states are backtracked.

Base case.

Because the search is finite and performed in depth-first order, the first backtracked state must be a deadlock state in which no transition is enabled. Thus, the post-condition for the first backtracked state is

$$\forall u : Post(S, len(S), u)$$

and is directly established by Lines 4-10 in Algorithm 7.

Inductive case.

Assume that each call to **Explore**$(S.t)$ satisfies its postcondition. By Lemma 20, T is persistent in s. Show that **Explore**(S) satisfies its postcondition for any sequence ω of transitions from s in A_G and for any thread u.

Case 22.1. $\forall i \in \textbf{\textit{dom}}(\omega) : \omega_i \notin T$ **and** $u \in \textbf{\textit{backtrack}}(s)$**.**

Because $u \in backtrack(s)$, $next(s, u) \in T$. Thus, by Definition 2.5 of persistent sets, $next(s, u) \leftrightarrow \omega$, and

$$next(final(S.\omega), u) = next(s, u)$$

Thus, $next(final(S.\omega), u) \leftrightarrow \omega$, and therefore $Post(S.\omega, len(S), u)$ iff $Post(S, len(S), u)$. The latter is directly established by Lines 4-10 in Algorithm 7.

Case 22.2. $\forall i \in \textbf{\textit{dom}}(\omega) : \omega_i \notin T$ **and** $u \notin \textbf{\textit{backtrack}}(s)$**.**

Let t be any transition in T. Consider the sequence $\omega' = t.\omega$. By Definition 2.5 of persistent sets, $t \leftrightarrow \omega$, and thus

$$\omega.t \in [\omega']$$

By the inductive hypothesis for the recursive call **Explore**$(S.t)$,

$$Post(S.\omega', len(S) + 1, u)$$

Assume that t is dependent with $next(final(S.\omega'), u)$. Because $t \leftrightarrow \omega$, t is the most recent transition by $t.tid$ that is dependent with $next(final(S.\omega'), u)$. Thus, by Definition 5.2 of *Post*, either $u \in backtrack(s)$ or $backtrack(s) = enabled(s)$ and thus $\omega_1 \in T$. In either case, we have a contradiction.

Assume that $t \leftrightarrow next(final(S.\omega'), u)$. Because $t \in T$ and $u \notin backtrack(s)$, $t.tid \neq u$. Thus, $next(final(S.\omega), u) = next(final(S.\omega'), u)$ and

$$t \leftrightarrow next(final(S.\omega), u)$$

Thus, by Lemma 21 where $\beta = t$ and $\omega.t \in [\omega']$,

$$Post(S.\omega, len(S), u)$$

Case 22.3. $\exists i \in \textbf{\textit{dom}}(\omega) : \omega_i \in T$**.**

Let $\omega = \alpha.t.\gamma$ such that

1. $\forall i \in dom(\alpha) : \alpha_i \notin T$
2. $t \in T$

Consider the sequence $\omega' = t.\alpha.\gamma$, i.e., ω with t moved to the beginning. By Definition 2.5 of persistent sets, $t \leftrightarrow \alpha$. Thus, by Definition 2.2 of a trace,

$$\omega' \in [\omega]$$

By the inductive hypothesis for the recursive call **Explore**$(S.t)$,

$$\mathit{Post}(S.\omega', \mathit{len}(S) + 1, u)$$

and thus by Lemma 21 where β is empty and $\omega \in [\omega']$,

$$\mathit{Post}(S.\omega, \mathit{len}(S), u)$$

□

Thus, Algorithm 7 explores a persistent set T of transitions from each state s. By Theorem 5, T is local sufficient in s, and by Theorem 4, T is deadlock sufficient in s. Thus, Algorithm 7 explores all reachable deadlock and local states in an acyclic state space. We next modify Algorithm 7 to compute a bound persistent set in each state for various bound evaluation functions.

5.2 Bounded Partial-Order Reduction

Algorithm 8 presents bounded dynamic partial-order reduction (BPOR), a modified version of DPOR (Algorithm 7) that computes a bound persistent set in each state. We specialize BPOR to compute bound persistent sets for the depth, context, preemption, delta, and fair bounds. We prove that the resulting algorithms guarantee bounded coverage. First, we highlight differences between Algorithm 8 and Algorithm 7.

The procedure **Explore** in Algorithm 8 is common to all bound evaluation

Algorithm 8 BPOR with bound function Bv and bound c.

```
1:  Initially, Explore(ϵ) from s_0
2:  procedure Explore(S) begin
3:    Let s = final(S)
      # Add backtracking points for each thread's next transition.
4:    for all u ∈ Tid do
5:      for all v ∈ Tid | v ≠ u do
          # Find most recent dependent transition.
6:        if ∃i = max({i ∈ dom(S) | (S_i, next(s,u)) ∈ D and S_i.tid = v}) then
7:          Backtrack(S, i, u)
8:        end if
9:      end for
10:   end for
      # Continue the search by exploring successor states.
11:   Initialize(S)
12:   Let visited = ∅
13:   while ∃u ∈ (enabled(s) ∩ backtrack(s) \ visited) do
14:     add u to visited
15:     if Bv(S.next(s,u)) ≤ c then
16:       Explore(S.next(s,u))
17:     end if
18:   end while
19: end
```

functions. **Explore** is similar to the **Explore** procedure from Algorithm 7, except that Algorithm 8 explores only transitions that do not exceed the bound. Line 15 ensures that thread u's next transition is within bound c.

The **Backtrack** and **Initialize** procedures are specific to each bound evaluation function. We define these procedures for each bound evaluation function but do not specify their behavior for the generic algorithm. The **Backtrack** procedure adds backtrack points for dependent transitions, and for bound dependences. **Initialize** initializes $backtrack(final(S))$ with at least one enabled transition that does not exceed the bound, if one exists. The initial transition affects the size of the final bound persistent set, so each bound function carefully selects the initial transition to maximize its likelihood of reaching each state via the cheapest sequence of tran-

sitions first. We provide specialized versions of **Backtrack** and **Initialize** as we discuss each bound evaluation function.

BPOR computes local sufficient sets for each bound, but it cannot compute deadlock sufficient sets that are not also local sufficient. BPOR relies on local state reachability because it is dynamic – it must explore subsequent dependent transitions to realize that they require backtrack points. Without local state reachability, BPOR may never reach subsequent dependent transitions and may thus never explore backtrack points that lead to new deadlock states within the bound. DPOR also requires local state reachability. Flanagan and Godefroid assume that the state space is acyclic. Persistent sets are local sufficient in an acyclic state space so this distinction does not matter [Godefroid, 1996].

Local sufficient sets for bound functions that are not extensible require additional backtrack points to guarantee local state reachability within the bound. Each bound's **Initialize** and **Backtrack** procedures incorporate these additional backtrack points. In the next section, we intuitively describe the types of conservative backtrack points that various bound functions require.

5.2.1 Conservative Backtrack Points

The **Initialize** and **Backtrack** procedures for each bound function add conservative backtrack points to accommodate dependences that the bound introduces. With these conservative backtrack points, BPOR guarantees bounded coverage even though the search does not explore transitions that exceed the bound. We overview dependences that the bound introduces and the backtrack points that they require.

The cheapest enabled transition is often more likely than other enabled transitions to reach new states via the cheapest path. For some bound functions, the cheapest transition *always* reaches subsequent states via the cheapest path. For example, the cheapest transition in a state s always provides the cheapest path to

subsequent states for the context, preemption, and delta bounds, provided that the cheapest transition in s is unique. More expensive transitions leave states that are reachable within the bound unreachable, so the search must conservatively backtrack cheaper transitions when it explores a more expensive one.

When a transition increments the bound, the cheapest alternative path to the states it reaches may require a conservative backtrack point in a prior state. If the search does not explore that alternative path, then it may lose bounded coverage. Transitions that cost more as a result of a prior transition are dependent with that prior transition, so some bound functions insert conservative backtrack points earlier in the stack to compensate for these dependences.

When the enabledness of threads is an input to a bound function, transitions that enable and disable other threads may introduce dependences among otherwise independent transitions. For example, release operations may be dependent with subsequent transitions whose cost they affect. Some bound functions backtrack *all* enabled threads prior to release operations to compensate for these dependences. Identifying the specific dependent transitions instead is challenging, but doing so could further optimize the search.

Each bound function requires some combination of these conservative backtrack points to maintain coverage. We discuss each bound function in Chapter 4, and describe the conservative backtrack points that it requires. We next show how to compute bound persistent sets for each bound function.

5.2.2 Computing Depth-Bound Persistent Sets

BPOR cannot compute depth-bound persistent sets unless it also supplies local state reachability because, as described in Section 4.2.2, depth-bound persistent sets are deadlock sufficient but not local sufficient. Depth-bounded search sacrifices all partial-order reduction to achieve local state reachability. Thus, the **Initialize**

Algorithm 9 BPOR procedures for depth-bounded search.

```
1: procedure Initialize(S) begin
2:     backtrack(final(S)) = enabled(final(S))
3: end
4: procedure Backtrack(S, i, u) begin
       # Do nothing.
5: end
```

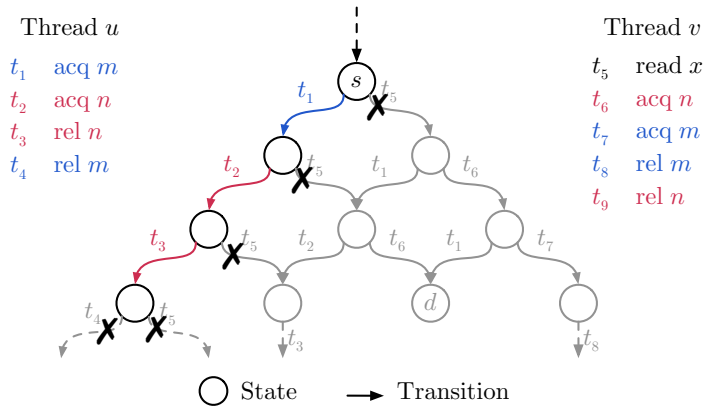


Figure 5.1: Counter-example for depth-bounded DPOR with bound 3. DPOR may not reach deadlock state d even though d is reachable within the bound.

procedure in Algorithm 9 adds all enabled threads to the backtrack set in each state. The **Backtrack** procedure does nothing because the **Initialize** procedure already added all enabled threads to the backtrack set. BPOR computes a local sufficient set for depth-bounded search in each state with these procedures, but it performs no partial-order reduction.

Figure 5.1 illustrates why BPOR must sacrifice partial-order reduction to compute depth-bound persistent sets. Assume that the depth bound in this example is three. Deadlock state d is reachable from initial state s_0 within the bound via various sequences of transitions, for example $t_5.t_6.t_1$. BPOR may never explore d, however. Assume that BPOR first explores the sequence $t_1.t_2.t_3$. Then, both enabled transitions t_4 and t_5 exceed the bound, so the search does not explore t_4 or

Algorithm 10 BPOR procedures for context-bounded search.

```
1: procedure Initialize(S) begin
2:   if last(S).tid ∈ enabled(final(S)) then
3:     add last(S).tid to backtrack(final(S))
4:   else
5:     backtrack(final(S)) = enabled(final(S))
6:   end if
7: end
8: procedure Backtrack(S, i, u) begin
9:   backtrack(pre(S, i)) = enabled(pre(S, i))
10: end
```

t_5. The next transition by thread v, t_5, is not dependent with any of the explored transitions, and thus BPOR does not explore t_5 from any visited state. Thus, the search does not explore d, and the search is not deadlock sufficient. If the search were local sufficient it would explore d, but it cannot guarantee local state reachability without sacrificing partial-order reduction.

Depth-bound persistent sets do provide deadlock state reachability in depth-bounded search. The BPOR algorithm cannot guarantee that it has explored a depth-bound persistent set without exploring a local sufficient set, however. BPOR prunes a transition t in a state s under the assumption that after exploring a series of independent transitions, t will still be reachable. In depth-bounded search, every explored transition increments the bounded value and may leave a portion of the state space unreachable. Thus, BPOR sacrifices partial-order reduction, as shown in Algorithm 9, to guarantee deadlock state reachability for depth-bounded search. In Chapter 6, we optimize BPOR and allow limited partial-order reduction with depth-bounded search and in Chapter 7, we show how to reach all local states within the depth-bounded without sacrificing partial-order reduction at all.

5.2.3 Computing Context-Bound Persistent Sets

In this section we specialize Algorithm 8 to compute context-bound persistent sets.

Deadlock sufficient context-bound persistent sets are deadlock sufficient but not local sufficient, as shown in Section 4.2.3. To compute deadlock sufficient context-bound persistent sets, BPOR must compute local sufficient sets because BPOR relies on local state reachability, as described in Section 5.2.2. Thus, we specialize BPOR for context-bound persistent sets and prove this algorithm's safety guarantees.

Algorithm 10 contains the **Initialize** and **Backtrack** procedures to compute context-bound persistent sets. The **Initialize** procedure adds last(S).*tid* to the backtrack set in *final*(S), if it is enabled there, because its next transition does not require a context switch and is thus free. Otherwise, **Initialize** adds all enabled threads to the backtrack set because they all require a context switch and by Definition 4.9 of context-bound persistent sets, transitions in the context-bound persistent set must be cheaper than transitions not in the context-bound persistent set. Similarly, the **Backtrack** procedure adds all enabled threads to the backtrack set in $pre(S, i)$.

Assume that Algorithm 8 uses the procedures in Algorithm 10. To prove that Algorithm 8 computes a context-bound persistent set in each state, we modify the postconditions from Section 5.1 for context-bounded search. Before popping the most recent transition off the stack and returning to the previous state, Algorithm 8 guarantees postcondition *PC*.

Definition 5.3. ***PC* for Explore(S) for context-bounded BPOR.**
$\forall u \forall \omega :$ **if** $Cs(S.\omega) \leq c$ **then** $Post(S.\omega, len(S), u)$

Definition 5.4. ***Post*(S, k, u) for context-bounded BPOR.**
$\forall v :$ **if** $i = max(\{i \in dom(S) \mid S_i \not\rightsquigarrow next(final(S), u)$ **and** $S_i.tid = v\})$ **and** $i \leq k$ **then**
$backtrack(pre(S, i)) = enabled(pre(S, i))$

Definition 5.3 is similar to Definition 5.1, except that it requires that *Post* hold only

for sequences of transitions that are within the context bound. Definition 5.4 is similar to Definition 5.2, but Definition 5.4 requires that all enabled transitions be in the backtrack set in $pre(S, i)$, prior to dependent transition S_i. Next, we prove that the set of transitions explored from s is context-bound persistent in s, provided that PC holds for each recursive call to **Explore**.

Lemma 23. *Whenever Algorithm 8 backtracks a state $s = \text{final}(S)$, the set T of transitions explored from s is context-bound persistent in s, provided that postcondition* PC *holds for every recursive call* ***Explore***$(S.t)$ *for all $t \in T$.*

Proof. Let $T = next(s, u) \mid u \in backtrack(s)$. Show that if T violates any requirement in Definition 4.9 of context-bound persistent sets, then we have a contradiction.

Case 23.1. T **violates Requirement 1.**

Proceed by contradiction. Assume that there exist transitions $t \in T$ and $t' \notin T$ such that $t.tid, t'.tid \in enabled(s)$ and $Cs(S.t') \leq Cs(S.t)$. Assume $Cs(S.t') < Cs(S.t)$. Then, the **Initialize** procedure adds t' to the backtrack set at Line 3 in Algorithm 10, so $t' \in T$ and we have a contradiction.

Assume that $Cs(S.t') = Cs(S.t)$. Then, either all transitions require a context switch from s, or the search added t to the backtrack set from the **Backtrack** procedure. If all transitions require a context switch, then Line 5 of Algorithm 10 adds t' to the backtrack set. Otherwise, Line 9 of Algorithm 10 must have added t to the backtrack set, and it must therefore have added t' to the backtrack set as well. In either case, $t' \in T$ and we have a contradiction.

Case 23.2. T **violates Requirement 2.**

Proceed by contradiction. Assume that there exists a nonempty sequence α of transitions from s in $A_{G(Cs,c)}$ and a transition $t \in T$ such that

1. $Cs(S.t) < Cs(S.\alpha_1)$
2. $\forall i \in dom(\alpha) : \alpha_i \notin T$

3. t is dependent with last(α)

Let $n = \mathit{len}(\alpha)$ and let $\omega = \alpha_1 \ldots \alpha_{n-1}$, i.e., α with its last transition removed. Let there be no prefixes of α that also meet the criteria above, and thus

4. $t \leftrightarrow \omega$

Let $u = \text{last}(\alpha).\mathit{tid}$. Assume that $t.\mathit{tid} = u$. Because $t \leftrightarrow \omega$,

$$t = \mathit{next}(\mathit{final}(S), u) = \mathit{next}(\mathit{final}(S.\omega), u) = \text{last}(\alpha)$$

Thus, last(α) $\in T$ and we have a contradiction. Assume that $t.\mathit{tid} \neq u$. Consider the postcondition

$$\mathit{Post}(S.t.\omega, \mathit{len}(S) + 1, u)$$

for the recursive call **Explore**($S.t$). By Lemma 9, $t.\omega$ is a sequence of transitions from s in $A_{G(Cs,c)}$. Because $t \leftrightarrow \omega$, t is the most recent transition by $t.\mathit{tid}$ that is dependent with $\mathit{next}(\mathit{final}(S.t.\omega), u)$. Thus, by Definition 5.4 of *Post*, $\mathit{backtrack}(s) = \mathit{enabled}(s)$ and thus $\alpha_1 \in T$, and we have a contradiction.

□

Thus, if postcondition PC holds in each state s that Algorithm 8 explores with the **Backtrack** procedure from Algorithm 10, then the set of transitions it explores from s is context-bound persistent in s. Next, we prove that postcondition PC holds in each state s that Algorithm 8 explores. First, we prove a lemma to simplify the inductive step that is very similar to Lemma 21 and its proof.

Lemma 24. *Let s = final(S) be a state in $A_{R(\text{Cs},c)}$, let ω and ω' be nonempty sequences of transitions from s in $A_{G(\text{Cs},c)}$, and let u be a thread such that*

1. $\exists \beta : \omega.\beta \in [\omega']$ ***and*** $\beta \leftrightarrow \text{next}(\text{final}(S.\omega), u)$, *or*
2. $\exists \beta : \omega'.\beta \in [\omega]$ ***and*** $\beta \leftrightarrow \text{next}(\text{final}(S.\omega), u)$

Then, $\text{Post}(S.\omega', \text{len}(S)+1, u) \implies \text{Post}(S.\omega, \text{len}(S), u)$.

Proof. Because $\beta \leftrightarrow \mathit{next}(\mathit{final}(S.\omega), u)$,

$$\mathit{next}(\mathit{final}(S.\omega), u) = \mathit{next}(\mathit{final}(S.\omega'), u)$$

Assume that for some thread v in Definition 5.4 of $\mathit{Post}(S.\omega, \mathit{len}(S), u)$, $i > k$. Then, *Post* does not require any backtrack points for v.

Assume that for some thread v in Definition 5.4 of $\mathit{Post}(S.\omega, \mathit{len}(S), u)$, $i \leq k$. Then, because $\beta \leftrightarrow \mathit{next}(\mathit{final}(S.\omega), u)$, i is the same for thread v in $\mathit{Post}(S.\omega', \mathit{len}(S), u)$. Thus, by Definition 5.4 of *Post*,

$$\mathit{Post}(S.\omega, \mathit{len}(S), u) \text{ iff } \mathit{Post}(S.\omega', \mathit{len}(S), u) \tag{5.2}$$

Because Definition 5.4 of *Post* requires that i be less than or equal to k,

$$\mathit{Post}(S.\omega', \mathit{len}(S)+1, u) \implies \mathit{Post}(S.\omega', \mathit{len}(S), u)$$

Thus, by Equation 5.2,

$$\mathit{Post}(S.\omega', \mathit{len}(S)+1, u) \implies \mathit{Post}(S.\omega, \mathit{len}(S), u)$$

□

Theorem 25. *Whenever a state* $s = \text{final}(S)$ *is backtracked during the search performed by Algorithm 8 in an acyclic state space, the postcondition* Post *for* ***Explore***(S) *is satisfied, and the set* T *of transitions explored from* s *is context-bound persistent in* s.

Proof. The proof is by induction on the order in which states are backtracked.

Base case.

Because the search is acyclic and is performed in depth-first order, the first backtracked state must be either a deadlock state in which no transition is enabled, or a state in which all transitions exceed the bound. Thus, the postcondition for the first backtracked state is

$$\forall u : \mathit{Post}(S, \mathit{len}(S), u)$$

and is directly established by Lines 4-10 in Algorithm 8.

Inductive case.

Assume that each call to **Explore**$(S.t)$ satisfies its postcondition. By Lemma 23, T is context-bound persistent in s. Show that **Explore**(S) satisfies its postcondition for any sequence ω of transitions from s in $A_{G(Cs,c)}$ and for any thread u.

Case 25.1. $\forall i \in \mathbf{dom}(\omega) : \omega_i \notin T$ **and** $u \in \mathbf{backtrack}(s)$**.**

Because $u \in \mathit{backtrack}(s)$, $\mathit{next}(s, u) \in T$. Thus, by Definition 4.9 of context-bound persistent sets, $\mathit{next}(s, u) \leftrightarrow \omega$, and thus

$$\mathit{next}(\mathit{final}(S.\omega), u) = \mathit{next}(s, u)$$

Thus, $\mathit{next}(\mathit{final}(S.\omega), u) \leftrightarrow \omega$, and therefore $\mathit{Post}(S.\omega, \mathit{len}(S), u)$ iff $\mathit{Post}(S, \mathit{len}(S), u)$. The latter is directly established by Lines 4-10 in Algorithm 8.

Case 25.2. $\forall i \in \mathbf{dom}(\omega) : \omega_i \notin T$ **and** $u \notin \mathbf{backtrack}(s)$**.**

Let t be any transition in T. Consider the sequence $\omega' = t.\omega$. By Definition 4.9 of context-bound persistent sets, $\mathit{Cs}(S.t) < \mathit{Cs}(S.\omega_1)$ and $t \leftrightarrow \omega$. Thus, by Lemma 9, ω' is a sequence of transitions from s in $A_{G(Cs,c)}$. By the inductive hypothesis for the recursive call **Explore**$(S.t)$,

$$\mathit{Post}(S.\omega', \mathit{len}(S) + 1, u)$$

Assume that t is dependent with $\mathit{next}(\mathit{final}(S.\omega'), u)$. Because $t \leftrightarrow \omega$, t is the most recent transition by $t.\mathit{tid}$ that is dependent with $\mathit{next}(\mathit{final}(S.\omega'), u)$. Thus, by Definition 5.4 of *Post*, $\mathit{backtrack}(s) = \mathit{enabled}(s)$ and thus $\omega_1 \in T$ and we have a contradiction.

Assume that $t \leftrightarrow \mathit{next}(\mathit{final}(S.\omega'), u)$. Because $t \leftrightarrow \omega$, $\omega.t \in [\omega']$. Because $t \in T$ and $u \notin \mathit{backtrack}(s)$, $t.\mathit{tid} \neq u$. Thus, $\mathit{next}(\mathit{final}(S.\omega), u) = \mathit{next}(\mathit{final}(S.\omega'), u)$ and

$$t \leftrightarrow \mathit{next}(\mathit{final}(S.\omega), u)$$

Thus, by Lemma 24 where $\beta = t$ and $\omega.t \in [\omega']$,

$$\mathit{Post}(S.\omega, \mathit{len}(S), u)$$

Case 25.3. $\exists i \in \mathbf{dom}(\omega) : \omega_i \in T$**.**
Let $\omega = \alpha.t.\gamma$ such that

1. $\forall i \in \mathit{dom}(\alpha) : \alpha_i \notin T$
2. $t \in T$

Assume that α is empty. Then, $\omega_1 \in T$, and by the inductive hypothesis

$$\mathit{Post}(S.\omega, \mathit{len}(S) + 1, u)$$

Thus, because Definition 5.4 of *Post* requires that $i \leq k$,

$$\mathit{Post}(S.\omega, \mathit{len}(S), u)$$

as required.

Assume that α is nonempty. Consider the sequence $\omega' = t.\alpha.\gamma$, i.e., ω with t moved to the beginning. By Definition 4.9 of context-bound persistent sets,

$Cs(S.t) < Cs(S.\alpha_1)$ and $t \leftrightarrow \alpha$. Thus, by Definition 2.2 of a trace, $\omega' \in [\omega]$. By Lemma 7, ω' is a sequence of transitions from s in $A_{G(Cs,c)}$. By the inductive hypothesis for the recursive call **Explore**$(S.t)$,

$$Post(S.\omega', len(S) + 1, u)$$

and thus by Lemma 24 where β is empty and $\omega \in [\omega']$,

$$Post(S.\omega, len(S), u)$$

□

Thus, Algorithm 8 explores a context-bound persistent set in each state with the procedures from Algorithm 10. By Theorem 10 and Theorem 4, Algorithm 8 explores all local and deadlock states reachable within the bound. This search performs relatively little partial-order reduction, however. If all transitions in a state s require a context switch or if the search explores multiple transitions from s, then the search must explore *all* enabled transitions from s. Preemption-bounded search improves over this result because there exists a zero-cost transition in every state. In the next section, we specialize BPOR to compute preemption-bound persistent sets.

5.2.4 Computing Preemption-Bound Persistent Sets

BPOR can reduce the state space by computing preemption-bound persistent sets because preemption-bound persistent sets are local sufficient. Algorithm 11 contains the **Initialize** and **Backtrack** procedures for preemption-bounded search. **Initialize** adds the executing thread to the backtrack set in $final(S)$ if it is enabled there. Otherwise, **Initialize** adds any $u \in enabled(final(S))$ to the backtrack set. **Initialize** adds only one enabled thread to the backtrack set because the search may be able to prove that all local states reachable within the bound are reachable via that

Algorithm 11 BPOR procedures for preemption-bounded search.

```
 1: procedure Initialize(S) begin
 2:   if last(S).tid ∈ enabled(final(S)) then
 3:     add last(S).tid to backtrack(final(S))
 4:   else
 5:     add any u ∈ enabled(final(S)) to backtrack(final(S))
 6:   end if
 7: end
 8: procedure Backtrack(S, i, u) begin
 9:   AddBacktrackPoint(S, i, u)
10:   if j = max({j ∈ dom(S) | j = 0 or S_{j−1}.tid ≠ S_j.tid and j < i}) then
11:     AddBacktrackPoint(S, j, u)
12:   end if
13: end
14: procedure AddBacktrackPoint(S, i, u) begin
15:   if u ∈ enabled(pre(S, i)) then
16:     Add u to backtrack(pre(S, i))
17:   else
18:     backtrack(pre(S, i)) = enabled(pre(S, i))
19:   end if
20: end
```

initial transition.

The **Backtrack** procedure adds two backtrack points: at Line 9 it adds one prior to the most recent dependent transition S_i, and at Line 11 it adds one prior to the most recent transition to S_i at which the executing thread changed. The first backtrack point satisfies Requirement 2 of Definition 4.10 of preemption-bound persistent sets, and the second backtrack point satisfies Requirement 3 of Definition 4.10. The procedure **AddBacktrackPoint** adds u to the backtrack set if it is enabled in $pre(S, i)$; otherwise, it conservatively adds all enabled threads to the backtrack set.

To prove that Algorithm 8 computes a preemption-bound persistent set in each state, we modify the postconditions from Section 5.1 for preemption-bounded search. Before popping the most recent transition off the stack and returning to the

previous state, Algorithm 8 guarantees postcondition PC.

Definition 5.5. ***PC* for Explore(S) for preemption-bounded BPOR.**
$\forall u \forall \omega :$ **if** $Pb(S.\omega) \leq c$ **then** $Post(S.\omega, len(S), u)$

Definition 5.6. ***Post*(S, k, u) for preemption-bounded BPOR.**
$\forall v :$ **if** $i = max(\{i \in dom(S) \mid S_i \not\leftrightarrow next(final(S), u)$ **and** $S_i.tid = v\})$ **then**

1. **if** $i \leq k$ **then**
   if $u \in enabled(pre(S, i))$ **then** $u \in backtrack(pre(S, i))$
   else $backtrack(pre(S, i)) = enabled(pre(S, i))$

2. **if** $j = max(\{j \in dom(S) \mid j = 0$ **or** $S_{j-1}.tid \neq S_j.tid$ **and** $j < i\})$ **and** $j < k$ **then**
   if $u \in enabled(pre(S, j))$ **then** $u \in backtrack(pre(S, j))$
   else $backtrack(pre(S, j)) = enabled(pre(S, j))$

Definition 5.5 is similar to Definitions 5.1 and 5.3, except that it requires that $Post$ hold only for sequences of transitions that are reachable within the preemption bound. Definition 5.6 differs from Definitions 5.2 and 5.4 because it requires an additional backtrack point. This extra backtrack point guarantees that $ext(s, t)$ will be independent with transitions not in the local sufficient set when transitions cost the same amount.

$Post(S, k, u)$ identifies the most recent transition by each thread v that is dependent with $next(final(S), u)$, if any. Requirement 1 of $Post$ requires a backtrack point in $pre(S, i)$, the state prior to dependent transition S_i. This backtrack point satisfies Requirement 2 of Definition 4.10 of preemption-bound persistent sets. If u is enabled in $pre(S, i)$, u must be in the backtrack set in $pre(S, i)$. Otherwise, *all* enabled threads must conservatively be in the backtrack set in $pre(S, i)$. Requirement 2 of $Post$ requires a backtrack point prior to the most recent transition to S_i at

which the executing thread changed. This backtrack point satisfies Requirement 3 of Definition 4.10 of preemption-bound persistent sets.

We show that the set T of transitions explored from s is preemption-bound persistent in s, provided that PC holds for each recursive call to **Explore**.

Lemma 26. *Whenever a state $s = \text{final}(S)$ is backtracked by Algorithm 8, the set T of transitions explored from s is preemption-bound persistent in s, provided that postcondition* PC *holds for every recursive call* ***Explore***$(S.t)$ *for all $t \in T$.*

Proof. Let $T = \mathit{next}(s, u) \mid u \in \mathit{backtrack}(s)$. Show that if T violates any requirement in Definition 4.10 of preemption-bound persistent sets, then we have a contradiction.

Case 26.1. T **violates Requirement 1.**
Proceed by contradiction. Assume that there exist transitions $t \in T$ and $t' \notin T$ such that t and t' are both enabled in s and $Pb(S.t') < Pb(S.t)$. By Definition 2.9 of the preemption bound

$$t'.tid = \text{last}(S).tid$$

Thus, by Line 3 of Algorithm 11, $t'.tid \in \mathit{backtrack}(s)$ and thus $t' \in T$, and we have a contradiction.

Case 26.2. T **violates Requirement 2.**
Proceed by contradiction. Assume that there exists a nonempty sequence α of transitions from s in $A_{G(Pb,c)}$ and a transition $t \in T$ such that, if we let $u = \text{last}(\alpha).tid$:

1. $\forall i \in \mathit{dom}(\alpha) : \alpha_i \notin T$
2. $Pb(S.t) < Pb(S.\alpha_1)$
3. t is dependent with $\text{last}(\alpha)$ or with $\mathit{next}(\mathit{final}(S.\alpha), u)$

Let $n = \mathit{len}(\alpha)$ and let $\omega = \alpha_1 \ldots \alpha_{n-1}$, i.e., α with its last transition removed. Let there be no prefixes of α that also meet the criteria above, and thus

4. $t \leftrightarrow \omega$ and $\forall i \in \mathit{dom}(\omega) : t \leftrightarrow \mathit{next}(\mathit{final}(S.\omega_1 \ldots \omega_i), \omega_i.\mathit{tid})$

Assume that $t.\mathit{tid} = u$. Because $t \leftrightarrow \omega$,

$$t = \mathit{next}(\mathit{final}(S), u) = \mathit{next}(\mathit{final}(S.\omega), u) = \text{last}(\alpha)$$

Thus, $\text{last}(\alpha) \in T$ and we have a contradiction.

Assume that $t.\mathit{tid} \neq u$. Let $\omega' = \omega$ if t is dependent with $\text{last}(\alpha)$, and let $\omega' = \alpha$ if $t \leftrightarrow \alpha$ and t is dependent with $\mathit{next}(\mathit{final}(S.\alpha), u)$. Consider the postcondition

$$\mathit{Post}(S.t.\omega', \mathit{len}(S) + 1, u)$$

for the recursive call **Explore**$(S.t)$. By Lemma 11, $t.\omega'$ is a sequence of transitions from s in $A_{G(Pb,c)}$. Because $t \leftrightarrow \omega'$, t is the most recent transition by $t.\mathit{tid}$ that is dependent with $\mathit{next}(\mathit{final}(S.t.\omega'), u)$. Thus, by Definition 5.6 of *Post*, either $u \in \mathit{backtrack}(s)$, or $\mathit{backtrack}(s) = \mathit{enabled}(s)$ and thus $\alpha_1 \in T$. In either case, we have a contradiction.

Case 26.3. T **violates Requirement 3.**

Proceed by contradiction. Assume that there exists a nonempty sequence α of transitions from s in $A_{G(Pb,c)}$ and a transition $t \in T$ such that, if we let $u = \text{last}(\alpha).\mathit{tid}$ and let $\beta = \mathit{ext}(s, t)$:

1. $\mathit{Pb}(S.t) = \mathit{Pb}(S.\alpha_1)$
2. $\forall i \in \mathit{dom}(\alpha) : \alpha_i \notin T$
3. a transition in β is dependent with $\text{last}(\alpha)$ or with $\mathit{next}(\mathit{final}(S.\alpha), u)$

Let $n = \mathit{len}(\alpha)$, and let $\omega = \alpha_1 \ldots \alpha_{n-1}$, i.e., α with its last transition removed. Let there be no prefixes of α that also meet the criteria above, and thus

4. $\beta \leftrightarrow \omega$ and $\forall i \in \mathit{dom}(\omega) : \beta \leftrightarrow \mathit{next}(\mathit{final}(S.\omega_1 \ldots \omega_i), \omega_i.\mathit{tid})$

Assume that $\beta_1.\mathit{tid} = u$. Because $\beta \leftrightarrow \omega$,

$$\beta_1 = \mathit{next}(\mathit{final}(S), u) = \mathit{next}(\mathit{final}(S.\omega), u) = \text{last}(\alpha)$$

Thus, $\text{last}(\alpha) \in T$ and we have a contradiction.

Assume that $\beta_1.\mathit{tid} \neq u$. Let β_k be the last transition in β that is dependent with $\text{last}(\alpha)$ or with $\mathit{next}(\mathit{final}(S.\alpha), u)$. Let $\omega' = \omega$ if β_k is dependent with $\text{last}(\alpha)$, and let $\omega' = \alpha$ if $\beta \leftrightarrow \alpha$ and β_k is dependent with $\mathit{next}(\mathit{final}(S.\alpha), u)$. By Lemma 11, $\beta.\omega'$ is a sequence of transitions from s in $A_{G(Pb,c)}$. Consider the postcondition

$$\mathit{Post}(S.\beta.\omega', \mathit{len}(S) + 1, u)$$

for the recursive call **Explore**$(S.\beta_1)$. Because $\beta \leftrightarrow \omega'$, β_k is the most recent transition by $\beta_1.\mathit{tid}$ that is dependent with $\mathit{next}(\mathit{final}(S.\beta.\omega'), u)$. Because $\mathit{Pb}(S.\beta_1) = \mathit{Pb}(S.\alpha_1)$, by Definition 2.9 of the preemption bound either $\beta_1.\mathit{tid} \neq \text{last}(S).\mathit{tid}$, or S is empty. Because all transitions in β are by the same thread, β_1 is the most recent such location to β_k. Thus, by Requirement 2 of Definition 5.6 of postcondition *Post*, either $u \in \mathit{backtrack}(s)$, or $\mathit{backtrack}(s) = \mathit{enabled}(s)$ and thus $\alpha_1 \in T$. In either case, we have a contradiction.

□

Thus, if postcondition PC holds in each state s that Algorithm 8 explores with the **Backtrack** procedure from Algorithm 11, then the set of transitions Algorithm 8 explores from s is preemption-bound persistent in s.

Next, we prove that postcondition PC holds in each state s that Algorithm 8 explores. First, we prove a lemma that simplifies the inductive step. Lemma 27 differs from Lemmas 21 and 24 because it must account for preemption-bounded search's more complex postcondition in Definition 5.6.

Lemma 27. *Let* $s = \text{final}(S)$ *be a state in* $A_{R(\text{Pb},c)}$*, let* ω *and* ω' *be nonempty sequences of transitions from* s *in* $A_{G(\text{Pb},c)}$ *such that* $\text{Pb}(S.\omega'_1) \leq \text{Pb}(S.\omega_1)$*, and let* u *be a thread such that*

1. $\exists \beta : \omega.\beta \in [\omega']$ ***and*** $\beta \leftrightarrow \text{next}(\text{final}(S.\omega), u)$*, or*
2. $\exists \beta : \omega'.\beta \in [\omega]$ ***and*** $\beta \leftrightarrow \text{next}(\text{final}(S.\omega), u)$

Then, $\text{Post}(S.\omega', \text{len}(S) + 1, u) \implies \text{Post}(S.\omega, \text{len}(S), u)$.

Proof. Because $\beta \leftrightarrow \mathit{next}(\mathit{final}(S.\omega), u)$,

$$\mathit{next}(\mathit{final}(S.\omega), u) = \mathit{next}(\mathit{final}(S.\omega'), u)$$

Assume that in Definition 5.6 of postcondition $Post$, $i \leq k$ for $Post(S.\omega, len(S), u)$. Then, i and j have the same values in $Post(S.\omega', len(S), u)$ that they have in $Post(S.\omega, len(S), u)$ because $\beta \leftrightarrow next(final(S.\omega), u)$.

Assume that $i > k$ for $Post(S.\omega, len(S), u)$. Because $Pb(S.\omega'_1) \leq Pb(S.\omega_1)$, by Definition 2.9 of the preemption bound either S is empty or $\omega_1.tid \neq \text{last}(S).tid$. Thus, $j \geq k$ for $Post(S.\omega, len(S), u)$, so Definition 5.6 of $Post$ does not require any backtrack points. In either case,

$$Post(S.\omega', len(S), u) \implies Post(S.\omega, len(S), u) \tag{5.3}$$

Because Requirement 1 of Definition 5.6 of $Post$ requires that $i \leq k$ and Require-

ment 2 of Definition 5.6 of *Post* requires that $j < k$

$$Post(S.\omega', len(S) + 1, u) \implies Post(S.\omega', len(S), u)$$

Thus, by Equation 5.3,

$$Post(S.\omega', len(S) + 1, u) \implies Post(S.\omega, len(S), u)$$

□

Theorem 28. *Whenever a state* $s = \text{final}(S)$ *is backtracked during the search performed by Algorithm 8 in an acyclic state space, the postcondition* Post *for* ***Explore***(S) *is satisfied, and the set* T *of transitions explored from* s *is preemption-bound persistent in* s*.*

Proof. The proof is by induction on the order in which states are backtracked.

Base case.

Because the search is acyclic, is performed in depth-first order, and the preemption bound provides a zero-cost transition in each state, the first backtracked state must be a deadlock state in which no transition is enabled. Thus, the postcondition for the first backtracked state is

$$\forall u : Post(S, len(S), u)$$

and is directly established by Lines 4-10 in Algorithm 8.

Inductive case.

Assume that each recursive call to **Explore**$(S.t)$ satisfies its postcondition. By Lemma 26, T is preemption-bound persistent in s. Show that **Explore**(S) satisfies its postcondition for any sequence ω of transitions from s in $A_{G(Pb,c)}$ and for any

thread u.

Case 28.1. $\forall i \in \textbf{\textit{dom}}(\omega) : \omega_i \notin T$ **and** $u \in \textbf{\textit{backtrack}}(s)$**.**
Because $u \in \mathit{backtrack}(s)$, $\mathit{next}(s, u) \in T$. By Definition 2.9 of preemption-bound persistent sets, $\mathit{next}(s, u) \leftrightarrow \omega$, and thus

$$\mathit{next}(\mathit{final}(S.\omega), u) = \mathit{next}(s, u)$$

Thus, $\mathit{next}(\mathit{final}(S.\omega), u) \leftrightarrow \omega$, and therefore $\mathit{Post}(S.\omega, \mathit{len}(S), u)$ iff $\mathit{Post}(S, \mathit{len}(S), u)$. The latter is directly established by Lines 4-10 in Algorithm 8.

Case 28.2. $\forall i \in \textbf{\textit{dom}}(\omega) : \omega_i \notin T$ **and** $u \notin \textbf{\textit{backtrack}}(s)$**.**
Because $u \notin \mathit{backtrack}(s)$, $\mathit{next}(s, u) \notin T$. Let t be any transition in T, and thus $t.\mathit{tid} \neq u$. Let $\beta = t$ if $\mathit{Pb}(S.t) < \mathit{Pb}(S.\omega_1)$, and let $\beta = \mathit{ext}(s, t)$ otherwise. Consider the sequence $\omega' = \beta.\omega$. By Definition 4.10 of preemption-bound persistent sets,

1. $\mathit{Pb}(S.t) \leq \mathit{Pb}(S.\omega_1)$
2. $\beta \leftrightarrow \omega$
3. $\forall i \in \mathit{dom}(\omega) : \beta \leftrightarrow \mathit{next}(\mathit{final}(S.\omega_1 \ldots \omega_i), \omega_i.\mathit{tid})$

By Lemma 11, ω' is a sequence of transitions from s in $A_{G(Pb,c)}$. Because $\beta \leftrightarrow \omega$, $\omega.\beta \in [\omega']$. By the inductive hypothesis for the recursive call **Explore**$(S.t)$,

$$\mathit{Post}(S.\omega', \mathit{len}(S) + 1, u)$$

Assume that a transition in β is dependent with $\mathit{next}(\mathit{final}(S.\omega'), u)$. Because $\beta \leftrightarrow \omega$, the most recent dependent transition to $\mathit{next}(\mathit{final}(S.\omega'), u)$ by $\beta_1.\mathit{tid}$ must be in β. If β_1 is the most recent dependent transition, then by Requirement 1 of Definition 5.6 of *Post* either $u \in \mathit{backtrack}(s)$, or $\mathit{backtrack}(s) = \mathit{enabled}(s)$ and thus $\omega_1 \in T$. If the most recent dependent transition is another transition in β, then $\mathit{Pb}(S.t) = \mathit{Pb}(S.\omega_1)$

because otherwise β would contain only a single transition, and thus either S is empty or $\text{last}(S).tid \neq \beta_1.tid$. Thus, j must be $len(S)$ in Definition 5.6, and thus either $u \in backtrack(s)$, or $backtrack(s) = enabled(s)$ and thus $\omega_1 \in T$. In either case, we have a contradiction.

Assume that $\beta \leftrightarrow next(final(S.\omega'), u)$. Because $\beta_1.tid \neq u$, $next(final(S.\omega), u) = next(final(S.\omega'), u)$ and

$$\beta \leftrightarrow next(final(S.\omega), u)$$

Thus, by Lemma 27 where $\omega.\beta \in [\omega']$,

$$Post(S.\omega, len(S), u)$$

Case 28.3. $\exists i \in \textbf{\textit{dom}}(\omega) : \omega_i \in T$**.**
Let $\omega = \alpha.\beta.\gamma$ such that

1. $\forall i \in dom(\alpha) : \alpha_i \notin T$
2. $\beta_1 \in T$
3. $\forall i \in dom(\beta) : \beta_i.tid = \beta_1.tid$
4. if $Pb(S.\beta_1) < Pb(S.\alpha_1)$ then $len(\beta) = 1$
5. if $Pb(S.\beta_1) = Pb(S.\alpha_1)$ and γ is nonempty, then $\gamma_1.tid \neq \beta_1.tid$

Assume that α is empty. Then, $\omega_1 \in T$ and by the inductive hypothesis,

$$Post(S.\omega, len(S) + 1, u)$$

Because Requirement 1 of Definition 5.6 of *Post* requires that $i \leq k$ and Requirement 2 of Definition 5.6 of *Post* requires that $j < k$,

$$Post(S.\omega, len(S), u)$$

as required.

Assume that α is nonempty. By Requirement 1 of Definition 4.10 of preemption-bound persistent sets, $Pb(S.\beta_1) \leq Pb(S.\alpha_1)$.

Case 28.3a. γ is nonempty, or γ is empty and $\beta_1.tid \notin$ *enabled*(*final*$(S.\beta))$, or *Pb*$(S.\beta_1) <$ *Pb*$(S.\alpha_1)$.

Consider the sequence $\omega' = \beta.\alpha.\gamma$, i.e., ω with β moved to the beginning. By Requirements 2 and 3 of Definition 4.10 of preemption-bound persistent sets, $\beta \leftrightarrow \alpha$ and $\forall i \in dom(\alpha) : \beta \leftrightarrow next(final(S.\alpha_1 \ldots \alpha_i), \alpha_i.tid)$. Thus, by Definition 2.2 of a trace, $\omega' \in [\omega]$. By Lemma 12, ω' is a sequence of transitions from s in $A_{G(Pb,c)}$. By the inductive hypothesis for the recursive call **Explore**$(S.\beta_1)$,

$$Post(S.\omega', len(S) + 1, u)$$

and thus by Lemma 27 where β is empty and $\omega' \in [\omega]$,

$$Post(S.\omega, len(S), u)$$

Case 28.3b. γ is empty, $\beta_1.tid \in$ *enabled*(*final*$(S.\beta))$, *Pb*$(S.\beta_1) =$ *Pb*$(S.\alpha_1)$, and $u \in$ *backtrack*(s).

Because γ is empty, $\omega = \alpha.\beta$. Consider the sequence $\omega' = \beta$. By Requirement 3 of Definition 4.10 of preemption-bound persistent sets, $\beta \leftrightarrow \alpha$ and thus

$$\omega'.\alpha \in [\omega]$$

Because $u \in backtrack(s)$, $next(s, u) \in T$ and $next(s, u) \leftrightarrow \alpha$. If $\beta_1.tid = u$, then $next(final(S.\omega), u)$ is a transition in $ext(s, \beta_1)$ and by Requirement 3 of Definition 4.10 of preemption-bound persistent sets $next(final(S.\omega), u) \leftrightarrow \alpha$. If $\beta_1.tid \neq u$,

then $next(s, u) = next(final(S.\omega), u)$. In either case,

$$next(final(S.\omega), u) \leftrightarrow \alpha$$

Because $Pb(S.\beta_1) = Pb(S.\alpha_1)$ and all transitions in β are by the same thread and thus do not require a preemption, ω' is a sequence of transitions from s in $A_{G(Pb,c)}$. By the inductive hypothesis for the recursive call **Explore**$(S.\beta_1)$,

$$Post(S.\omega', len(S) + 1, u)$$

and thus by Lemma 27 where $\beta = \alpha$ and $\omega'.\alpha \in \omega$,

$$Post(S.\omega, len(S), u)$$

Case 28.3c. γ is empty, $\beta_1.tid \in$ *enabled*(*final*$(S.\beta)$), *Pb*$(S.\beta_1) =$ *Pb*$(S.\alpha_1)$, and $u \notin$ *backtrack*(s).

Because γ is empty, $\omega = \alpha.\beta$. Let β' be the unique, nonempty sequence of transitions from $final(S.\beta)$ such that $\beta.\beta' = ext(s, \beta_1)$. Consider the sequence $\omega' = \beta.\beta'.\alpha$. By Requirement 3 of Definition 4.10 of preemption-bound persistent sets, $\beta.\beta' \leftrightarrow \alpha$ and $\forall i \in dom(\alpha) : \beta.\beta' \leftrightarrow next(final(S.\alpha_1 \ldots \alpha_i), \alpha_i.tid)$. Thus, by Lemma 11, ω' is a sequence of transitions from s in $A_{G(Pb,c)}$. Because $\beta.\beta' \leftrightarrow \alpha$,

$$\omega.\beta' \in [\omega']$$

By the inductive hypothesis for the recursive call **Explore**$(S.\beta_1)$,

$$Post(S.\omega', len(S) + 1, u)$$

Assume that a transition in β' is dependent with $next(final(S.\omega'), u)$. Then, because

$\beta.\beta' \leftrightarrow \alpha$, the most recent dependent transition to $next(final(S.\omega'), u)$ by $\beta_1.tid$ is in β'. Thus, by Definition 5.6 of *Post*, either $u \in backtrack(s)$ or $backtrack(s) = enabled(s)$ and thus $\omega_1 \in T$. In either case, we have a contradiction.

Assume that $\beta' \leftrightarrow next(final(S.\omega'), u)$. Because $\beta_1 \in T$ and $u \notin backtrack(s)$, $\beta_1.tid \neq u$. Thus, $next(final(S.\omega), u) = next(final(S.\omega'), u)$, and

$$\beta' \leftrightarrow next(final(S.\omega), u)$$

Thus, by Lemma 27 where $\beta = \beta'$ and $\omega.\beta' \in [\omega']$,

$$Post(S.\omega, len(S), u)$$

□

Thus, Algorithm 8 explores a preemption-bound persistent set in each state with the procedures from Algorithm 11. By Theorem 13, Algorithm 8 explores all local states reachable within the bound. By Theorem 4, Algorithm 8 also explores all deadlock states reachable within the bound.

Algorithm 8 permits more partial-order reduction with preemption-bounded search than it does with context-bounded search because preemption-bounded search can more frequently guarantee that it has reached states as cheaply as possible. If the executing thread is blocked in a state s, then preemption-bounded search can explore any transition without incrementing the bound. Preemption-bounded search may therefore be able to explore the subsequent state space without leaving states unreachable within the bound that would be reachable within the bound via an unexplored, independent transition. Next, we show how to compute delta-bound persistent sets, which provide a unique cheapest transition in every state.

5.2.5 Computing Delta-Bound Persistent Sets

Algorithm 12 BPOR procedures for delta-bounded search.

```
1: procedure Initialize(S) begin
2:   Backtrack(S, len(S), u) where u is the cheapest enabled thread in final(S)
3: end
4: procedure Backtrack(S, i, u) begin
5:   if u ∈ enabled(pre(S, i)) and next(pre(S, i), u) is not a release operation then
6:     add all equal or higher priority threads than u to backtrack(pre(S, i))
7:   else
8:     backtrack(pre(S, i)) = enabled(pre(S, i))
9:   end if
10: end
```

In this section we specialize Algorithm 8 to compute delta-bound persistent sets. Algorithm 12 contains the **Initialize** and **Backtrack** procedures for delta-bounded search. Assume that Algorithm 8 calls these procedures at Lines 7 and 11, respectively. The **Initialize** procedure adds the highest priority enabled thread, which always costs zero, to the backtrack set.

The **Backtrack** procedure adds u and all higher priority threads to the backtrack set if u is enabled in $pre(S, i)$ and S_i is not a release operation. If u is disabled in $pre(S, i)$ or if S_i is a release operation, then the search conservatively adds all enabled threads to the backtrack set. To prove that Algorithm 8 computes a delta-bound persistent set in each state, we modify the postconditions from Section 5.1 for delta-bounded search.

Definition 5.7. ***PC*** **for Explore**(S) **for delta-bounded BPOR.**

$\forall u \forall \omega$: **if** $De(S.\omega) \leq c$ **then** $Post(S.\omega, len(S), u)$

Definition 5.8. ***Post***(S, k, u) **for delta-bounded BPOR.**

$\forall v$: **if** $i = max(\{i \in dom(S) \mid S_i \not\leftrightarrow next(final(S), u)$ **and** $S_i.tid = v\})$ **and** $i \leq k$ **then**

> **if** $u \in enabled(pre(S, i))$ **and** S_i is not a release operation **then**
>
> $backtrack(pre(S, i))$ contains all equal or higher priority threads than u

else *backtrack*($pre(S, i)$) = *enabled*($pre(S, i)$)

Definition 5.7 requires that *Post* hold only for sequences of transitions that are within the delta bound. If u is enabled in $pre(S, i)$ and S_i is not a release operation, then Definition 5.8 requires that u and all threads whose next transition is cheaper than u be in *backtrack*($pre(S, i)$). Otherwise, Definition 5.8 conservatively requires that all enabled threads be in the backtrack set in $pre(S, i)$.

Lemma 29. *Whenever Algorithm 8 backtracks a state* $s = \text{final}(S)$*, the set* T *of transitions explored from* s *is delta-bound persistent in* s*, provided that postcondition* PC *holds for every recursive call* ***Explore***$(S.t)$ *for all* $t \in T$*.*

Proof. Let $T = next(s, u) \mid u \in backtrack(s)$. Show that if T violates any requirement in Definition 4.11 of delta-bound persistent sets, then we have a contradiction.

Case 29.1. T **violates Requirement 1.**

Proceed by contradiction. Assume that there exist transitions $t \in T$ and $t' \notin T$ such that $t.tid, t'.tid \in enabled(s)$ and $De(S.t) \geq De(S.t')$. Because $t \in T$, $t.tid \in backtrack(s)$ and thus either Line 6 or Line 8 in Algorithm 12 must have added $t.tid$ to *backtrack*(s). If Line 6 added $t.tid$ to *backtrack*(s) then it also added $t'.tid$ to *backtrack*(s) because $De(S.t) \geq De(S.t')$. If Line 8 added $t.tid$ to *backtrack*(s) then it also added $t'.tid$ to *backtrack*(s) because $t'.tid \in enabled(s)$. In either case, $t'.tid \in backtrack(s)$ and $t' \in T$, so we have a contradiction.

Case 29.2. T **violates Requirement 2.**

Proceed by contradiction. Assume that there exists a transition $t \in T$ such that t is a release operation and a thread $u \in enabled(s)$ such that $next(s, u) \notin T$. Because t is a release operation Line 8 in Algorithm 12 must add it to *backtrack*(s). Because $u \in enabled(s)$, Line 8 also adds u to *backtrack*(s) and thus $next(s, u) \in T$ and we have a contradiction.

Case 29.3. T **violates Requirement 3.**

Proceed by contradiction. Assume that there exists a nonempty sequence α of transitions from s in $A_{G(De,c)}$ such that $\forall i \in dom(\alpha) : \alpha_i \notin T$, and a transition $t \in T$ such that

1. $De(S.t) < De(S.\alpha_1)$
2. t is not a release operation
3. t is dependent with last(α)

Let $n = len(\alpha)$ and let $\omega = \alpha_1 \ldots \alpha_{n-1}$, i.e., α with its last transition removed. Let there be no prefixes of α that also meet the criteria above, and thus

3. $t \leftrightarrow \omega$

Let $u = \text{last}(\alpha).tid$. Assume that $t.tid = u$. Because $t \leftrightarrow \omega$,

$$t = next(final(S), u) = next(final(S.\omega), u) = \text{last}(\alpha)$$

Thus, last(α) $\in T$ and we have a contradiction.

Assume that $t.tid \neq u$. Consider the postcondition

$$Post(S.t.\omega, len(S) + 1, u)$$

for the recursive call **Explore**($S.t$). By Lemma 14, $t.\omega$ is a sequence of transitions from s in $A_{G(De,c)}$. Because $t \leftrightarrow \omega$, t is the most recent transition by $t.tid$ that is dependent with $next(final(S.t.\omega), u)$. Thus, by Definition 5.8 of *Post*, $u \in backtrack(s)$ and thus a transition in α must be in T so we have a contradiction.

□

Thus, if postcondition PC holds in each state s explored by Algorithm 8 with the **Backtrack** procedure from Algorithm 12, then the set of transitions explored from

s is delta-bound persistent in s. Next, we prove that postcondition PC holds in each state s explored by Algorithm 8. First, we provide a lemma to simplify the inductive step. This lemma and its proof are very similar to Lemmas 21 and 24 and their proofs.

Lemma 30. *Let* $s = \text{final}(S)$ *be a state in* $A_{R(\text{De},c)}$*, let* ω *and* ω' *be nonempty sequences of transitions from* s *in* $A_{G(\text{De},c)}$*, and let* u *be a thread such that*

1. $\exists \beta : \omega.\beta \in [\omega']$ ***and*** $\beta \leftrightarrow \text{next}(\text{final}(S.\omega), u)$*, or*
2. $\exists \beta : \omega'.\beta \in [\omega]$ ***and*** $\beta \leftrightarrow \text{next}(\text{final}(S.\omega), u)$

Then, $\text{Post}(S.\omega', \text{len}(S) + 1, u) \implies \text{Post}(S.\omega, \text{len}(S), u)$.

Proof. Because $\beta \leftrightarrow \mathit{next}(\mathit{final}(S.\omega), u)$,

$$\mathit{next}(\mathit{final}(S.\omega), u) = \mathit{next}(\mathit{final}(S.\omega'), u)$$

Assume that for some thread v in Definition 5.8 of $\mathit{Post}(S.\omega, \mathit{len}(S), u)$, $i > k$. Then, *Post* does not require any backtrack points for v.

Assume that for some thread v in Definition 5.8 of $\mathit{Post}(S.\omega, \mathit{len}(S), u)$, $i \leq k$. Then, because $\beta \leftrightarrow \mathit{next}(\mathit{final}(S.\omega), u)$, i is the same for thread v in $\mathit{Post}(S.\omega', \mathit{len}(S), u)$. Because $i \leq \mathit{len}(S)$, the thread priorities in $\mathit{pre}(S, i)$ are the same, as well. Thus, by Definition 5.8 of *Post*,

$$\mathit{Post}(S.\omega, \mathit{len}(S), u) \text{ iff } \mathit{Post}(S.\omega', \mathit{len}(S), u) \tag{5.4}$$

Because Definition 5.8 of *Post* requires that i be less than or equal to k,

$$\mathit{Post}(S.\omega', \mathit{len}(S) + 1, u) \implies \mathit{Post}(S.\omega', \mathit{len}(S), u)$$

Thus, by Equation 5.4,

$$Post(S.\omega', len(S)+1, u) \implies Post(S.\omega, len(S), u)$$

□

Theorem 31. *Whenever a state* $s = \text{final}(S)$ *is backtracked during the search performed by Algorithm 8 in an acyclic state space, the postcondition* Post *for* ***Explore***(S) *is satisfied, and the set* T *of transitions explored from* s *is delta-bound persistent in* s*.*

Proof. The proof is by induction on the order in which states are backtracked.

Base case.

Because the search is acyclic, is performed in depth-first order, and the delta bound always provides a zero-cost transition, the first backtracked state must be a deadlock state in which no transition is enabled. Thus, the postcondition for the first backtracked state is

$$\forall u : Post(S, len(S), u)$$

and is directly established by Lines 4-10 in Algorithm 8.

Inductive case.

Assume that each call to **Explore**$(S.t)$ satisfies its postcondition. By Lemma 29, T is delta-bound persistent in s. Show that **Explore**(S) satisfies its postcondition for any sequence ω of transitions from s in $A_{G(De,c)}$ and for any thread u. If ω is empty then the postcondition is directly established by Lines 4-10 in Algorithm 8, so assume that ω is nonempty.

Case 31.1. $\forall i \in$ ***dom***$(\omega) : \omega_i \notin T$ **and** $u \in$ ***backtrack***(s)**.**

Because $u \in backtrack(s)$, $next(s, u) \in T$. Thus, by Requirement 3 of Definition 4.11

of delta-bound persistent sets, $next(s, u) \leftrightarrow \omega$, and thus

$$next(final(S.\omega), u) = next(s, u)$$

Thus, $next(final(S.\omega), u) \leftrightarrow \omega$, and therefore $Post(S.\omega, len(S), u)$ iff $Post(S, len(S), u)$. The latter is directly established by Lines 4-10 in Algorithm 8.

Case 31.2. $\forall i \in \textbf{\textit{dom}}(\omega) : \omega_i \notin T$ **and** $u \notin \textbf{\textit{backtrack}}(s)$**.**
Let t be any transition in T. Consider the sequence $\omega' = t.\omega$. By Definition 4.11 of delta-bound persistent sets, $De(S.t) < De(S.\omega_1)$ and $t \leftrightarrow \omega$. Because ω is nonempty and $\omega_1 \notin T$, by Requirement 2 of Definition 4.11 of delta-bound persistent sets, t is not a release operation. Thus, by Lemma 14, ω' is a sequence of transitions from s in $A_{G(De,c)}$. Because $t \leftrightarrow \omega$,

$$\omega.t \in [\omega']$$

By the inductive hypothesis for the recursive call $\textbf{Explore}(S.t)$,

$$Post(S.\omega', len(S) + 1, u)$$

If t is dependent with $next(final(S.\omega'), u)$, then because $t \leftrightarrow \omega$, ω'_1 must be the most recent dependent transition to $next(final(S.\omega'), u)$ by $t.tid$. Thus, by Definition 5.8 of *Post*, either $u \in backtrack(s)$ or $backtrack(s) = enabled(s)$, in which case $\omega_1 \in T$. In either case, we have a contradiction. Thus, $t \leftrightarrow next(final(S.\omega'), u)$ and $t \leftrightarrow next(final(S.\omega), u)$. Thus, by Lemma 30 where $\beta = t$ and $\omega.t \in [\omega']$,

$$Post(S.\omega, len(S), u)$$

Case 31.3. $\exists i \in \textbf{\textit{dom}}(\omega) : \omega_i \in T$**.**
Let $\omega = \alpha.t.\gamma$ such that

1. $\forall i \in dom(\alpha) : \alpha_i \notin T$

2. $t \in T$

Assume that α is empty. Then, $\omega_1 \in T$, and by the inductive hypothesis

$$Post(S.\omega, len(S) + 1, u)$$

Thus, because Definition 5.8 of *Post* requires that $i \leq k$,

$$Post(S.\omega, len(S), u)$$

as required.

Assume that α is nonempty. Consider the sequence $\omega' = t.\alpha.\gamma$, i.e., ω with t moved to the beginning. By Definition 4.11 of delta-bound persistent sets, $De(S.t) < De(S.\alpha_1)$ and $t \leftrightarrow \alpha$. Thus, by Definition 2.2 of a trace,

$$\omega' \in [\omega]$$

By Lemma 15, ω' is a sequence of transitions from s in $A_{G(De,c)}$. By the inductive hypothesis for the recursive call **Explore**$(S.t)$,

$$Post(S.\omega', len(S) + 1, u)$$

and thus by Lemma 30 where β is the empty set and $\omega' \in \omega$,

$$Post(S.\omega, len(S), u)$$

□

Thus, Algorithm 8 explores a delta-bound persistent set in each state with the

Algorithm 13 BPOR procedures for fair-bounded search.

```
1: procedure Initialize(S) begin
2:   if len(S) > MAX_DEPTH then
3:     report livelock and exit
4:   end if
5:   Backtrack(S, len(S), u) where u is a lowest cost enabled thread in final(S)
6: end
7: procedure Backtrack(S, i, u) begin
8:   if u ∈ enabled(pre(S, i)) and next(pre(S, i), u) is not a release operation then
9:     add u to backtrack(pre(S, i))
10:  else
11:    backtrack(pre(S, i)) = enabled(pre(S, i))
12:  end if
13: end
```

Backtrack and **Initialize** procedures from Algorithm 12. By Theorem 16 and Theorem 4, Algorithm 8 explores all local and deadlock states reachable within the bound. Context, preemption, and delta-bounded search permit varying degrees of partial-order reduction. These bounds do not handle cycles in the state space, however. To handle cycles in the state space, we show how to compute fair-bound persistent sets.

5.2.6 Computing Fair-Bound Persistent Sets

This section specializes Algorithm 8 to compute fair-bound persistent sets. Algorithm 13 contains **Initialize** and **Backtrack** procedures for fair-bound persistent sets. Assume that Algorithm 8 uses these procedures. The **Initialize** procedure checks whether the size of the stack exceeds a user-specified bound. This depth bound should be very large, much larger than the user expects to see in practice. If the stack depth exceeds this bound, then the search has entered a cycle in the state space despite the fairness criterion and should terminate reporting a livelock.

After checking whether the search has exceeded the stack depth, the **Initialize** procedure adds any minimum-cost enabled thread to the backtrack set. The

Backtrack procedure adds u to the backtrack set in $pre(S, i)$ if it is enabled there and if u's next transition there is not a release operation. Otherwise, Line 11 conservatively adds all enabled threads to the backtrack set.

The procedures in Algorithm 13 are similar to those in Algorithm 12 for the delta bound. These bounds are similar – both place a bound on the relative priorities of enabled transitions. As a result, both bounds must reason about transitions that enable or disable other transitions.

The fair bound differs from the delta bound, however, because the fair bound is the *maximum* difference in yield count across all transitions, not the sum of the differences as in the delta bound. Thus, fewer transitions increment the fair bound, and the fair bound can be less conservative as a result. Rather than backtrack all cheaper transitions at Line 9, the fair bound can add only u to the backtrack set. To prove that Algorithm 8 computes a fair-bound persistent set in each state, we modify the postconditions from Section 5.1 for fair-bounded search.

Definition 5.9. ***PC*** **for Explore**(S) **for fair-bounded BPOR.**
$\forall u \forall \omega$: **if** $Fb(S.\omega) \leq c$ **and** $len(S.\omega) \leq MAX\_DEPTH$ **then** $Post(S.\omega, len(S), u)$

Definition 5.10. ***Post***(S, k, u) **for fair-bounded BPOR.**
$\forall v$: **if** $i = max(\{i \in dom(S) \mid S_i \not\rightarrow next(final(S), u)$ **and** $S_i.tid = v\})$ **and** $i \leq k$ **then**

if $u \in enabled(pre(S, i))$ and S_i is not a release operation **then**
$u \in backtrack(pre(S, i))$
else $backtrack(pre(S, i)) = enabled(pre(S, i))$

Definition 5.9 requires that *Post* hold only for sequences of transitions that are within the fair bound and do not exceed the max depth. If u is enabled in $pre(S, i)$, then Definition 5.10 requires that u be in $backtrack(pre(S, i))$. Otherwise, Definition 5.10 conservatively requires that all enabled threads be in the backtrack set in $pre(S, i)$.

Lemma 32. *Whenever Algorithm 8 backtracks a state $s = \text{final}(S)$, the set T of transitions explored from s is fair-bound persistent in s, provided that postcondition* PC *holds for every recursive call* ***Explore***$(S.t)$ *for all $t \in T$.*

Proof. Let $T = \mathit{next}(s, u) \mid u \in \mathit{backtrack}(s)$. Show that if T violates any requirement in Definition 4.12 of fair-bound persistent sets, then we have a contradiction.

Case 32.1. T **violates Requirement 1.**

Proceed by contradiction. Assume that for some $t \in T$, $\mathit{Fb}(S.t) > c$. By Line 15 in Algorithm 8, the search explores only transitions that do not exceed the bound from s. Thus, we have a contradiction.

Case 32.2. T **violates Requirement 2.**

Proceed by contradiction. Assume that there exists a transition $t \in T$ such that t is a release operation and a thread $u \in \mathit{enabled}(s)$ such that $\mathit{next}(s, u) \notin T$. Because t is a release operation Line 11 in Algorithm 13 must add it to $\mathit{backtrack}(s)$. Because $u \in \mathit{enabled}(s)$, Line 11 also adds u to $\mathit{backtrack}(s)$ and thus $\mathit{next}(s, u) \in T$ and we have a contradiction.

Case 32.3. T **violates Requirement 3.**

Proceed by contradiction. Assume that there exists a nonempty sequence α of transitions from s in $A_{G(\mathit{Fb},c)}$ such that $\forall i \in \mathit{dom}(\alpha) : \alpha_i \notin T$, and a transition $t \in T$ such that

1. $\mathit{Fb}(S.t) \leq c$
2. t is not a release operation
3. t is dependent with $\text{last}(\alpha)$

Let $n = \mathit{len}(\alpha)$ and let $\omega = \alpha_1 \ldots \alpha_{n-1}$, i.e., α with its last transition removed. Let there be no prefixes of α that also meet the criteria above, and thus

3. $t \leftrightarrow \omega$

Let $u = \text{last}(\alpha).tid$. Assume that $t.tid = u$. Because $t \leftrightarrow \omega$,

$$t = \mathit{next}(\mathit{final}(S), u) = \mathit{next}(\mathit{final}(S.\omega), u) = \text{last}(\alpha)$$

Thus, $\text{last}(\alpha) \in T$ and we have a contradiction.

Assume that $t.tid \neq u$. Consider the postcondition

$$\mathit{Post}(S.t.\omega, \mathit{len}(S) + 1, u)$$

for the recursive call **Explore**$(S.t)$. By Lemma 17, $t.\omega$ is a sequence of transitions from s in $A_{G(Fb,c)}$. Because $t \leftrightarrow \omega$, t is the most recent transition by $t.tid$ that is dependent with $\mathit{next}(\mathit{final}(S.t.\omega), u)$. Thus, by Definition 5.10 of *Post*, $u \in \mathit{backtrack}(s)$ and thus a transition in α must be in T so we have a contradiction.

□

Thus, if postcondition PC holds in each state s explored by Algorithm 8 with the **Backtrack** procedure from Algorithm 13, then the set of transitions explored from s is fair-bound persistent in s. Next, we prove that postcondition PC holds in each state s explored by Algorithm 8. First, we prove a lemma to simplify the inductive step. This lemma and its proof are very similar to Lemma 30 and its proof.

Lemma 33. *Let* $s = \text{final}(S)$ *be a state in* $A_{R(\text{Fb},c)}$*, let* ω *and* ω' *be nonempty sequences of transitions from* s *in* $A_{G(\text{Fb},c)}$*, and let* u *be a thread such that*

1. $\exists \beta : \omega.\beta \in [\omega']$ ***and*** $\beta \leftrightarrow \text{next}(\text{final}(S.\omega), u)$*, or*
2. $\exists \beta : \omega'.\beta \in [\omega]$ ***and*** $\beta \leftrightarrow \text{next}(\text{final}(S.\omega), u)$

Then, $\text{Post}(S.\omega', \text{len}(S) + 1, u) \implies \text{Post}(S.\omega, \text{len}(S), u)$.

Proof. Because $\beta \leftrightarrow \mathit{next}(\mathit{final}(S.\omega), u)$,

$$\mathit{next}(\mathit{final}(S.\omega), u) = \mathit{next}(\mathit{final}(S.\omega'), u)$$

Assume that for some thread v in Definition 5.10 of $\mathit{Post}(S.\omega, \mathit{len}(S), u)$, $i > k$. Then, *Post* does not require any backtrack points for v.

Assume that for some thread v in Definition 5.10 of $\mathit{Post}(S.\omega, \mathit{len}(S), u)$, $i \leq k$. Then, because $\beta \leftrightarrow \mathit{next}(\mathit{final}(S.\omega), u)$, i is the same for thread v in $\mathit{Post}(S.\omega', \mathit{len}(S), u)$. Because $i \leq \mathit{len}(S)$, the yield counts for all threads are the same in $\mathit{pre}(S, i)$, as well. Thus, by Definition 5.10 of *Post*,

$$\mathit{Post}(S.\omega, \mathit{len}(S), u) \text{ iff } \mathit{Post}(S.\omega', \mathit{len}(S), u) \tag{5.5}$$

Because Definition 5.10 of *Post* requires that i be less than or equal to k,

$$\mathit{Post}(S.\omega', \mathit{len}(S) + 1, u) \implies \mathit{Post}(S.\omega', \mathit{len}(S), u)$$

Thus, by Equation 5.5,

$$\mathit{Post}(S.\omega', \mathit{len}(S) + 1, u) \implies \mathit{Post}(S.\omega, \mathit{len}(S), u)$$

□

Theorem 34. *Whenever a state $s = \text{final}(S)$ is backtracked during the search performed by Algorithm 8, the postcondition* Post *for* ***Explore***(S) *is satisfied, and the set T of transitions explored from s is fair-bound persistent in s.*

Proof. The proof is by induction on the order in which states are backtracked.

Base case.

If the stack depth exceeds *MAX_DEPTH*, then the search terminates and reports

a livelock. Thus, the state space that the search may explore without reporting a livelock is a subset of the cyclic state space. Assume that the test does not contain a livelock. Because the search is performed in depth-first order, and the fair bound always provides a zero-cost transition, the first backtracked state must be a deadlock state in which no transition is enabled. Thus, the postcondition for the first backtracked state is

$$\forall u : \mathit{Post}(S, \mathit{len}(S), u)$$

and is directly established by Lines 4-10 in Algorithm 8.

Inductive case.

Assume that each call to **Explore**$(S.t)$ satisfies its postcondition. By Lemma 32, T is fair-bound persistent in s. Show that **Explore**(S) satisfies its postcondition for any sequence ω of transitions from s in $A_{G(Fb,c)}$ and for any thread u. If ω is empty then the postcondition is directly established by Lines 4-10 in Algorithm 8, so assume that ω is nonempty.

Case 34.1. $\forall i \in \mathbf{dom}(\omega) : \omega_i \notin T$ **and** $u \in \mathbf{backtrack}(s)$**.**

Because $u \in \mathit{backtrack}(s)$, $\mathit{next}(s, u) \in T$. Thus, by Requirement 3 of Definition 4.12 of fair-bound persistent sets, $\mathit{next}(s, u) \leftrightarrow \omega$, and thus

$$\mathit{next}(\mathit{final}(S.\omega), u) = \mathit{next}(s, u)$$

Thus, $\mathit{next}(\mathit{final}(S.\omega), u) \leftrightarrow \omega$, and therefore $\mathit{Post}(S.\omega, \mathit{len}(S), u)$ iff $\mathit{Post}(S, \mathit{len}(S), u)$. The latter is directly established by Lines 4-10 in Algorithm 8.

Case 34.2. $\forall i \in \mathbf{dom}(\omega) : \omega_i \notin T$ **and** $u \notin \mathbf{backtrack}(s)$**.**

Let t be any transition in T. Consider the sequence $\omega' = t.\omega$. By Definition 4.12 of fair-bound persistent sets, $\mathit{Fb}(S.t) \leq c$ and $t \leftrightarrow \omega$. Because ω is nonempty and $\omega_1 \notin T$, by Requirement 2 of Definition 4.12 of fair-bound persistent sets, t is not

a release operation. Thus, by Lemma 17, ω' is a sequence of transitions from s in $A_{G(Fb,c)}$. Because $t \leftrightarrow \omega$,

$$\omega.t \in [\omega']$$

By the inductive hypothesis for the recursive call **Explore**$(S.t)$,

$$\mathit{Post}(S.\omega', \mathit{len}(S) + 1, u)$$

If t is dependent with $\mathit{next}(\mathit{final}(S.\omega'), u)$, then because $t \leftrightarrow \omega$, ω'_1 must be the most recent dependent transition to $\mathit{next}(\mathit{final}(S.\omega'), u)$ by $t.\mathit{tid}$. Thus, by Definition 5.10 of *Post*, either $u \in \mathit{backtrack}(s)$ or $\mathit{backtrack}(s) = \mathit{enabled}(s)$, in which case $\omega_1 \in T$. In either case, we have a contradiction. Thus, $t \leftrightarrow \mathit{next}(\mathit{final}(S.\omega'), u)$ and $t \leftrightarrow \mathit{next}(\mathit{final}(S.\omega), u)$. Thus, by Lemma 33 where $\beta = t$ and $\omega.t \in [\omega']$,

$$\mathit{Post}(S.\omega, \mathit{len}(S), u)$$

Case 34.3. $\exists i \in \mathbf{dom}(\omega) : \omega_i \in T$**.**
Let $\omega = \alpha.t.\gamma$ such that

1. $\forall i \in \mathit{dom}(\alpha) : \alpha_i \notin T$
2. $t \in T$

Assume that α is empty. Then, $\omega_1 \in T$, and by the inductive hypothesis

$$\mathit{Post}(S.\omega, \mathit{len}(S) + 1, u)$$

Thus, because Definition 5.10 of *Post* requires that $i \leq k$,

$$\mathit{Post}(S.\omega, \mathit{len}(S), u)$$

as required.

Assume that α is nonempty. Consider the sequence $\omega' = t.\alpha.\gamma$, i.e., ω with t moved to the beginning. By Definition 4.12 of fair-bound persistent sets, $Fb(S.t) \leq c$ and $t \leftrightarrow \alpha$. Thus, by Definition 2.2 of a trace,

$$\omega' \in [\omega]$$

By Lemma 18, ω' is a sequence of transitions from s in $A_{G(Fb,c)}$. By the inductive hypothesis for the recursive call **Explore**$(S.t)$,

$$Post(S.\omega', len(S) + 1, u)$$

and thus by Lemma 33 where β is empty and $\omega' \in [\omega]$,

$$Post(S.\omega, len(S), u)$$

□

Thus, Algorithm 8 explores a fair-bound persistent set in each state with the **Backtrack** and **Initialize** procedures in Algorithm 13. By Theorem 19 and Theorem 4, Algorithm 8 explores all local and deadlock states reachable within the bound.

Fair-bounded search permits more partial-order reduction than depth, context, preemption, or delta-bounded search do, and it prunes cycles in the state space. The fair bound does not, however, otherwise limit the state space. The fair bound must be combined with other bounds to provide incremental guarantees. The next section optimizes Algorithm 8 to further reduce the bounded state space. We then add sleep sets to the search and while maintaining bounded coverage. Finally, we combine multiple bound functions so that fair-bounded search may provide better incremental guarantees.

Chapter 6

Optimizations

This chapter optimizes bounded partial-order reduction (BPOR), introduced in Chapter 5, to provide additional state space reduction while maintaining bounded coverage. We first review several optimizations from prior work and show how they apply to bounded search. We then introduce optimizations designed specifically to reduce the overhead of bounded search. For each optimization, we discuss how it changes the algorithms in Chapter 5, and how it affects the correctness proofs for those algorithms. We implement the following optimizations to further reduce the state space with BPOR:

1. **Transitive Reduction**: Add backtrack points only for dependences that are in the transitive reduction of the partial order on dependent transitions
2. **Alternative Thread**: When a thread is disabled, rather than backtrack all threads, backtrack a thread that performs a dependent transition
3. **Release**: Do not backtrack all threads prior to release operations; instead, backtrack prior to the most recent acquire
4. **Bound**: Explore conservative backtrack points only if the search exceeds the bound in the subsequent state space

5. **Sleep sets**: Use a modified version of the sleep sets algorithm [Godefroid, 1990] to reduce the bounded state space

The first three optimizations were included in the original DPOR algorithm [Flanagan and Godefroid, 2005]. We show how they interact with bounded search. The bound optimization is specific to the bound. This optimization is particularly effective as the bound approaches saturation, where the entire state space is reachable within the bounded value. Although all of our tests use a conservative form of sleep sets with bounded search, the sleep sets optimization makes sleep sets less conservative for bounded search. The degree to which these optimizations are effective varies with the test, and also with the bound. We show in Chapter 8 how each optimization reduces the state space individually, and find that the optimizations are complementary. Applying all of them reduces the state space more than applying any one optimization individually.

6.1 Transitive Reduction Optimization

BPOR may introduce unnecessary backtrack points if it backtracks dependences that are not part of the *transitive reduction* on the program's partial-order. The transitive reduction of a partial-order is the minimal graph that represents the same partial-order. We modify BPOR to backtrack only dependences that are part of the transitive reduction. Flanagan and Godefroid implement a similar optimization for DPOR [Flanagan and Godefroid, 2005].

The non-minimal graph in Figure 6.1 shows a simple program containing two threads, with two pairs of dependent transitions. Thread u's write to x and Thread v's read of x are dependent, yet this dependence is not part of the transitive reduction. Even if these transitions were not dependent with one another, Thread u's write to x would still happen before Thread v's read of x because Thread u's end operation is dependent with Thread v's join operation. The second picture in

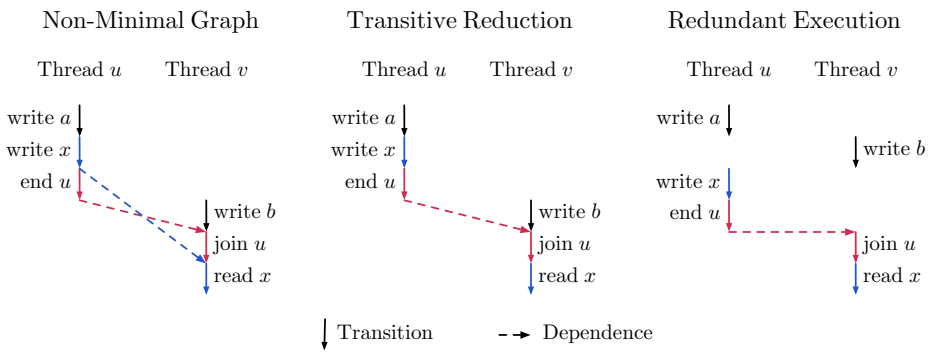


Figure 6.1: A non-minimal graph, its transitive reduction, and the redundant execution that the transitive reduction optimization prunes.

Figure 6.1 shows the transitive reduction for the same graph. The partial-order for this graph is the same; a transition t happens before a transition t' in the graph on the left if and only if t also happens before t' in the graph on the right. To detect edges that are part of the transitive reduction we introduce the following relations:

Definition 6.1. Transitive dependence [Flanagan and Godefroid, 2005].
The relation $\rightarrow_S$ is the smallest relation on $dom(S)$ such that

1. if $i \leq j$ and t_i is dependent with t_j, then $i \rightarrow_S j$
2. $\rightarrow_S$ is transitively closed.

Definition 6.2. Transitive thread dependence [Flanagan and Godefroid, 2005].
Let S be a sequence of transitions. The relation $i \rightarrow_S u$ holds for $i \in dom(S)$ and thread u if and only if there exists $j \in dom(S)$ such that $i < j$, $i \rightarrow_S j$, and $S_j \not\leftrightarrow next(final(S), u)$.

Intuitively, $i \rightarrow_S u$ if and only if t_i would be transitively dependent with $next(final(S), u)$ if it were to execute from $final(S)$. We define this relation with respect to u's next transition from $final(S)$, $next(final(S), u)$, because $next(final(S), u)$ may or may not

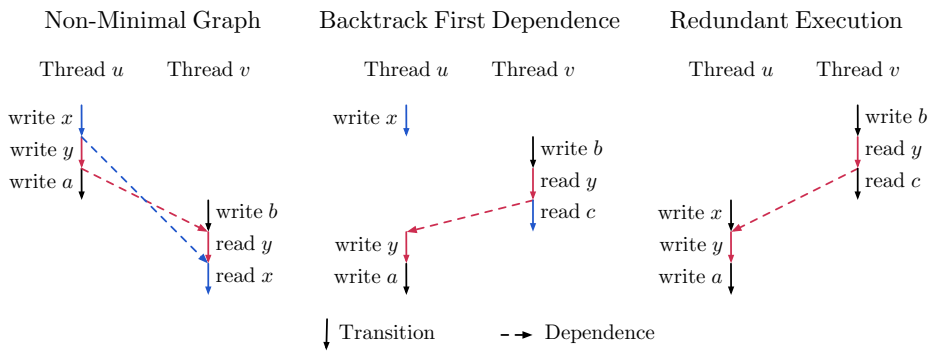


Figure 6.2: The transitive reduction optimization prunes the execution on the right because it does not place a backtrack point prior to Thread u's write to y.

be enabled in $\mathit{final}(S)$. Whether or not $\mathit{next}(\mathit{final}(S), u)$ is enabled in $\mathit{final}(S)$, the algorithm must reason about its transitive dependences.

The transitive reduction optimization is beneficial because it may prevent the search from exploring redundant executions. For example, the picture on the right in Figure 6.1 shows the execution that results if the search explores Thread v prior to Thread u's write to x. Thread v blocks because its join operation must wait for Thread u to terminate. The partial-order that results is the same as the partial-order for the figure on the left, and thus the execution on the right is redundant. The bulk of the benefit from the transitive reduction optimization is due to scenarios similar to the one in Figure 6.1.

Figure 6.2 illustrates a different scenario in which the transitive reduction optimization is beneficial. The execution on the left shows the non-minimal graph for a short program that contains no blocking operations. Because none of the threads block, the order of the dependent transitions can be reversed. In the execution in the center, the search reorders Thread u's write to y and Thread v's read of y. As a result, Thread v observes a new value for y and its behavior changes in response. Rather than write x, Thread v reads c. This transition is not dependent

Algorithm 14 Backtrack (Context-bounded BPOR, transitive reduction).

```
1: procedure Backtrack(S, i, u) begin
2:   if i ↛_S u then
3:     if i = 0 or u ≠ S_{i−1}.tid or u ∉ enabled(pre(S, i)) then
4:       backtrack(pre(S, i)) = enabled(pre(S, i))
5:     else
6:       add u to backtrack(pre(S, i))
7:     end if
8:   end if
9: end
```

with any transition by Thread u. Because the search already added a backtrack point for Thread v prior to Thread u's write to x, however, the search performs the redundant execution on the right. If Thread v's behavior did not change, however, then the search would still have the opportunity to insert a backtrack point prior to Thread u's write to x after the execution in the center.

To prevent the search from exploring unnecessary transitions like the ones in Figures 6.1 and 6.2, we add backtrack points only for dependences that are part of the transitive reduction. This optimization applies to both DPOR and BPOR. Flanagan and Godefroid include a similar optimization in the original DPOR algorithm [Flanagan and Godefroid, 2005]. Because they add backtrack points for only the most recent dependent transition, however, they use the transitive reduction slightly differently. At Line 6 in Algorithm 7, they choose the most recent dependent transition that is in the transitive reduction. Because we add the most recent transition by each thread, we incorporate the transitive reduction check later, during the **Backtrack** procedure.

Algorithm 14 shows the modified **Backtrack** procedure for context-bounded BPOR. Line 2 checks whether the dependence being backtracked is in the transitive reduction before adding a backtrack point. Thus, the postcondition for context-bounded search changes.

Algorithm 15 Backtrack (Preemption-bounded BPOR, transitive reduction).

```
1: procedure Backtrack(S, i, u) begin
2:   if i ↛_S u then
3:     AddBacktrackPoint(S, i, u)
4:     if j = max({j ∈ dom(S) | j = 0 or S_{j−1}.tid ≠ S_j.tid and j < i}) then
5:       AddBacktrackPoint(S, j, u)
6:     end if
7:   end if
8: end
```

Definition 6.3. ***Post***(S, k, u) (Context-bounded BPOR, transitive reduction).
$\forall v$: **if** $i = max(\{i \in dom(S) \mid S_i \not\leftrightarrow next(final(S), u)$ **and** $S_i.tid = v\})$ **and** $i \leq k$ **and** $i \not\rightarrow_S u$ **then**

$$backtrack(pre(S, i)) = enabled(pre(S, i))$$

Definition 6.3 is identical to Definition 5.4, except that Definition 6.3 checks both that $i \leq k$, *and* that $i \not\rightarrow_S u$.

Algorithm 15 provides the **Backtrack** procedure for preemption-bounded BPOR with the transitive reduction optimization. At Line 2, Algorithm 15 checks whether the dependence being backtracked is in the transitive reduction. If the dependence is in the transitive reduction, then Algorithm 15 inserts two backtrack points: one prior to the most recent dependent transition and one prior to the most recent cheaper transition. Otherwise, Algorithm 15 inserts neither backtrack point. Definition 6.4 shows the modified postcondition for preemption-bounded BPOR. Both backtrack points require that $i \not\rightarrow_S u$.

Definition 6.4. ***Post***(S, k, u) (Preemption-bounded BPOR, transitive reduction).
$\forall v$: **if** $i = max(\{i \in dom(S) \mid S_i \not\leftrightarrow next(final(S), u)$ **and** $S_i.tid = v\})$ **and** $i \not\rightarrow_S u$ **then**

1. **if** $i \leq k$ **then**

   if $u \in enabled(pre(S, i))$ **then** $u \in backtrack(pre(S, i))$

else $\mathit{backtrack}(\mathit{pre}(S,i)) = \mathit{enabled}(\mathit{pre}(S,i))$

2. **if** $j = \mathit{max}(\{j \in \mathit{dom}(S) \mid j = 0$ **or** $S_{j-1}.\mathit{tid} \neq S_j.\mathit{tid}$ **and** $j < i\})$ **and** $j < k$ **then**
   if $u \in \mathit{enabled}(\mathit{pre}(S,j))$ **then** $u \in \mathit{backtrack}(\mathit{pre}(S,j))$
   else $\mathit{backtrack}(\mathit{pre}(S,j)) = \mathit{enabled}(\mathit{pre}(S,j))$

Next, we describe how the transitive reduction optimization modifies the correctness proofs for preemption-bounded BPOR. Many parts of these proofs remain the same because the transitive reduction does not affect them. Thus, we highlight only the parts of each proof that are explicitly affected by the transitive reduction optimization. Because this optimization changes the postcondition *Post*, we discuss each use of the postcondition below. We do not describe changes to the context-bounded BPOR proofs because they are similar to yet simpler than the changes to the preemption-bounded BPOR proofs.

Cases 26.2 and 26.3 of Lemma 26 change slightly to accommodate the transitive reduction optimization. In Case 26.2, when we consider the postcondition

$$\mathit{Post}(S.t.\omega', \mathit{len}(S)+1, u)$$

for the recursive call **Explore**$(S.t)$, we must show that the dependence between t and $\mathit{next}(\mathit{final}(S.t.\omega'), u)$ is in the transitive reduction. Let $i = \mathit{len}(S)$. We are given that $t \leftrightarrow \omega'$. Thus, there cannot exist $j \in \mathit{dom}(S.t.\omega')$ such that $i < j$ and $i \rightarrow_{S.t.\omega'} j$. Thus, by Definition 6.2 of transitive dependence, $i \not\rightarrow_{S.t.\omega'} u$, so the dependence between t and $\mathit{next}(\mathit{final}(S.t.\omega'), u)$ is in the transitive reduction. A similar argument applies to postcondition $\mathit{Post}(S.t.\omega, \mathit{len}(S)+1, u)$ in Case 23.2 of Lemma 23 for context-bounded BPOR.

In Case 26.3, when we consider the postcondition

$$Post(S.\beta.\omega', len(S) + 1, u)$$

for the recursive call **Explore**$(S.\beta_1)$, we must show that the dependence between β_k and $next(final(S.\beta.\omega'), u)$ is in the transitive reduction. Let $i = len(S) + k$. We are given that $\beta \leftrightarrow \omega'$, and that β_k is the last transition in β that is dependent with $next(final(S.\beta.\omega'), u)$. Thus, if there exists $j \in dom(S.\beta.\omega')$ such that $i < j$ and $i \rightarrow_{S.\beta.\omega'} j$, then j must be the index of a transition in β, and the transition at index j therefore cannot be dependent with $next(final(S.\beta.\omega'), u)$. Thus, by Definition 6.2 of transitive dependence, $i \not\rightarrow_{S.\beta.\omega'} u$, and the dependence between β_k and $next(final(S.\beta.\omega'), u)$ is in the transitive reduction.

Consider Lemma 27. We must show that for any $i \leq len(S)$, if $i \not\rightarrow_{S.\omega} u$, then $i \not\rightarrow_{S.\omega'} u$, because then any backtrack point that $Post(S.\omega, len(S), u)$ requires, $Post(S.\omega', len(S), u)$ also requires. Assume that for some $i \leq len(S)$, $i \not\rightarrow_{S.\omega} u$. Because $\beta \leftrightarrow next(final(S.\omega), u)$ and either $\omega.\beta \in [\omega']$ or $\omega'.\beta \in [\omega]$, $i \not\rightarrow_{S.\omega'} u$. A similar argument applies for context-bounded BPOR in Lemma 24 with the transitive reduction optimization.

Although we do not provide the entire modified version of each proof with this optimization, the examples we provide demonstrate how the proofs change to accommodate the optimization. Next, we introduce an optimization that limits the number of conservative backtrack points that the search must include when a thread is disabled. We show how the optimization affects the **Backtrack** procedure and provide intuition for how it modifies the correctness proofs.

6.2 Alternative Thread Optimization

When the **Backtrack** procedures backtrack threads that are disabled, they conservatively schedule *all* threads. Flanagan and Godefroid observe that when a thread u is blocked, a single transition, rather than all enabled transitions, often suffices for maintaining coverage [Flanagan and Godefroid, 2005]. We apply this optimization to BPOR. First, we define a set of threads in each state that are sufficient to backtrack thread u.

The helper function $E(S, i, u)$ returns a set of threads whose next transition from $pre(S, i)$ backtracks a dependence between S_i and $next(final(S), u)$. This set includes u, if it is enabled in $pre(S, i)$, and any threads whose next transition from $pre(S, i)$ is transitively dependent with $next(final(S), u)$ according to Definition 6.2. Exploring any of these transitions from $pre(S, i)$ allows $next(final(S), u)$ to execute prior to S_i, if possible, changing the partial order. We define $E(S, i, u)$ as follows:

Definition 6.5. $\boldsymbol{E}(S, i, u)$ [Flanagan and Godefroid, 2005].
$E(S, i, u) = \{v \in enabled(pre(S, i)) \mid u = v$ **or**

$$\exists j \in dom(S) : i < j, v = S_j.tid \textbf{ and } j \rightarrow_S u$$

We modify the **Backtrack** procedure for preemption-bounded BPOR to ensure that the thread in $E(S, i, u)$ with the cheapest next transition is in $backtrack(pre(S, i))$, if $E(S, i, u)$ is nonempty. This optimization does not apply to context-bounded BPOR because in each state, context-bounded BPOR must explore *only* the executing thread or it must explore all threads. No alternative to the executing thread will suffice because any other thread may leave otherwise reachable local states unreachable within the bound.

Algorithm 16 shows the **Backtrack** procedure for preemption-bounded BPOR with the alternative thread optimization. Lines 2-6 in Algorithm 16 select the thread in $E(S, i, u)$ with the lowest cost next transition to backtrack, if $E(S, i, u)$

Algorithm 16 Backtrack (Preemption-bounded BPOR, alternative thread).

```
1: procedure Backtrack(S, i, u) begin
2:   if E(S, i, u) ≠ ∅ then
3:     Let v be the thread in E(S, i, u) with the minimum cost from pre(S, i)
4:   else
5:     Let v = u
6:   end if
7:   AddBacktrackPoint(S, i, v)
8:   if j = max({j ∈ dom(S) | j = 0 or S_{j−1}.tid ≠ S_j.tid and j ≤ i}) then
9:     AddBacktrackPoint(S, j, v)
10:  end if
11: end
```

is nonempty. Otherwise, if $E(S, i, u)$ is empty, then u is disabled and no suitable alternative can be found, so Algorithm 16 backtracks u. Because u is disabled, **AddBacktrackPoint** backtracks all threads in $pre(S, i)$. This alternative thread applies to both of the two backtrack points that **Backtrack** creates.

The postcondition for preemption-bounded BPOR must reflect this change. Definition 6.6 checks whether $E(S, i, u)$ is nonempty, and if so requires that at least one of the transitions in $E(S, i, u)$ be in $backtrack(pre(S, i))$. The same check applies to j, the location of the most recent cheaper transition.

Definition 6.6. ***Post***(S, k, u) (Preemption-bounded BPOR, alternative thread).
$\forall v$: **if** $i = max(\{i \in dom(S) \mid S_i \not\leadsto next(final(S), u)$ **and** $S_i.tid = v\})$ **then**

1. **if** $i \leq k$ **then**
   if $E(S, i, u) \neq \emptyset$ **then** $backtrack(pre(S, i)) \cap E(S, i, u) \neq \emptyset$
   else $backtrack(pre(S, i)) = enabled(pre(S, i))$

2. **if** $j = max(\{j \in dom(S) \mid j = 0$ **or** $S_{j-1}.tid \neq S_j.tid$ **and** $j < i\})$ **and** $j < k$ **then**
   if $E(S, i, u) \neq \emptyset$ **then** $backtrack(pre(S, j)) \cap E(S, i, u) \neq \emptyset$
   else $backtrack(pre(S, j)) = enabled(pre(S, j))$

The alternative thread optimization affects the correctness proofs for DPOR and BPOR significantly, but the change that the optimization requires is similar across all of the proofs. We provide an example for preemption-bounded BPOR, but do not provide the detailed proof for each bound function or for each case where the alternative thread optimization affects the proof, because these cases are all similar to one another.

Consider Case 26.2 of Lemma 26. Because $Post(S.t.\omega', len(S)+1, u)$ holds and t is the most recent dependent transition to $next(final(S.t.\omega'), u)$, the proof for Lemma 26 concludes that either $u \in backtrack(s)$ or $backtrack(s) = enabled(s)$, in which case $\alpha_1 \in T$. Either of these cases leads to a contradiction. With the alternative thread optimization, however, it may be the case that u is disabled in s and some *other* thread is in $backtrack(s)$, rather than all enabled threads. By Definition 6.5 of $E(S.t.\omega', len(S), u)$, however, for any thread v in $E(S.t.\omega', len(S), u)$ there must exist $j \in dom(S.t.\omega')$ such that $len(S) < j$, $v = S_j.tid$, and $j \rightarrow_{S.t.\omega'} u$. Any such j must therefore be in ω', and because none of the transitions in ω' are in $backtrack(s)$, we still have a contradiction.

In Case 26.3 of Lemma 26, a similar argument applies. Rather than a single transition t prior to ω', there exists a series of transitions β all by $t.tid$ that execute prior to ω'. The most recent dependent transition to $next(final(S.\beta.\omega'), u)$ is β_k. Again, any thread in $E(S.\beta.\omega', len(S)+k, u)$, if it is nonempty, must either be u or be the thread that performs a transition in α. In either case, we have a contradiction. A similar argument applies for each use of the postcondition.

6.3 Release Optimization

Flanagan and Godefroid optimize backtrack points prior to release operations by placing them prior to the corresponding acquire operation, instead [Flanagan and Godefroid, 2005]. A dependent acquire operation will never be enabled until the

Algorithm 17 Backtrack (Context-bounded BPOR, release optimization).

```
1: procedure Backtrack(S, i, u) begin
2:   if S_i is a release and next(final(S), u) is an acquire then
3:     Backtrack(S, GetAcquire(i), u)
4:   else
5:     if i = 0 or u ≠ S_{i−1}.tid or u ∉ enabled(pre(S, i)) then
6:       backtrack(pre(S, i)) = enabled(pre(S, i))
7:     else
8:       add u to backtrack(pre(S, i))
9:     end if
10:  end if
11: end
```

release operation completes, so backtracking an acquire prior to a release does not explore any new states. Assume that the most recent dependent transition to an acquire of lock m is a release of m by a thread u. Immediately prior to the release, u still holds the lock. Thus, placing a backtrack point prior to u's release operation will not reverse the order of the dependent transitions; it will lead to a redundant, wasted execution. Instead, Flanagan and Godefroid place a backtrack point prior to the most recent dependent transition *that may be co-enabled*. For a lock release operation, this transition is the most recent acquire.

For context-bounded BPOR, this optimization requires only a small change, as shown in Algorithm 17. We assume that a lock acquire and a lock release to the same variable may not be co-enabled, and that a thread begin and a thread fork operation may not be co-enabled. There cannot exist *any* dependent transition with which a thread begin operation may be co-enabled, so we do not insert any backtrack points for this operation when the release optimization is enabled.

For a release operation, Algorithm 17 must backtrack prior to the most recent dependent, co-enabled transition. At Line 2 in Algorithm 17, if S_i is a release and $\mathit{next}(\mathit{final}(S), u)$ is an acquire of the same variable, then Line 3 backtracks prior to the appropriate acquire operation. The procedure **GetAcquire**(i) returns the index

of the most recent dependent acquire operation to S_i. If the programming language supports nested acquire operations by a single thread, then **GetAcquire**(i) returns the index of the acquire at the same level of nesting.

Definition 6.7 provides the postcondition for context-bounded BPOR with the release optimization. This postcondition requires that i be the index of the most recent dependent transition *that may be co-enabled* with $next(final(S), u)$.

Definition 6.7. ***Post***(S, k, u) (Context-bounded BPOR, release optimization).
$\forall v$: **if** $i = max(\{i \in dom(S) \mid S_i \not\leftrightarrow next(final(S), u), S_i$ and $next(final(S), u)$ may be co-enabled, **and** $S_i.tid = v\})$ **then**
if $i \leq k$ **then** $backtrack(pre(S, i)) = enabled(pre(S, i))$

Preemption-bounded BPOR cannot exploit the release optimization in the same way that context-bounded BPOR does. Recall from Definition 4.10 of preemption-bound persistent sets that the transitions in the preemption-bound persistent set must be independent not only with all sequences of transitions α such that $\forall i \in dom(\alpha) : \alpha_i \notin T$, but also with $next(final(S.\alpha), \text{last}(\alpha).tid)$ for all such α. Due to this requirement, preemption-bounded BPOR must sometimes place backtrack points prior to release operations.

Figure 6.3 illustrates a scenario where preemption-bounded BPOR requires a backtrack point prior to a release operation. Execution 1 requires two preemptions, shown with pink, horizontal lines in Figure 6.3. The first preemption allows Thread v to write x before Thread u does. The second preemption allows Thread u to write x before Thread v writes x a second time. Execution 2 has precisely the same partial-order that Execution 1 has, yet Execution 2 requires only one preemption. By scheduling Thread v prior to Thread u's release operation, the search ensures that Thread v's acquire operation blocks. Thus, the search can perform a context switch for free, because the executing thread is disabled. As a result, preemption-bounded BPOR cannot optimize away all backtrack points prior to release operations.

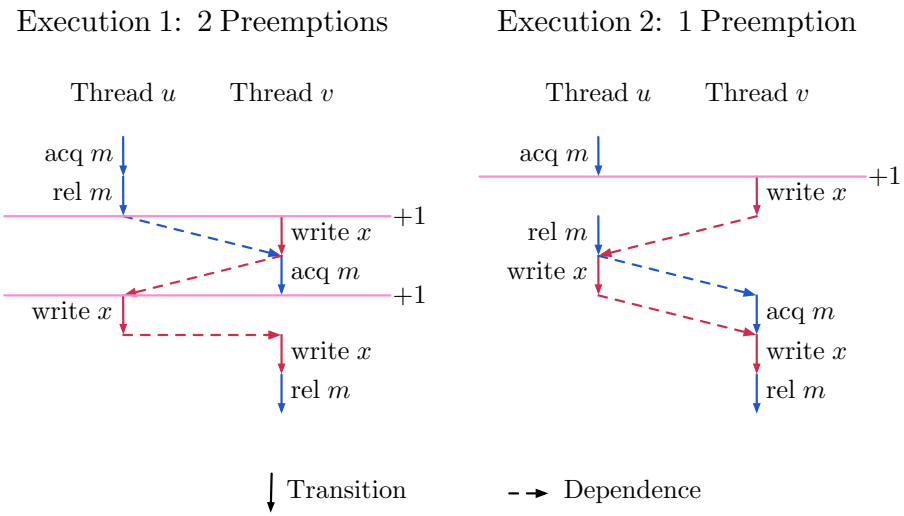


Figure 6.3: Scenario that requires a preemption prior to a release operation for preemption-bounded search.

Preemption-bounded BPOR need not backtrack prior to *all* release operations, however, and it need not backtrack *all* enabled threads prior to a release operation as it would with the algorithms in Chapter 5. BPOR must backtrack prior to a release operation only if that backtrack point allows a preemptive context switch that the search explored to occur without a preemption. By exploiting dynamic information about the subsequent state space, the search could backtrack prior to release operations less frequently while preserving bounded coverage. A similar optimization could apply to delta and fair-bounded search, which also must backtrack release operations. We leave these optimizations for future work.

6.4 Bound Optimization

The bound optimization ensures that with a sufficiently high bound, BPOR behaves exactly like DPOR. This optimization ensures that BPOR does not perform work specific to the bound when the bound has not been exceeded. With a sufficiently

high bound, BPOR never excludes a transition because it exceeds the bound. In this scenario, BPOR need not add conservative backtrack points to account for the bound. This optimization applies to all bound functions, including the depth, context, and preemption bound.

In each state s, the bound optimization keeps track of whether the search has excluded a transition because it exceeds the bound in the state space reachable from s. If the search never excludes a transition because it exceeds the bound in the state space reachable from s, then the search explores only those backtrack points that DPOR requires from s. Otherwise, the search explores all backtrack points that BPOR requires from s. This optimization ensures that BPOR never explores bound-specific backtrack points until the search exceeds the bound. If the search does not exceed the bound, then all states are reachable within the bound from s, so backtrack points that compensate for the bound are unnecessary.

Algorithm 18 contains a modified version of the **Explore** procedure for BPOR that uses the bound optimization. This procedure keeps track of a variable, *boundExceeded*, that indicates whether any transition reachable from $\mathit{final}(S)$ exceeds the bound. **Explore** returns this result as a boolean value. Line 4 initializes *boundExceeded* to false. Lines 5-11 in Algorithm 18 are unchanged from Algorithm 8, though we do modify the **Backtrack** procedure, as discussed below. Lines 14-21 iterate through each thread u in $\mathit{backtrack}(s)$. If $\mathit{next}(\mathit{final}(S), u)$ does not exceed the bound, then Line 17 explores it, and updates *boundExceeded* with the result. If $\mathit{next}(\mathit{final}(S), u)$ exceeds the bound, then Line 19 sets *boundExceeded* to true.

After exploring all transitions in $\mathit{backtrack}(s)$, Line 22 checks whether *boundExceeded* is set. If *boundExceeded* is set, Lines 23-28 explore all threads in $\mathit{pending}(s)$. The **Backtrack** method must add all threads that are not required by DPOR to $\mathit{pending}(s)$ to differentiate them from backtrack points required by DPOR. Algorithm 19 shows this modified **Backtrack** method for preemption-bounded search.

Algorithm 18 BPOR with bound optimization.

```
1: Initially, Explore(ϵ) from s_0
2: procedure bool Explore(S) begin
3:   Let s = final(S)
4:   boundExceeded = false
     # Add backtracking points for each thread's next transition.
5:   for all u ∈ Tid do
6:     for all v ∈ Tid | v ≠ u do
         # Find most recent dependent transition.
7:       if ∃i = max({i ∈ dom(S) | (S_i, next(s,u)) ∈ D and S_i.tid = v}) then
8:         Backtrack(S, i, u)
9:       end if
10:    end for
11:  end for
     # Continue the search by exploring successor states.
12:  Initialize(S)
13:  Let visited = ∅
14:  while ∃u ∈ (enabled(s) ∩ backtrack(s) \ visited) do
15:    add u to visited
16:    if Bv(S.next(s,u)) ≤ c then
17:      boundExceeded |= Explore(S.next(s,u))
18:    else
19:      boundExceeded = true
20:    end if
21:  end while
22:  if boundExceeded then
23:    while ∃u ∈ (enabled(s) ∩ pending(s) \ visited) do
24:      add u to visited
25:      if Bv(S.next(s,u)) ≤ c then
26:        Explore(S.next(s,u))
27:      end if
28:    end while
29:  end if
30:  return boundExceeded
31: end
```

Algorithm 19 Backtrack (Preemption-bounded BPOR, bound optimization).

```
 1: procedure Backtrack(S, i, u) begin
 2:   AddBacktrackPoint(S, i, u, false)
 3:   if j = max({j ∈ dom(S) | j = 0 or S_{j−1}.tid ≠ S_j.tid and j < i}) then
 4:     AddBacktrackPoint(S, j, u, true)
 5:   end if
 6: end
 7: procedure AddBacktrackPoint(S, i, u, isConservative) begin
 8:   if u ∈ enabled(pre(S, i)) then
 9:     if isConservative then
10:       Add u to pending(pre(S, i))
11:     else
12:       Add u to backtrack(pre(S, i))
13:     end if
14:   else
15:     if isConservative then
16:       pending(pre(S, i)) = enabled(pre(S, i))
17:     else
18:       backtrack(pre(S, i)) = enabled(pre(S, i))
19:     end if
20:   end if
21: end
```

6.5 Sleep Sets

A sleep set is like a visited set that is allowed to propagate to subsequent states. We introduced sleep sets in Section 2.5.3. Assume that the search explores a transition t, and the entire state space reachable from t, from a state s. Then, the search pops t off the stack and explores another transition from s. Until the search explores a transition that is dependent with t, exploring t again will lead only to states that the search has already visited. The search explores an alternative transition t' from s because transitions that are dependent with t are reachable via t', yet t' itself may not be dependent with t. Sleep sets prevent the search from reordering t and t' back to their original order in this scenario.

Combining sleep sets with DPOR is not trivial. Flanagan and Godefroid

suggest that sleep sets combine with DPOR in a straight-forward manner [Flanagan and Godefroid, 2005], but we showed that their interactions are more subtle, which the authors acknowledged [Flanagan and Godefroid, 2011]. We focus here on the interaction between sleep sets and bounded search.

The key reason that the search may place t in the sleep set after exploring it from s is that the search has already explored the *entire* state space reachable via t prior to backtracking it. Thus, one obvious way to use sleep sets with BPOR is to never place a transition in the sleep set when exploring a conservative backtrack point. The conservative backtrack point was necessary because the entire search space was *not* accessible via t – the search believes it can reach states more cheaply via the conservative backtrack point. If t is in the sleep set, it may never reach those states. The simplest way we combine sleep sets with BPOR is to track whether each backtrack point was placed conservatively to offer a cheaper path to a previously visited state. If so, we do not add any transitions to the sleep set.

The bound optimization, described in Section 6.4, both enables sleep sets and complicates sleep sets. The bound optimization enables sleep sets because it tracks when the entire state space is reachable within the bound. If the entire state space was reachable, then the search does not add any conservative backtrack points. Conservative backtrack points limit sleep sets, so the bound optimization makes sleep sets more effective at reducing the state space. When we use the bound optimization, we apply this optimization to sleep sets as well. We incorporate this limited form of sleep sets in all of our DPOR and BPOR experiments.

Never placing transitions in the sleep set at a conservative backtrack point is unnecessarily conservative, however. For example, in preemption-bounded search, a conservative backtrack point backtracks a dependence with any of the transitions in an entire sequence of transitions by the executing thread. To optimize sleep sets for preemption-bounded search, the search can add *all* transitions in the sequence to

which the conservative backtrack point applies to the sleep set. If the search explores a transition that is dependent with *any* of those transitions, then it removes the corresponding thread from the sleep set and explores it when the next dependent transition occurs. We include this more aggressive algorithm as an optimization.

6.6 Combining Bound Functions

Combining bound functions may be advantageous if the bounds serve fundamentally different purposes. In particular, we would like to combine the fairness bound with any of the other bounds so that we can both reduce the size of the state space, and limit the number of times the search unrolls cycles in the state space. To combine bounds without sacrificing coverage, however, we must account for any interactions between those bounds and the dependences they introduce.

Fairness is particularly problematic when combined with bounds that reason about the enabledness of other threads, because the fairness criterion may enable and disable threads as their relative yield counts change. Although a transition that exceeds the bound is not "disabled" for the purposes of deadlock detection, a transition that exceeds the bound does leave part of the state space unreachable, so other bounds must account for another thread's being fair blocked. The context bound, in contrast, does not reason about the enabledness of other threads. The context bound cares only about which thread performed the prior transition, regardless of whether it, or any other thread, is enabled in the current state. Thus, context-bounded search combines more easily with fair-bounded search than preemption or delta-bounded search do.

To combine preemption and delta-bounded search with fairness, we introduce additional backtrack points to ensure that the search does not sacrifice bounded coverage. In addition to conservative backtrack points prior to release operations, we also place conservative backtrack points prior to acquire operations and any other

operation that may block. In delta-bounded search, for example, if a transition with a lower yield count but a higher thread priority is enabled, it may change the cost of other threads. This conservative approach ensures that all relevant states remain reachable within the bound, but it reduces the state space less aggressively than either bound does alone.

When using multiple bounds, it is also more likely that the search will reach a state in which all transitions exceed at least one of the bounds. For example, a preemption-bounded, fair-bounded search will often leave all transitions exceeding the bound, unless you assume that fair blocked threads get a free preemption. If all transitions exceed the bound, then the search must conservatively schedule all threads at their most recent cheaper locations to ensure that no states are left unexplored.

We conservatively place additional backtrack points whenever a thread's enabledness changes to eliminate these interactions. By exploiting dynamic information about the subsequent state space, however, these conservative assumptions could likely be optimized. We leave these additional optimizations to future work.

6.7 Discussion

The degree to which the optimizations described in this section reduce the state space varies with the benchmark and with the bound. Chapter 8 shows how these optimizations reduce the state space with each bound function. The benefit of these optimizations varies with the bound function and with the benchmark. Their benefits are complementary and applying all optimizations improves performance more than applying any one of them individually.

The optimizations in this chapter help BPOR reduce the state space more aggressively while providing bounded coverage. Ideally, however, BPOR would achieve as much partial-order reduction as unbounded search while still providing bounded

coverage. This ideal scenario is not possible because the bound introduces dependences between otherwise independent transitions. Unless a bound function is both stable and extensible, the search cannot guarantee that it has reached all states reachable within the bound without introducing conservative backtrack points. In the next section, we consider bound functions that are stable and extensible.

Chapter 7

Partial-Order Bounds

We propose a class of bound functions called *partial-order bounds* that bound the partial-order on a program's transitions, rather than the total order on those transitions. The context, preemption, delta, and fair bounds are all *total order bounds* because they bound properties of the total order on a program's transitions. As a result, these bounds introduce dependences between otherwise independent transitions. In Chapters 4 and 5, we compensate for these dependences by adding conservative backtrack points to maintain bounded coverage. As a result, we sacrifice partial-order reduction. In this chapter, we investigate bound functions that do not require conservative backtrack points because the state space reachable within the bound changes only with dependent transitions.

Many intuitive bounds, such as the context bound, bound the total order on a program's transitions. Total orders are easier to reason about than partial orders are, so a bound on the total order may be more useful to programmers. We modify several bounds on the total order such that they bound the partial order instead. The key insight is that we use the local state visible to each thread to determine the bounded value and use stable, extensible bound functions, as described in Section 4.2.1, to ensure that independent transitions do not leave portions of the state

space unreachable. We begin by showing that any bound function that is stable and extensible can achieve bounded coverage with BPOR without any conservative backtrack points.

7.1 Local Bound Sufficient Sets

We define sufficient sets for a stable, extensible bound function, prove that they are local sufficient, then show how to compute them with BPOR. Let Bv be a stable, extensible bound function. To provide local state reachability for Bv, we introduce local bound persistent sets.

Definition 7.1. Local bound persistent sets.
Let Bv be an extensible bound function. A set $T \subseteq \mathcal{T}$ of transitions enabled in a state $s = \mathit{final}(S)$ is *local bound persistent in* s if and only if for all nonempty sequences α of transitions from s in $A_{G(Bv,c)}$ such that $\forall i \in \mathit{dom}(\alpha), \alpha_i \notin T$ and for all $t \in T$,

1. $Bv(S.t) \leq c$
2. $t \leftrightarrow \text{last}(\alpha)$

Let $A_{R(Bv,c)}$ be the reduced state space explored by Algorithm 6 with bound function Bv and bound c. Assume that in each state, Algorithm 6 returns a local bound persistent set. We prove two lemmas to manage the bound, then show that a nonempty local bound persistent set is local sufficient.

Lemma 35. *Let α be a nonempty sequence of transitions from $s = \text{final}(S)$ in $A_{G(\text{Bv},c)}$ and let $t \in \text{enabled}(s)$ such that*

1. $\text{Bv}(S.t) \leq c$
2. $t \leftrightarrow \alpha$

Then, $t.\alpha$ is a sequence of transitions from s in $A_{G(\mathrm{Bv},c)}$.

Proof. By Assumption 2, $t.\alpha$ is a sequence of transitions from s in A_G. Because Bv is extensible and $t \leftrightarrow \alpha$,

$$Bv(S.t.\alpha) = max(Bv(S.t), Bv(S.\alpha))$$

By Assumption 1, $Bv(S.t) \leq c$. We are given that α is a sequence of transitions in $A_{G(Bv,c)}$ and thus $Bv(S.\alpha) \leq c$. Thus,

$$Bv(S.t.\alpha) \leq c$$

and $t.\alpha$ is a sequence of transitions from s in $A_{G(Bv,c)}$.

□

Lemma 36. *Let T be a nonempty local bound persistent set in a state $s = \mathrm{final}(S)$ in $A_{R(\mathrm{Bv},c)}$ and let $\alpha.t.\gamma$ be a sequence of transitions from s in $A_{G(\mathrm{Bv},c)}$ such that α is nonempty, $\forall i \in \mathrm{dom}(\alpha) : \alpha_i \notin T$, and $t \in T$. Then, $t.\alpha.\gamma$ is a sequence of transitions from s in $A_{G(\mathrm{Bv},c)}$.*

Proof. By Requirement 2 of Definition 7.1 of local bound persistent sets, $t \leftrightarrow \alpha$. Because $t \leftrightarrow \alpha$, by Definition 2.2 of a trace,

$$S.t.\alpha.\gamma \in [S.\alpha.t.\gamma]$$

Thus, because Bv is stable and $t \leftrightarrow \alpha$,

$$Bv(S.t.\alpha.\gamma) = Bv(S.\alpha.t.\gamma) \leq c$$

so $t.\alpha.\gamma$ is a sequence of transitions from s in $A_{G(Bv,c)}$.

□

Theorem 37. *If T is a nonempty local bound persistent set in a state s in $A_{R(\mathrm{Bv},c)}$, then T is local sufficient in s.*

Proof. Let s be a state in $A_{R(Bv,c)}$ and let l be a local state reachable from s in $A_{G(Bv,c)}$ via a nonempty sequence ω of transitions.

Case 37.1. $\forall i \in \boldsymbol{dom}(\omega) : \omega_i \notin T$**.**
Let t be any transition in T. By Requirement 1 of Definition 7.1 of local bound persistent sets, $Bv(S.t) \leq c$. Consider the sequence $\omega' = t.\omega$. By Requirement 2 of Definition 7.1 of local bound persistent sets, $t \leftrightarrow \omega$. Thus, $\omega.t \in [\omega']$, and $\omega \in \mathit{Pref}([\omega'])$. By Lemma 35, $t.\omega$ is a sequence of transitions from s in $A_{G(Bv,c)}$, and T is local sufficient in s.

Case 37.2. $\exists i \in \boldsymbol{dom}(\omega) : \omega_i \in T$**.**
Let $\omega = \alpha.t.\gamma$ such that $\forall i \in dom(\alpha) : \alpha_i \notin T$ and $t \in T$. Assume that α is empty. Then, T is local sufficient in s because $\omega_1 \in T$ and l is reachable via ω. Assume that α is nonempty. Consider the sequence $\omega' = t.\alpha.\gamma$, i.e., ω with t moved to the first position. By Definition 7.1 of local bound persistent sets, $t \leftrightarrow \alpha$. Thus, $\omega' \in [\omega]$ and $\omega \in \mathit{Pref}([\omega'])$. By Lemma 36, $t.\alpha.\gamma$ is a sequence of transitions from s in $A_{G(Bv,c)}$, and T is local sufficient in s.

□

By Theorems 37 and 3, if Algorithm 6 explores a nonempty local bound persistent set in each state then it reaches all local states reachable in $A_{G(Bv,c)}$. By Theorem 4, Algorithm 6 also reaches all deadlock states reachable in $A_{G(Bv,c)}$.

7.2 Computing Local Bound Persistent Sets

Algorithm 20 contains the **Initialize** and **Backtrack** procedures to compute local bound persistent sets with BPOR. The **Initialize** procedure adds any thread whose

Algorithm 20 BPOR procedures for monotonic, extensible, deterministic bound functions.

```
1: procedure Initialize(S) begin
2:    add any u ∈ enabled(final(S)) where Bv(S.next(final(S), u)) ≤ c to
      backtrack(final(S))
3: end
4: procedure Backtrack(S, i, u) begin
5:    if u ∈ enabled(pre(S, i)) then
6:       Add u to backtrack(pre(S, i))
7:    else
8:       backtrack(pre(S, i)) = enabled(pre(S, i))
9:    end if
10: end
```

next transition is enabled and within the bound to the backtrack set in *final*(S). The **Backtrack** procedure is the same procedure used by DPOR, because local bound persistent sets require no conservative backtrack points. In each state the search ensures the following postcondition before it backtracks.

Definition 7.2. ***PC*** **for Explore**(S) **for local bounded BPOR.**

$\forall u \forall \omega$: **if** $Bv(S.\omega) \leq c$ **then** $Post(S.\omega, len(S), u)$

Definition 7.3. ***Post***(S, k, u) **for extensible bound functions.**

$\forall v$: **if** $i = max(\{i \in dom(S) \mid S_i \not\rightarrow next(final(S), u)$ **and** $S_i.tid = v\})$ **and** $i \leq k$ **then**

if $u \in enabled(pre(S, i))$ **then** $u \in backtrack(pre(S, i))$

else $backtrack(pre(S, i)) = enabled(pre(S, i))$

Lemma 38. *Whenever Algorithm 8 backtracks a state* $s = \text{final}(S)$*, the set* T *of transitions explored from* s *is local bound persistent in* s*, provided that postcondition* PC *holds for every recursive call* ***Explore***$(S.t)$ *for all* $t \in T$*.*

Proof. Let $T = next(s, u) \mid u \in backtrack(s)$. Show that if T violates any requirement in Definition 7.1 of local bound persistent sets, then we have a contradiction.

Case 38.1. T **violates Requirement 1.**

Algorithm 8 explores only transitions that do not exceed the bound. Because the search explores t, it must not exceed the bound from $\mathit{final}(S)$.

Case 38.2. T **violates Requirement 2.**

Let $T = \mathit{next}(s, u) \mid u \in \mathit{backtrack}(s)$. Proceed by contradiction. Assume that there exists a nonempty sequence α of transitions from s in $A_{G(Bv,c)}$ and a transition $t \in T$ such that

1. $Bv(S.t) \leq c$
2. $\forall i \in \mathit{dom}(\alpha) : \alpha_i \notin T$
3. t is dependent with $\text{last}(\alpha)$

Let $n = \mathit{len}(\alpha)$ and let $\omega = \alpha_1 \ldots \alpha_{n-1}$, i.e., α with its last transition removed. Let there be no prefixes of α that also meet the criteria above, and thus

3. $t \leftrightarrow \omega$

Let $u = \text{last}(\alpha).\mathit{tid}$. Assume that $t.\mathit{tid} = u$. Because $t \leftrightarrow \omega$,

$$t = \mathit{next}(\mathit{final}(S), u) = \mathit{next}(\mathit{final}(S.\omega), u) = \text{last}(\alpha)$$

Thus, $\text{last}(\alpha) \in T$ and we have a contradiction. Assume that $t.\mathit{tid} \neq u$. Consider the postcondition

$$\mathit{Post}(S.t.\omega, \mathit{len}(S) + 1, u)$$

for the recursive call **Explore**$(S.t)$. By Lemma 35, $S.t.\omega$ is a sequence of transitions from s in $A_{G(Bv,c)}$. Because $t \leftrightarrow \omega$, t is the most recent transition by $t.\mathit{tid}$ that is dependent with $\mathit{next}(\mathit{final}(S.t.\omega), u)$. Thus, by Definition 7.3 of *Post*, either $u \in \mathit{backtrack}(s)$, or $\mathit{backtrack}(s) = \mathit{enabled}(s)$ and thus $\alpha_1 \in T$. In either case, we have a contradiction.

□

Thus, if postcondition PC holds in each state s explored by Algorithm 8, then the set of transitions explored from s is local bound persistent in s. Next, we prove that postcondition PC holds in each state s explored by Algorithm 8. First, we prove a lemma that simplifies the inductive step.

Lemma 39. *Let* $s = \text{final}(S)$ *be a state in* $A_{R(\text{Bv},c)}$*, let* ω *and* ω' *be nonempty sequences of transitions from* s *in* $A_{G(\text{Bv},c)}$*, and let* u *be a thread such that*

1. $\exists \beta : \omega.\beta \in [\omega']$ ***and*** $\beta \leftrightarrow \text{next}(\text{final}(S.\omega), u)$*, or*
2. $\exists \beta : \omega'.\beta \in [\omega]$ ***and*** $\beta \leftrightarrow \text{next}(\text{final}(S.\omega), u)$

Then, $\text{Post}(S.\omega', \text{len}(S) + 1, u) \implies \text{Post}(S.\omega, \text{len}(S), u)$.

Proof. Because $\beta \leftrightarrow \mathit{next}(\mathit{final}(S.\omega), u)$,

$$\mathit{next}(\mathit{final}(S.\omega), u) = \mathit{next}(\mathit{final}(S.\omega'), u)$$

Assume that for some thread v in Definition 7.3 of $\mathit{Post}(S.\omega, \mathit{len}(S), u)$, $i > k$. Then, *Post* does not require any backtrack points for v.

Assume that for some thread v in Definition 7.3 of $\mathit{Post}(S.\omega, \mathit{len}(S), u)$, $i \leq k$. Then, because $\beta \leftrightarrow \mathit{next}(\mathit{final}(S.\omega), u)$, i is the same for thread v in $\mathit{Post}(S.\omega', \mathit{len}(S), u)$. Thus, by Definition 7.3 of *Post*,

$$\mathit{Post}(S.\omega, \mathit{len}(S), u) \text{ iff } \mathit{Post}(S.\omega', \mathit{len}(S), u) \tag{7.1}$$

Because Definition 5.2 of *Post* requires that i be less than or equal to k,

$$\mathit{Post}(S.\omega', \mathit{len}(S) + 1, u) \implies \mathit{Post}(S.\omega', \mathit{len}(S), u)$$

Thus, by Equation 7.1,

$$Post(S.\omega', len(S)+1, u) \implies Post(S.\omega, len(S), u)$$

□

Theorem 40. *Whenever Algorithm 8 backtracks a state $s = \text{final}(S)$ in an acyclic state space, the postcondition* Post *for* ***Explore***(S) *is satisfied, and the set T of transitions explored from s is local bound persistent in s.*

Proof. The proof is by induction on the order in which states are backtracked.

Base case.

Because the search is acyclic and performed in depth-first order, the first backtracked state must be a deadlock state in which no transition is enabled, or a state in which all transitions exceed the bound. Thus, the postcondition for the first backtracked state is

$$\forall u : Post(S, len(S), u)$$

and is directly established by Lines 4-10 in Algorithm 8.

Inductive case.

Assume that each call to **Explore**$(S.t)$ satisfies its postcondition. By Lemma 38, T is local bound persistent in s. Show that **Explore**(S) satisfies its postcondition for any sequence ω of transitions from s in $A_{G(Bv,c)}$ and for any thread u. If ω is empty then $Post(S, len(S), u)$ is directly established by Lines 4-10 in Algorithm 8, so assume that ω is nonempty.

Case 40.1. $\forall i \in$ ***dom***$(\omega) : \omega_i \notin T$ **and** $u \in$ ***backtrack***(s)**.**

Because $u \in backtrack(s)$, $next(s, u) \in T$. Thus, by Definition 7.1 of local bound

persistent sets, $next(s, u) \leftrightarrow \omega$, and

$$next(final(S.\omega), u) = next(s, u)$$

Thus, $next(final(S.\omega), u) \leftrightarrow \omega$, and therefore $Post(S.\omega, len(S), u)$ iff $Post(S, len(S), u)$. The latter is directly established by Lines 4-10 in Algorithm 8.

Case 40.2. $\forall i \in \textbf{\textit{dom}}(\omega) : \omega_i \notin T$ **and** $u \notin \textbf{\textit{backtrack}}(s)$**.**
Let t be any transition in T. Consider the sequence $\omega' = t.\omega$. By Requirements 1 and 2 of Definition 7.1 of local bound persistent sets, $Bv(S.t) \leq c$ and $t \leftrightarrow \omega$. Thus, by Lemma 35 ω' is a sequence of transitions in $A_{G(Bv,c)}$. Because $t \leftrightarrow \omega$

$$\omega.t \in [\omega'] \tag{7.2}$$

By the inductive hypothesis for the recursive call **Explore**$(S.t)$,

$$Post(S.\omega', len(S) + 1, u)$$

We will show by contradiction that $t \leftrightarrow next(final(S.\omega'), u)$. Otherwise, assume that t is dependent with $next(final(S.\omega'), u)$. Because $t \leftrightarrow \omega$, t is the most recent transition by $t.tid$ that is dependent with $next(final(S.\omega'), u)$. Thus, by Definition 7.3 of *Post*, either $u \in backtrack(s)$ or $backtrack(s) = enabled(s)$ and thus $\omega_1 \in T$. In either case, we have a contradiction.

Assume that $t \leftrightarrow next(final(S.\omega'), u)$. Because $t \in T$ and $u \notin backtrack(s)$, $t.tid \neq u$. Thus, $next(final(S.\omega), u) = next(final(S.\omega'), u)$ and

$$t \leftrightarrow next(final(S.\omega), u)$$

Thus, by Lemma 39 where $\beta = t$ and $\omega.\beta \in [\omega']$ by Equation 7.2,

$$Post(S.\omega, len(S), u)$$

Case 40.3. $\exists i \in \textbf{\textit{dom}}(\omega) : \omega_i \in T$**.**

Let $\omega = \alpha.t.\gamma$ such that

1. $\forall i \in dom(\alpha) : \alpha_i \notin T$
2. $t \in T$

If α is empty, then $\omega_1 \in T$ and by the inductive hypothesis

$$Post(S.\omega, len(S) + 1, u)$$

Because Definition 5.2 of *Post* requires that i be less than or equal to k,

$$Post(S.\omega, len(S) + 1, u) \implies Post(S.\omega, len(S), u)$$

as required.

Assume that α is nonempty. Consider the sequence $\omega' = t.\alpha.\gamma$, i.e., ω with t moved to the beginning. By Lemma 36, ω' is a sequence of transitions from s in $A_{G(Bv,c)}$. By Definition 7.1 of local bound persistent sets, $t \leftrightarrow \alpha$. Thus, by Definition 2.2 of a trace,

$$\omega' \in [\omega] \tag{7.3}$$

By the inductive hypothesis for the recursive call **Explore**$(S.t)$,

$$Post(S.\omega', len(S) + 1, u)$$

and thus by Lemma 39 where β is empty and $\omega \in [\omega']$ by Equation 7.3,

$$Post(S.\omega, len(S), u)$$

□

Thus, Algorithm 8 explores an local bound persistent set T of transitions from each state s. By Theorem 5, T is local sufficient in s, and by Theorem 4, T is deadlock sufficient in s. Thus, Algorithm 8 explores all local and deadlock states reachable within the bound for acyclic state spaces if bound function Bv is stable and extensible. Because each increment of the bound in an extensible bound function coincides with a pair of dependent transitions, the bounded value cannot increment unless the search also enters a new local state. Next, we show how to build a stable, extensible bound function for depth-bounded search.

7.3 Local Depth Bound

We introduce a *local depth bound* to provide local state reachability with depth-bounded search. Depth-bounded search cannot provide local state reachability and achieve partial-order reduction because the depth bound is not extensible (Definition 4.5). Different sequences of transitions that lead to the same local state may have different depths. To make the depth bound extensible, we bound the depth of the partial order, rather than the total order, on the transitions in an execution.

We use vector clocks to keep track of the local depth for each thread because vector clocks naturally encode information about the local state visible to each thread [Fidge, 1988]. A vector clock stores one value for each thread in the program. For each thread, we add a vector clock. For each variable, we add two vector clocks, one for reads and one for writes. DPOR uses vector clocks to keep track of the most recent dependent transitions and to compute the happens-before relationship,

so using these vector clocks to compute the bound function does not add significant additional overhead. First, we define several standard functions on vector clocks.

Definition 7.4. Vector clock join.

Let C and R be vector clocks and let v be a thread identifier.

$$join(C, R) = \lambda v.max(C[v], R[v])$$

Definition 7.5. Vector clock increment.

Let C be a vector clock and let u and v be thread identifiers.

$$inc(C, u) = \lambda v. \begin{cases} C[v] + 1 & \text{if } v = u \\ C[v] & \text{otherwise} \end{cases}$$

Definition 7.6. Vector clock sum.

Let C be a vector clock of size n. Then,

$$sum(C) = \sum_{u=0}^{n} C[u]$$

The local depth bound first defines several state updates that occur each time a transition by a thread u reads or writes a variable x. These state updates maintain the vector clock for each thread u, C_u, the vector clock for writes to each variable x, W_x, and the vector clock for reads to each variable x, R_x. The local depth bound uses these vector clocks.

Definition 7.7. Local depth bound (*Ld*).

We update C_u, R_x, and W_x as follows for all threads u and for all variables x. After each read operation t, $read(t.tid, t.var)$. After each write operation t, $write(t.tid, t.var)$.

$$
\begin{aligned}
read(u,x): \quad & C'_u = inc(join(C_u, W_x), u) \\
& R'_x[u] = C_u[u] + 1 \\
write(u,x): \quad & C'_u = inc(join(join(C_u, W_x), R_x), u) \\
& W'_x = inc(join(join(C_u, W_x), R_x), u)
\end{aligned}
$$

Define the local depth bound recursively as follows:

$$Ld(t) = 1$$

$$Ld(S.t) = max(Ld(S), sum(C_{t.tid}))$$

Intuitively, the local depth bound limits the maximum number of transitions visible to a particular thread, across all threads. The read and write operations in Definition 7.7 maintain standard vector clocks. The read operation joins the value in W_x into C_u to ensure that all transitions that were visible to the thread that last wrote x at the time it wrote x are now visible to u, as well. The ith transition by a thread v is "visible" to u if $i \leq C_u[v]$.

The increment operation accounts for u having performed its access to x. Note that the read operation's update to C_u and its update to R_x are the same, except that the read update accesses only u's component in the vector clock. The join operation is not necessary for the read vector update because u always has the most up-to-date value for its own slot in its vector clock.

The write operation performs the same join operation that the read operation does because u's write to x is also dependent with the most recent write to x. The write operation then additionally performs a join with the read vector because the write is also dependent with the most recent read by each thread. Finally, the write operation increments its own clock to account for its having performed its write operation. The updates to C_u and W_x are identical in this case because accesses to write operations are totally ordered.

Finally, the bounded value is equal to the maximum vector clock sum across all threads. Each thread u's vector clock indicates how many transitions by each

other thread are visible to u in its local state. The key insight for the local depth bound is that vector clocks naturally encode the bound increments that are visible to each thread in its local state. A thread u's vector clock stores the number of transitions by each thread that are visible to u. The sum of these values is the depth of the sequence of transitions that results if the search explored *only* the transitions visible in u's local state. So, even if the search explores a sequence of transitions that exceeds the depth bound, if that sequence does not exceed the local depth bound then there must exist a valid sequence of transitions that would lead to the same local state within the bound.

Theorem 41. *The local depth bound is stable.*

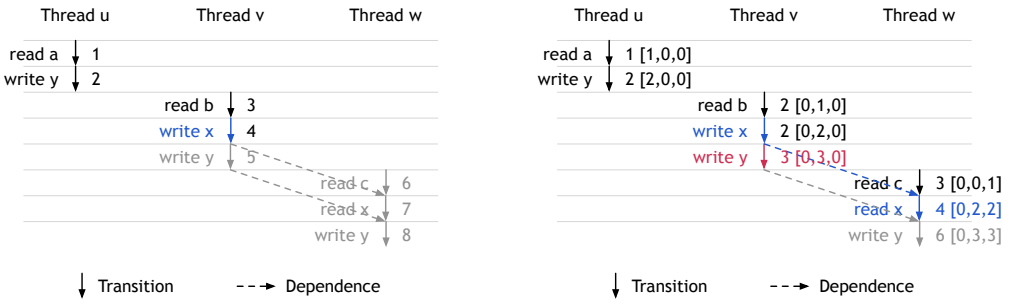
The proof of this theorem follows from Definition 7.7 and the fact that the vector clock values after $S.\omega$ and $S.\omega'$ must be equal [Fidge, 1988].

Theorem 42. *The local depth bound is extensible.*

Proof. Proceed by contradiction. Let $s = \mathit{final}(S)$ be a state in $A_{G(Ld,c)}$. Assume that there exist a transition t enabled in s and a sequence of transitions α from s such that $t \leftrightarrow \alpha$. Assume that

$$Ld(S.t.\alpha) \neq max(Ld(S.t), Ld(S.\alpha)) \tag{7.4}$$

Because $t \leftrightarrow \alpha$, $\forall i \in dom(\alpha) : t.tid \neq \alpha_i.tid$. Thus, by Definition 7.7 of the local depth bound, t updates $C_{t.tid}$ and may update $W_{t.var}$ or $R_{t.var}$. Similarly, each transition α_i updates $C_{\alpha_i.tid}$ and may update $W_{\alpha_i.var}$ or $R_{\alpha_i.var}$. Because the local depth bound returns the maximum of its current value and the vector sum of the most recent transition's thread in each state, the only way that Equation 7.4 can be true is if t accesses the same variable as one of the transitions in α, and thus changes its vector clock. Thus, for some $i \in dom(\alpha) : t.var \cap \alpha_i.var \neq \emptyset$. If both t and α_i are read operations then they do not affect one another's vector clocks because read



(a) Depth bound = 4

(b) Local depth bound = 4

Figure 7.1: A sequence of transitions with a depth bound and local depth bound of four.

operations only affect thread's vector clocks on write operations. Thus, either t or α_i is a write operation. By Definition 2.4, t is dependent with α_i, so we have a contradiction.

□

Because the local depth bound is stable and extensible, dynamic partial-order reduction will combine with local depth-bounded search without requiring conservative backtrack points, as shown in Section 7.1. Because local depth-bounded search with bound c explores all local states that contain c or fewer transitions, local depth-bounded search explores all of the same local states that depth-bounded search explores. Local depth-bounded search permits full partial-order reduction, however, while depth-bounded search cannot reduce the state space at all without sacrificing bounded coverage.

Figure 7.1 illustrates a sequence of transitions with a depth bound of four and a local depth bound of four. The bounded value after performing each transition appears next to the transition. The figure on the right also shows the vector clock for the thread that performed the most recent transition next to each transition.

With the depth bound, as shown on the left side of Figure 7.1, if the search

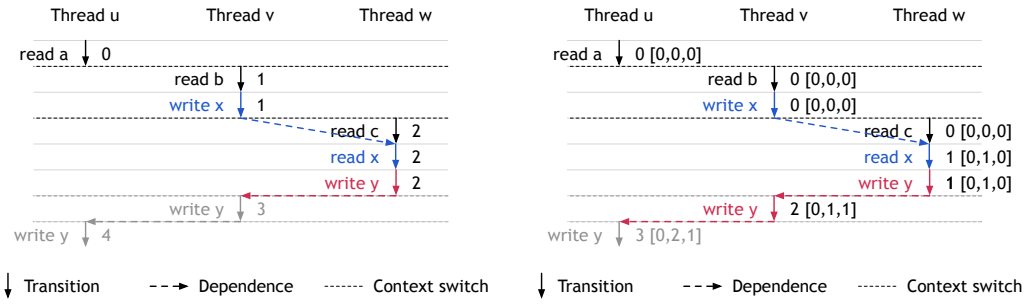
performs DPOR then it never explores local states that are reachable within the bound. Because the transitions by Thread w that are dependent with transitions by Thread v exceed the bound, DPOR never observes them and thus never adds any backtrack points for them. To ensure that the search visits all local states, this search must sacrifice partial-order reduction.

With the local depth bound, as shown on the right side of Figure 7.1, the search explores all local states that are reachable from the initial state via a sequence that contains four or fewer transitions. The right side of Figure 7.1 shows the additional local states that are reachable within the bound. Note that each local state that is reachable on the right side of Figure 7.1 is reachable within the bound from the initial state. For example, thread w's local state after its read of x is reachable from the initial state via the sequence of transitions that begins with thread v's first two transitions, followed by Thread w's first two transitions. Even though the search explores this local state via a path that contains seven transitions, the vector clocks reveal that the state is actually reachable with only four transitions so DPOR explores Thread w's read of x, observes its dependence with Thread v's write to x, and inserts a backtrack point for w prior to Thread v's write to x.

This technique for bounding the partial order on the program's transitions is effective for other bounds. Next, we show how to compute a local context bound. The general vector clock concept is still applicable with the context bound, but we modify the vector clock updates to reflect the new bound.

7.4 Local Context Bound

We use a similar approach to the local depth bound to compute a local context bound in each state. The key difference between the local depth bound and the local context bound is that the local depth bound for a particular thread increments with every transition by that thread, while the local context bound only increments



(a) Context bound = 2

(b) Local context bound = 2

Figure 7.2: A sequence of transitions with a context bound and local context bound of two.

due to cross-thread dependences. In particular, the local context bound increments only after exploring the sink node of an inter-thread dependence.

Definition 7.9 defines read and write functions for local context-bounded search, and defines the local context bound. First, we illustrate the local context bound to provide intuition. Figure 7.2 shows an example of two sequences of transitions with context bound two and local context bound two. The vector clock value for the thread that performed each transition appears beside the transition in the figure on the right. We define the updates for these vector clocks in Definition 7.9. The number next to each transition indicates the bounded value. Context-bounded search does not explore Thread v's write to y because the execution contains more than two context switches. The local context bound, in contrast, counts only context switches that are visible to a given thread. The context switch away from Thread u's first transition is not visible to any thread because no other threads perform conflicting accesses to a. Thus, local context-bounded search explores Thread v's write to y and thus discovers new dependences that BPOR must backtrack.

Local context-bounded search must increment a thread's vector clock only after an inter-thread dependence has been observed. Thus, we do not update a thread's vector clock every time it performs a transition as we did with local depth-

bounded search. Instead, we embed the vector clock increment in the variable's vector clock, so that the bound will update only when a different thread performs a dependent transition. To accommodate this difference, we define a modified version of the *join* function for context-bounded search.

Definition 7.8. Vector clock join, local context bound.
Let C and R be vector clocks.

$$csjoin(C, R, v) = \lambda w. \begin{cases} C[w] & \text{if } w = v \\ max(C[w], R[w]) & \text{otherwise} \end{cases}$$

Intuitively, the *csjoin* function takes the point-wise maximum of the values in its two vector clock inputs, *except* for the value associated with Thread v, which it leaves equal to the left-hand argument. The local context bound must not increase until the search explores the sink node of an inter-thread dependence, because any prior context switches may be unnecessary. When a write operation joins values from prior read operations, it must not incorporate reads by its own thread because they are concurrent with reads by other threads and thus do not require a context switch unless an intervening write has occurred. The *csjoin* function allows write operations to incorporate other thread's read operations, but not its own.

Definition 7.9. Local context bound (*Lc*).
We update C_u, R_x, and W_x as follows for all threads u and for all variables x. After each read operation t, $read(t.tid, t.var)$. After each write operation t, $write(t.tid, t.var)$. Let $O_x = u$ if and only if u performed the most recent write to x and no other thread has read x since.

$$read(u,x): \quad C'_u = \begin{cases} C_u & \text{if } O_x = u \\ join(C_u, W_x) & \text{otherwise} \end{cases}$$

$$R'_x[u] = \begin{cases} C_u[u] + 1 & \text{if } O_x = u \\ join(C_u, W_x)[u] + 1 & \text{otherwise} \end{cases}$$

$$write(u,x): \quad C'_u = \begin{cases} C_u & \text{if } O_x = u \\ csjoin(join(C_u, W_x), R_x, u) & \text{otherwise} \end{cases}$$

$$W'_u = \begin{cases} inc(C_u, u) & \text{if } O_x = u \\ inc(csjoin(join(C_u, W_x), R_x, u), u) & \text{otherwise} \end{cases}$$

Define the local context bound recursively as follows:

$Lc(t) = 1$

$Lc(S.t) = max(Lc(S), sum(C_{t.tid}))$

Intuitively, the local context bound tracks the number of context switches that must occur to reach each thread's local state. Definition 7.9 is very similar to Definition 7.7, even though they may appear quite different at first glance. There are two primary differences between the two definitions. First, rather than increment the vector clock C_u after each transition, Definition 7.9 leaves C_u unchanged and increments only the variable's vector clocks, R_x and W_x. By incrementing the variable's vector clock, the thread records its access for threads that access the variable in the future without updating its own clock.

The second primary difference is that in Definition 7.9, the read and write operations do not incorporate read and write values from other threads in the case where $O_x = u$. In this case, Definition 7.9 increments only the read and write values that record Thread u's access for future accesses to observe. If the same thread accesses the same variable multiple times without any intervening dependent accesses, then it will observe the incremented value that it wrote even if no context

switch occurs. Checking that $O_x \neq u$ thus ensures that an intervening dependent access to x has occurred since u last accessed it. Note that although the increment operation for the read and write vectors occurs repeatedly when the same thread accesses a variable repeatedly, the increment operation is performed on the vector clock, which remains unchanged, so the value increments by only one.

The read operation in Definition 7.9 updates u's vector clock and u's slot in x's vector clock. The join operation with W_x makes context switches that were required to reach the most recent prior write visible to u after the read operation. The read operation update is identical to the vector clock update, except that it accesses only u's slot within the vector clock, and it increments that value. This increment represents the context switch away from thread u that may occur in the future, and it will not be incorporated into any thread's vector clock until a thread other than u performs a conflicting access to x.

The write operation in Definition 7.9 updates u's vector clock and x's vector clock. The update to W_x is identical to the update to C_u, except that the update to W_x includes an increment operation. Similar to the read operation, the increment of W_x indicates to future conflicting accesses to x that a context switch has occurred. Thread u's vector clock does not increment because Thread u has not yet observed this future context switch.

The write operation performs $join(C_u, W_x)$, just like the read operation does, because the write is also dependent with the most recent write to x. The write operation additionally performs a modified join operation to incorporate the information in the read vector for x. The modified join operation is important because Thread u may have previously performed a read operation on x and updated u's slot in x's read vector by incrementing it. This increment is a signal to *other threads* that u performed a read operation on x, so u must not incorporate this value. The only way u incorporates this context switch is if an intervening write operation to x by

another thread occurs.

The local context bound is stable and extensible for the same reasons that the local depth bound is stable and extensible – the bound function returns a maximum, rather than a cumulative value at each step, and the only way the cost of a transition can change is if a dependent access occurs. The local context bound shows how to encode a second bound function into vector clocks to provide local-state reachability with partial-order reduction. Although we do not discuss other local bounds in depth, the same approach could likely be adapted to other bounds.

7.5 Discussion

Prior work suggests that combining bounded search with partial-order reduction is impractical [Musuvathi and Qadeer, 2007b]. In particular, Musuvathi and Qadeer prove the following theorem. Let $Pb(s)$ return the minimum value of $Pb(S)$ among all sequences S such that $\mathit{final}(S) = s$.

Theorem 43. *Given a state s and an integer $c \geq 0$, the problem of determining whether* $\mathrm{Pb}(s) \leq c$ *is NP-complete.*

Musuvathi and Qadeer prove this theorem by reduction to the minimum feedback-vertex set problem, which is known to be NP-complete. The minimum feedback-vertex set problem is to find, given a directed graph $G(V, E)$, a subset V' of V of size c such that every directed cycle of G contains at least one vertex in V'.

Musuvathi and Qadeer build states by mapping vertices to pairs of instructions (u_{src}, u_{dst}) with an intra-thread edge between them, and mapping edges to inter-thread dependences between (u_{src}, v_{dst}) for vertices (u_{src}, u_{dst}) and (v_{src}, v_{dest}). Thus, cycles map to dependences from a thread u to a thread v, with a dependence that also goes from v to u. Although Musuvathi and Qadeer prove this result for preemption-bounded search, a similar proof would hold for context-bounded search

because the proof does not rely on the enabledness of the threads.

A local context or preemption bound would appear to solve this problem – it returns the minimum bound with which the search could have explored the sequence in question. We do not prove that the local context bound explores all local states that can be reached with c or fewer context switches, but our experiments suggest that it may.

If local context-bounded search does explore all local states that contain c or fewer context switches, however, that does not necessarily imply that it returns the smallest number of context switches with which each state could be reached. In particular, the search can over-estimate the number of context switches required to reach a state *provided that the search need not explore that segment of the state space*. If the search inserts a backtrack point that is not necessary, then the local context bound will over-estimate its context switch bound. Because this backtrack point and all subsequent states are unnecessary, however, this inaccuracy does not affect coverage guarantees.

The local context bound performs best with a non-preemptive scheduler. For a context switch to really be "necessary", there must exist, as the minimum feedback-vertex set problem suggests, a cycle in which a transition by a thread v is dependent with a prior transition by a thread u, and then some subsequent transition by u is dependent with a transition by v, as well. The local context switch bound detects only one of these dependences, the first one that occurs. The local context bound implicitly assumes that the other dependence exists. In particular, BPOR would not insert a backtrack point unless the dependence existed. If the dependence does exist, then the local context bound is correct in incrementing the context bound. If the dependence does not exist, then the local context bound is inaccurate, but the state space that becomes unreachable is redundant and unnecessary, so the inaccuracy does not sacrifice coverage.

If a local context-bounded search inserts only necessary backtrack points, then it does appear to compute the minimum context bound for each state. Even with partial-order reduction, however, model checking is exponential in the number of conflicting transitions. The BPOR algorithm's exponential exploration of the reduced state space determines which backtrack points are necessary. The local context bound leverages this exponential computation because it assumes that each backtrack point is necessary, and that the search will never omit a reachable, necessary backtrack point. Thus, it is not surprising that it can approximate an NP-complete result quite accurately.

7.6 Other Bounds

Future work should use this approach to design other partial-order bounds that incrementally prune the state space in a useful manner while permitting partial-order reduction. This class of bounds combines insights from each of the other bound functions we study. The depth-bound is stable, but it is not extensible and cannot combine with dynamic partial-order reduction for local state reachability. The context bound demonstrates that providing a way to *not* increment the bound is crucial. In states where no context switch is required, the executing thread is free and the search can reduce the state space if it does not encounter any dependent, co-enabled transitions. The context bound also allows the tester to search deeper into the state space with a small bound. Both the depth bound and the context bound are not extensible, however, so they cannot provide local state reachability with bounded partial-order reduction.

The preemption bound always provides a zero-cost path to previously unvisited local states. Because the preemption bound reasons about the enabledness of other threads and about which transition performed the previous transition, however, it introduces artificial dependences that restrict transitions that would oth-

erwise be commutative. These restriction sacrifice partial-order reduction. Delta-bounded search concedes that BPOR must introduce conservative backtrack points, but incorporates a built-in heuristic to guide the search towards the cheapest executions first – the unique cost of each transition. This approach still falls short, however, because it reasons about the enabledness of independent threads, and it offers a cheaper path to portions of the state space when the search must explore transitions other than the cheapest transition.

Finally, the fair bound shows that the bound can prune cycles in a cyclic state space, and that the bound need not be a cumulative value – the fair bound tracks a maximum value, instead, and thus has a different effect on the state space and on partial-order reduction. Each bound provides insights about the state space, the power of bounded search, and its interactions with partial-order reduction. Each bound also helps clarify the value of partial-order bounds.

Partial-order bounds use the insights gathered from total order bounds to reduce the state space effectively while still providing bounded coverage. The key insight is that the bound cannot introduce new dependences without sacrificing partial-order reduction. By building stable, extensible bound functions that leverage only the locally visible transitions, the search can explore all local states reachable within the bound while still reducing the state space significantly. In the next section, we evaluate each of the algorithms that we describe in a dynamic model checking tool for concurrent software.

Chapter 8

Results

We evaluate BPOR by measuring its state space reduction, memory footprint, and time required to manifest known bugs. We compare different bound functions both with and without fairness, and we evaluate each optimization's effect on testing time and state space reduction. BPOR adds significant overhead to each execution, so we measure the number of total tests per unit time, in addition to the degree of state space coverage over time.

8.1 Methodology

We implement BPOR in CHESS, a stateless, dynamic model checker for concurrent software. When a concurrent test program runs under CHESS's control, CHESS places a thin wrapper between the program under test and the Win32 API using binary instrumentation [Musuvathi et al., 2009]. This wrapper intercepts calls into the Win32 and .NET APIs and provides hooks into CHESS that control thread scheduling completely without modifying the semantics of the API or the behavior of the program under test.

CHESS controls thread scheduling by inserting extra synchronization into its wrappers that ensures that only one thread is enabled at a time. Thus, CHESS totally orders a program's execution. The CHESS engine records the transitions in the stack for each execution, then pops those transitions off the stack to explore a new execution. CHESS is stateless – it does not explicitly store information about previously visited states, except to track progress for reporting results.

We implement GAMBIT, DPOR, bounded search, and BPOR inside the CHESS engine. After each execution of the program under test completes, CHESS provides a hook that allows it to evaluate the completed execution and choose which execution to explore next. The algorithms presented here answer this question – which execution should the scheduler explore next. The next execution is fully described by the transitions that remain on the stack from the prior execution, an explicit next transition that the scheduler must explore after those transitions complete, and the thread scheduler's default behavior. This systematic, deterministic default behavior of the thread scheduler determines which transitions it will explore after the explicit backtrack point. We use a non-preemptive, round-robin scheduler.

8.2 Benchmarks

We test each algorithm on concurrent unit tests developed by testers for concurrent software and libraries at Microsoft such as the Concurrency Coordination Runtime (CCR) and the .NET 4.0 concurrency libraries. We also provide results on a microbenchmark that we created explicitly to test fair-bounded search without partial-order reduction. Because fair-bounded search prunes only cycles in the state space, its state space is intractably large without partial-order reduction or other bound functions.

Table 8.1 summarizes the tests that we include and shows the symbol that we use to refer to them. The size of the test in Column 4 is the maximum depth of

Program	Unit test	Size	Threads	Description
CCR	`Exception`	127	3	Concurrent programming model based on message-passing with orchestration primitives that coordinate data and work.
Fairness	`Fair`	79	3	Microbenchmark to test fair-bounded search
Futures	`NQueens` `Matrix`	6384	4	Imperative task-parallelism for .NET 4.0
ParallelFFT	`Test4On4`	725	7	Parallel FFT computation
Region Ownership	`RegOwn`	1038	5	Ownership-based separation of the heap for parallel programs
Reset Event	`MRSE`	93	3	Unit test for manual reset event slim concurrency primitive.

Table 8.1: Programs and corresponding unit tests with the maximum number of transitions in a single execution in parenthesis.

the stack of transitions across all executions of the program under test. This number is generally a good indicator of the size of the unreduced state space. The `RegOwn`, `NQueens` and `Matrix` tests contain known bugs and thus do not run to completion. We use these tests to evaluate how long it takes each configuration of CHESS to manifest the bug, both in terms of visited states and in terms of time.

8.3 Validation

A coverage guarantee is not meaningful without a proof, and likewise a verification tool is not useful unless it is implemented correctly. We validate our implementation in several ways. First, we hash states to track the number of unique states that the search visits, and to compare different invocations of CHESS. Second, we automatically generate and test random concurrent programs. Third, we explicitly verify that the lemmas that we prove are true. We use the term "invocation of CHESS" to refer to CHESS searching the entire reachable state space for a program with a

particular set of inputs. One invocation of CHESS includes many invocations of the program under test – one for each explored interleaving.

Because CHESS is stateless it does not store any record of previously visited states, so determining how many unique states the search visits and whether it visits the same states during multiple invocations is non-trivial. Musuvathi and Qadeer create a hash of the partial-order on a program's states to track which states have been visited efficiently and with little storage overhead [Musuvathi and Qadeer, 2007b]. We use this approach to track the number of visited deadlock states, and augment their hash function by differentiating read operations from write operations. We also modify the hash computation to associate a unique value with each local state, rather than each deadlock state. This validation is imperfect because hash collisions do occur, but these hash values provide a very useful sanity check.

We use these hash values to track the number of unique local and deadlock states that a search visits and compare this value across multiple invocations of CHESS to ensure that different searches of the same state space explore the same number of unique local and deadlock states. In addition, we print these hash values and explicitly compare them across invocations to ensure that each search explores not only the same *number* of states, but also a set of states that maps to the same hash values. Otherwise, bonded search might explore the same number of local states, yet if some of those states are not reachable within the bound, the search may still fail to provide coverage within the bound. These hash functions are crucial to verifying the implementation quickly and effectively.

Random search detects bugs very effectively [Dwyer et al., 2007], so we generate small, random, concurrent programs and run them under CHESS both with and without BPOR to guarantee that BPOR explores the same state space that the bounded search explores without any partial-order reduction. We compare the hash values described above across invocations of thousands of small, randomly generated

test programs. This tool was also crucial because it generated a wider variety of behaviors than the limited set of available regression tests.

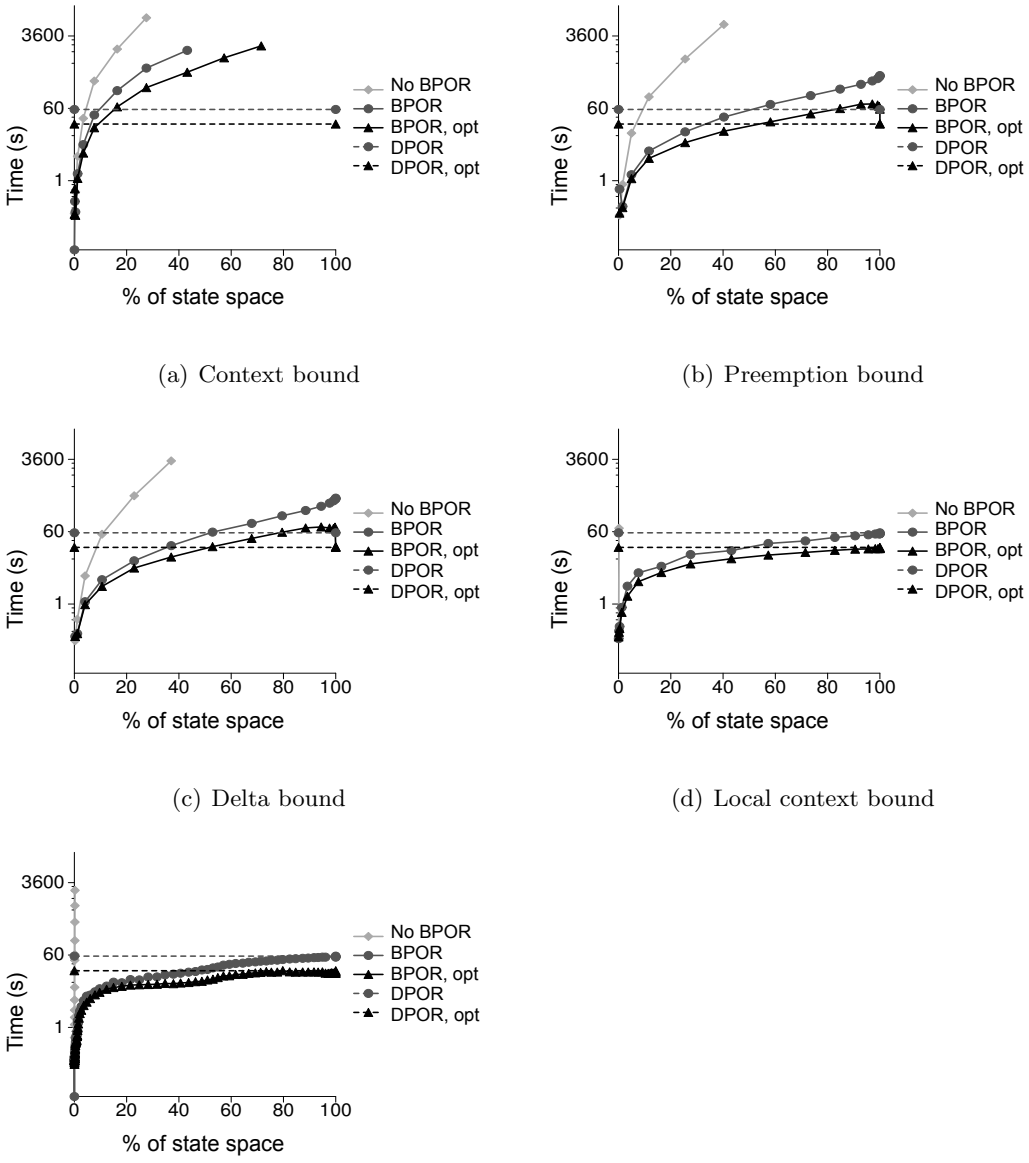
Finally, we explicitly check that the lemmas that we prove in Chapter 5 are true at runtime. We output information about each execution of the program under test including the set of backtrack points the search created, the enabled threads, and the next backtrack point that the search selected. We post-process this information, which includes data about the entire explored state space, and use it to explicitly check each bound function's postcondition *Post* in each state.

8.4 Coverage Time

We measure coverage over time for each benchmark with each bound function by tracking the percent of visited local states. If the total number of states is unknown because the state space is too large, then we use the total number of visited states rather than the percent of visited states. Because we measure time rather than total explored states, these results factor in the overhead of executing BPOR. Figures 8.1-8.4 provide these results for each benchmark, and for each benchmark we provide results with various bound functions.

Each point on the graphs in Figures 8.1-8.4 represents an invocation of CHESS with a particular bound function and bound. Lines connect points only for visual clarity – we test only integer bounds. The two dashed lines represent DPOR. DPOR is a single invocation of CHESS that searches the entire state space. Really, DPOR should appear as a single point at 100% of the state space. We provide the horizontal dotted line through this value for visual clarity, and so that DPOR may be easily compared with bounded search results. Note that the DPOR result does not change with the bound or with the bound function.

Smaller values for time are better than larger values in Figures 8.1-8.4 because they indicate that the search required less time to explore more unique states. We



(a) Context bound

(b) Preemption bound

(c) Delta bound

(d) Local context bound

(e) Local depth bound

Figure 8.1: Coverage vs. time as the bound increases for `MRSE`.

show both optimized and unoptimized search results. Optimized results include all of the optimizations described in Chapter 6. We evaluate each optimization individually in Section 8.6.

The `MRSE` test in Figure 8.1 is interesting because it terminates relatively quickly yet the bound increments significantly before saturating. This test may not be an ideal match for BPOR because it terminates quickly with DPOR, so DPOR would search the entire state space quite quickly. Because the search does terminate, however, and the bound increases significantly before saturating, the `MRSE` test provides more insight than other tests do on how the search changes as the bound increases.

Figure 8.1 shows results for `MRSE`, a unit test for a manual reset event primitive. In Figure 8.1, the context bound clearly requires the most overhead, and BPOR provides the least benefit when combined with context-bounded search. This result is intuitive because the context bound is neither stable nor extensible and context-bounded search places conservative backtrack points at each context switch. Note that the bound does not saturate in Figure 8.1(a). We terminate tests that take longer than an hour and the context bound exceeds this limit before it explores the entire state space.

The preemption bound notably improves over the context bound by bringing coverage time closer to DPOR. The optimized context bound explores only about 10% of the state space in the time DPOR requires to search the entire state space. The preemption bound, on the other hand, searches about 40% of the state space before it requires longer than DPOR. Unlike the context bound, the preemption bound saturates. Optimized, preemption-bounded BPOR performs almost exactly as well as DPOR after the bound saturates. The bound optimization is responsible for this result – if the entire state space is reachable within the bound then the search does not add any conservative backtrack points. The same result would occur with

context-bounded search if it were given time to saturate.

Delta-bounded search improves slightly over preemption-bounded search, with about 60% of the state space reached before bounded search requires more time than DPOR. The primary reason the delta bound performs better than the preemption bound on `MRSE` is that the `MRSE` test contains only three threads. The delta bound suffers when it must explore a low priority thread because then it must also conservatively explore all higher priority threads from that state. Two threads do the bulk of the work in `MRSE`, so there is little overhead due to scheduling lower priority threads. The preemption bound, in contrast, must schedule prior to the most recent cheaper transition for each transition that increments the bound, so it requires more conservative backtrack points for this benchmark.

The local delta bound and local context bound do not require any conservative backtrack points and thus never require longer than DPOR, aside from a small fixed overhead to compute the bound. In all cases the total number of local and deadlock states observed with context bound c is equivalent to the number observed with local context bound c, as we would expect if the local context bound accurately encodes the context bound in vector clocks. The same observation applies to the depth bound. Note that the depth bound may not be a useful heuristic because it biases the search toward early portions of the state space as described in Section 2.7.1. The depth bound and local depth bound provide a simple and intuitive proof-of-concept for partial order bounds, however, so we include them here.

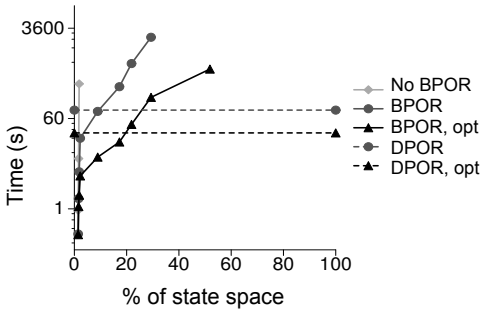
Local depth and local context-bounded search without DPOR perform very poorly. They are far less efficient than their counterpart bounds on the total order. Bounding the partial order guarantees that any local states reachable within bound c will be explored by the search, sometimes by a sequence of transitions whose total order bound is greater than c. As a result, the search explores many sequences of transitions that exceed the corresponding total order bound. Thus, the number of

global states reachable within partial order bound c is larger than the number of global states reachable within the corresponding total order bound c. All of these additional states are redundant, and BPOR prunes the ones it does not explore. Without BPOR, however, the search must explore these additional paths to the same local states.

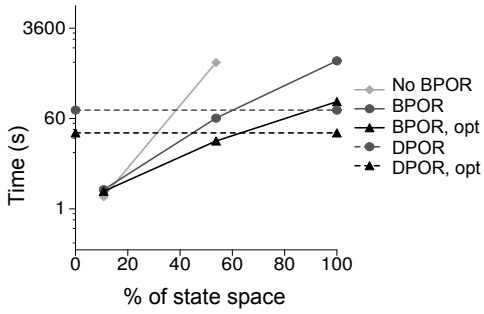
Figure 8.2 shows results for `FFT`, which computes a parallel fast Fourier-transform. These results differ from the `MRSE` results because all conflicting data accesses in `FFT` are protected by acquire and release operations. Additionally, each thread performs only one acquire and release operation, so preempting a thread is not useful. As a result, preemptions are not useful – any new state that can be reached via a preemptive context switch can also be reached via a non-preemptive context switch with very few exceptions. As a result, preemption-bounded search finds almost all states with a very small bound, because almost all context switches are non-preemptive and thus free. The preemption bound incurs notable overhead because `FFT` contains a very large number of release operations, and the preemption bound is conservative with respect to release operations.

The context bound, in contrast, takes longer than the preemption bound does to saturate in Figure 8.2 because context switches are very frequent and they always incur a cost. The context bound suffers relatively little overhead from scheduling all threads when a context switch occurs because most threads must be scheduled there anyway. The context bound also does not require conservative backtrack points prior to release operations, whereas the preemption and delta bounds do.

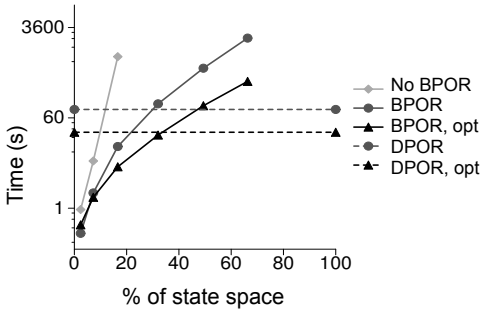
The cost of a transition in delta-bounded search depends upon the enabledness of other threads, so the delta bound backtracks prior to release operations similarly to the preemption bound. These backtrack points hurt partial-order reduction significantly for `FFT` because it contains so many release operations. In this case, none of these conservative backtrack points are necessary. In Section 6.3 we ob-



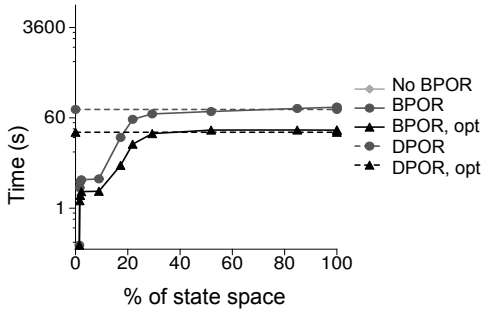
(a) Context bound



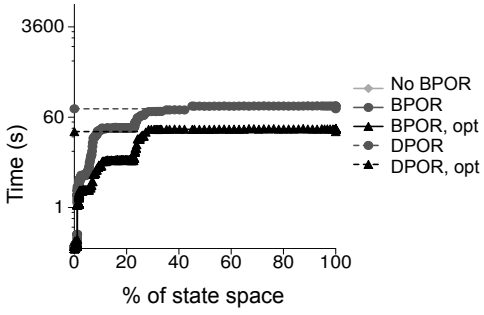
(b) Preemption bound



(c) Delta bound

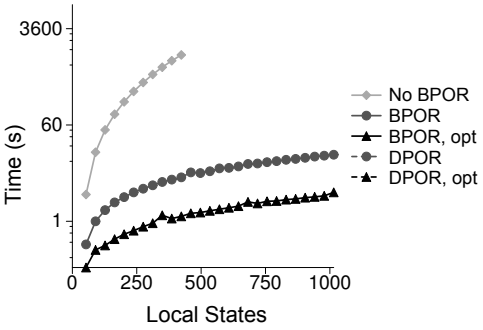


(d) Local context bound



(e) Local depth bound

Figure 8.2: Coverage vs. time as the bound increases for `FFT`.



(a) Fair bound

Figure 8.3: Coverage vs. time as the bound increases for `Fair`.

serve that more aggressively pruning prior to release operations would be beneficial. This example suggests that for some tests, further optimizing the preemption-bound persistent and delta-bound persistent sets prior to release operations would be very effective for reducing the state space.

Figures 8.2(d) and 8.2(e) illustrate characteristics of the state space. `FFT` contains several critical sections protected by locks, so increasing the depth bound during these sections does not make any new states reachable, leaving the flat lines in Figure 8.2(e). Likewise, context switches in these regions do not lead to new states. After the last context switch, the final thread executes for a long time, exposing many new local states, but without any opportunity for context switches because it is the only thread left in the system. Thus, Figures 8.2(d) and 8.2(e) a long, flat tail at the end.

`FFT` shows that results vary significantly with characteristics of the tested program. In particular, release operations affect the results for the preemption, delta, and context bounds significantly. The preemption bound is additionally affected by the frequency with which the threads in a program block.

Figure 8.3 shows state space coverage as the fair bound increments. The `Fair` test is a small microbenchmark that we created because running fair-bounded

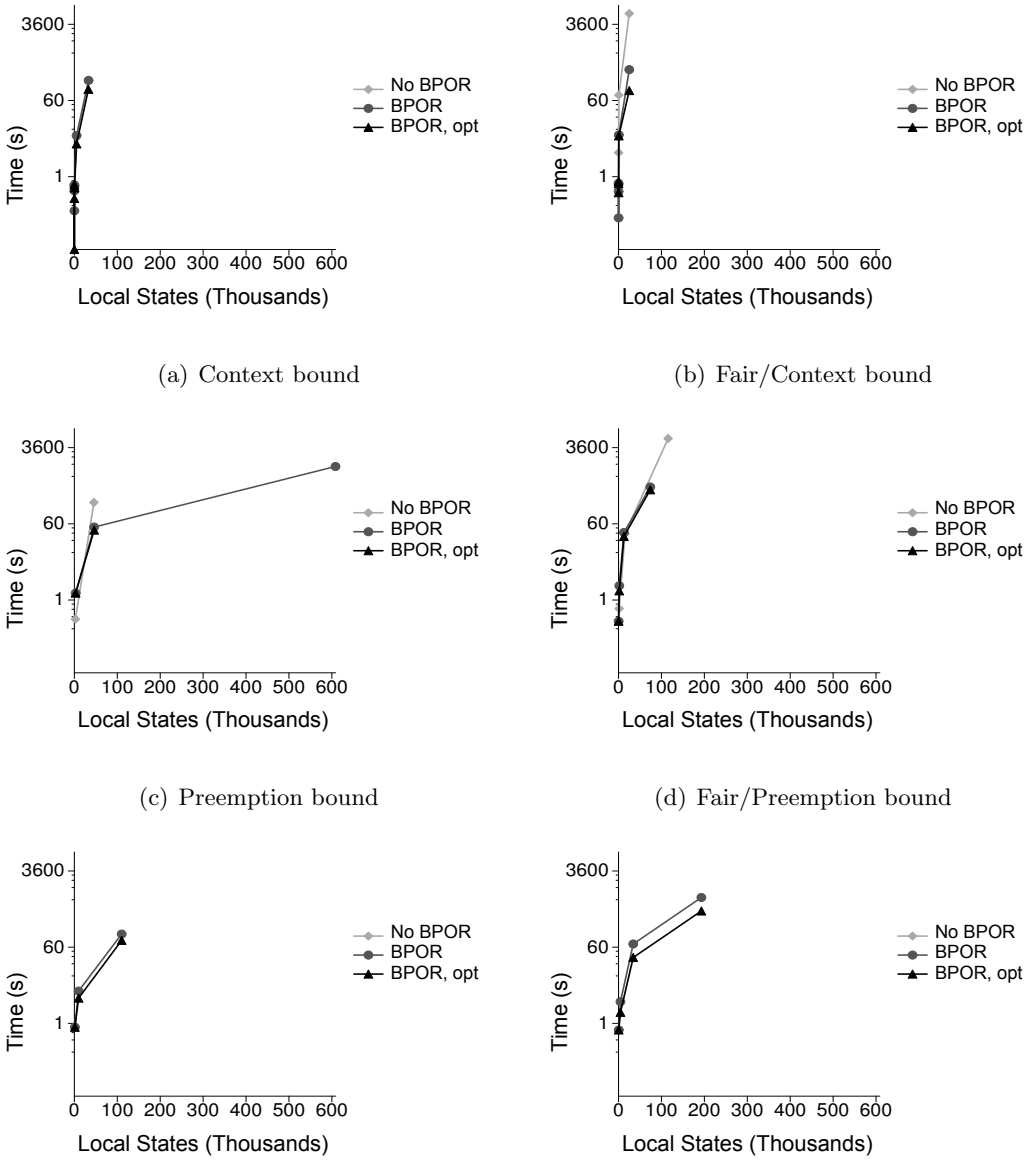
search without partial-order reduction and without any other bound on the search is impractical for large programs. This microbenchmark contains three threads that access shared variables in a loop and wait for one another to modify those variables before exiting their respective loops.

Without the fair bound, CHESS reports a livelock for this program. As the fair bound increases, the size of the state space grows, but not at the same rate that it does with the context, preemption, and delta bounds. The fair bound limits the state space in a very restricted way. It prunes only cycles in the state space, whereas the context, preemption, and delta bounds all prune the state space more generally. Thus, the size of the state space tends to grow in a polynomial fashion with the fair bound, not in an exponential fashion as with other bounds. Each increment of the fair bound unrolls another cycle in the state space that exposes another set of states precisely like the ones explored with the prior bound.

Figure 8.4 combines fair-bounded search with other bounds to demonstrate how their combined effects inhibit partial-order reduction, as discussed in Section 6.6. Note that the x-axis in Figure 8.4 indicates the raw number of local states explored, rather than the percent of the total state space. We chose this metric because none of these searches terminated, so we do not know the total size of the state space.

All of the experiments in Figure 8.4 use a fixed fair bound of two, but vary the context, preemption, or delta bounds. We chose a fixed fair bound of two under the assumption that two iterations through a loop in which a thread only yields the processor is sufficient to explore any meaningful interactions other threads might have with the code in that loop. Additionally, two is the default fairness bound used in CHESS.

We chose the `Exception` test for fair-bounded search because it contains cycles that the fair bound prunes, yet they are not infinite cycles so the search



(a) Context bound

(b) Fair/Context bound

(c) Preemption bound

(d) Fair/Preemption bound

(e) Delta bound

(f) Fair/Delta bound

Figure 8.4: Coverage vs. time as the bound increases for `Exception`.

terminates without a fair bound. Adjacent pairs of graphs in Figure 8.4 are not directly comparable – the context bounded state space and the fair/context bounded state space are not the same state spaces. The size, instructions, and behavior of these state spaces are similar enough, however, that comparing them is interesting.

Each bound prunes the state space less efficiently when combined with the fair bound, which makes sense because the search inserts additional conservative backtrack points. The preemption bound responds the worst by far, however. The context bound does not reason about thread enabledness, so the combined fair and context bound must accumulate the two bounds' respective overheads, but need not insert additional conservative backtrack points. The preemption and delta bounds both require additional conservative backtrack points because they reason about thread enabledness.

8.5 Visited Over Unique Visited States

We measure the number of total visited local states and divide it by the number of unique visited local states to understand the amount of redundant work each search performs. Figures 8.5-8.8 show these results for BPOR without any optimizations, and with all optimizations. The context switch bound visits more duplicate states than the other bounds do – note that Figure 8.5(a) uses a different scale than the other graphs in Figure 8.5 do. The wasted work for unoptimized search increases as the bound increases, but with the optimized search the redundant states decrease as the bound approaches saturation due to the bound optimization. The local depth and local context bounds visit relatively few duplicate states, and the ratio of explored to unique states is consistent as the bound increases because the bound adds no conservative backtrack points.

The `FFT` results in Figure 8.6 reflect the unique nature of this benchmark with respect to the preemption and context bounds. The context bound appears

relatively similar to the `MRSE` graphs, with a slight fluctuation at a context bound of four because previously unexplored, unique portions of the search space first become reachable with four context switches. The delta bound has surprisingly high overhead, particularly without optimizations. The delta bound is particularly ill-suited to the `FFT` benchmark because it contains many acquire and release operations, and the delta bound backtracks all threads prior to release operations. The delta bound does worse than the preemption bound because `FFT` contains many dependences between threads, and whenever a more expensive thread must be backtracked, delta-bounded search backtracks all cheaper threads. This program has seven threads, so backtracking cheaper threads is particularly costly.

Figure 8.7 shows the ratio of states visited to unique states visited with the fair bound but no other bounds. The optimizations are consistently effective for the fair bound because the size of the state space does not grow exponentially with the fair bound as it does for the other bounds. As the size of the state space grows, the amount of wasted work due to conservative backtrack points also grows exponentially with the context, preemption, and delta bounds, so the optimizations appear more effective as the bound increases. With the fair bound, the amount of wasted work grows more slowly with the bound.

Figure 8.8 shows total visited states over unique visited states for `Exception` with and without fairness. Adding the fair bound increases the fraction of duplicate states for each bound function, as expected. The context bound explores an unexpectedly higher fraction of duplicate states with a context bound of three and with a fair/context bound of four. These extra duplicate states are likely due to the search exploring, with three context switches, a set of states that lead to new behavior only with four context switches. After the bound increases, the search can reach those new states so the fraction of redundant states decreases.

8.6 Optimizations

Figures 8.9-8.10 differentiate the optimizations described in Chapter 6. Each set of three bars represents BPOR with a different optimization level. The three bars for each optimization show the ratio of total visited states to unique visited states with three different bound values. The "min" bound is always zero, except for the local depth bound where it is one because the local depth bound is one after the first transition. The "max" value is the highest bound for which we have results for BPOR with no optimizations. The "50%" bound is the first bound with which the search explores at least 50% of the state space. If no such bound exists among our data points then this bound is the ceiling of half of the max bound. The purpose of these three settings is to show how the optimizations' performance varies with the bound. Lower bars mean the optimization is more effective at reducing the state space. The "none" bars provide a comparison point. Note that the y-axis scale is different for each bound.

In Figure 8.9, the bound optimization provides the most significant reductions for context, preemption, and delta bounded search, primarily as the bound approaches saturation. The bound and sleep sets optimizations do not affect the local delta or local context bound because they do not insert any conservative backtrack points. The effects of the optimizations are complementary – their combined reduction is greater than the reduction of any optimization individually.

Figure 8.10 shows optimization results for `FFT`. The local depth and local context bounds benefit from the release optimization because this test contains many acquire and release operations. The alternative thread optimization is particularly important for `FFT` as well because threads block frequently in `FFT`. When a backtracked thread is blocked, the alternative thread optimization BPOR to schedule only one alternative thread rather than conservatively scheduling all threads.

The sleep sets optimization is particularly beneficial to preemption-bounded

search in `FFT`. `FFT` requires relatively few preemptions because the non-preemptive alternatives are typically sufficient. Without the sleep sets optimization, the common backtrack points in `FFT` are never allowed to enter the sleep set and the search re-explores the same state space repeatedly. With the sleep sets optimization, these values may be placed in the sleep set where they substantially reduce the state space.

8.7 Memory

We measure memory use with various configurations to confirm that none of the algorithms that we present incur a significant memory overhead. As expected, the memory use for each configuration quickly normalizes and remains steady at a relatively low level throughout the stateless search. For these tests, we disabled happens-before tracking because storing hashes of visited states does increase memory use over time but its purpose is strictly for bookkeeping to report results.

8.8 Bugs

Table 8.2 shows time required to find known bugs with different CHESS configurations. We choose the preemption bound for these tests because it has been widely used in prior work, and because many of these tests require fairness. We do not combine the local depth or local context bound with fairness, so we do not show bug results for those bounds. The `MRSE` and `CCR` tests are acyclic so the search terminates without the fair bound. We show results for these tests with DPOR, preemption-bounded search with no BPOR, and preemption-bounded search with BPOR. BPOR finds these bugs up to 7x faster than DPOR and up 4x faster than preemption-bounded search.

The `NQueens`, `Matrix` and `RegOwn` tests require fairness. We do not provide results for these tests with DPOR because DPOR does not handle cyclic state

Unit test	Bug	Time to manifest bug (s)			
		DPOR	No BPOR	BPOR	
			Pb		Pb
MRSE	Deadlock	2	6		1
CCR	Assertion	69	39		9
	Assertion	64	35		8
			Fb_Pb	Fb	Fb_Pb
NQueens	Assertion	-	75	5	4
	Livelock	-	3235	502	125
	Assertion	-	312	80	11
Matrix	Assertion	-	54	2	2
	Livelock	-	1089	787	137
	Livelock	-	-	694	136
RegOwn	Exception	-	-	3474	1586

Table 8.2: Time required to find bugs. Preemption/fair bounded search without BPOR requires much longer than fair bounded BPOR requires. Fair-bounded BPOR requires longer than preemption/fair-bounded BPOR.

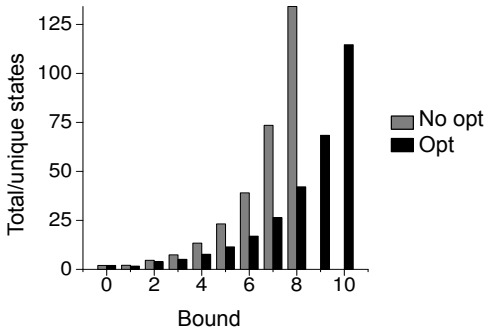
spaces and thus cannot detect these bugs. Fair-bounded search manifests these bugs between 1.3x faster and 27x faster with BPOR than it does without. Fair, preemption-bounded search finds each bug faster than fair-bounded search, aside from the assertion failure in `Matrix` that both searches find very quickly.

The second livelock in `Matrix` and the exception in `RegOwn` both did not manifest within three hours without BPOR. We therefore do not provide results for these configurations. These results motivate combining fair-bounded search with partial order bounds, or creating a local fair bound, if possible. With a fairness criterion the local depth and local context bounds would be far more powerful.

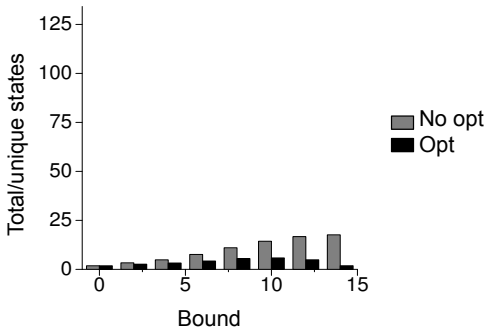
8.9 Discussion

Without partial-order reduction bounded search quickly becomes impractical as the bound increases. BPOR reduces the state space for bounded search considerably,

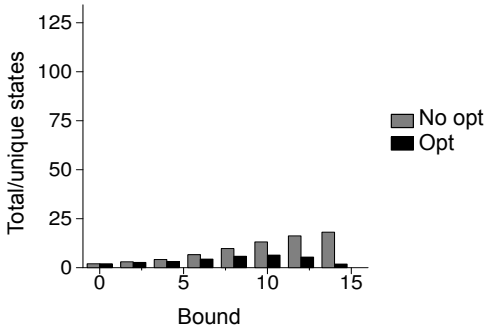
but still requires longer than DPOR to search only a portion of the state space as the bound increases. The optimizations in Chapter 6 reduce the state space further, but BPOR still performs considerably more



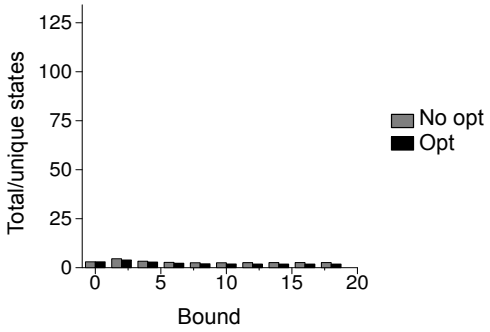
(a) Context bound



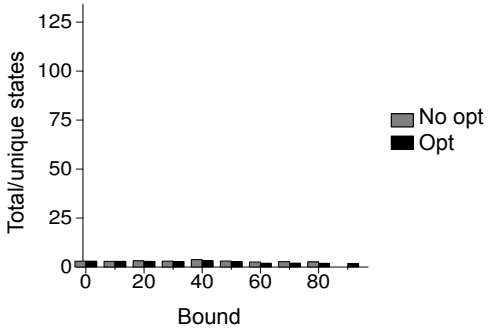
(b) Preemption bound



(c) Delta bound

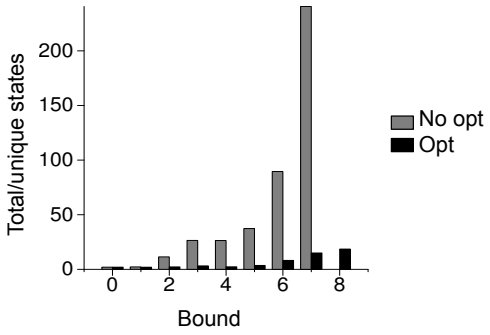


(d) Local context bound

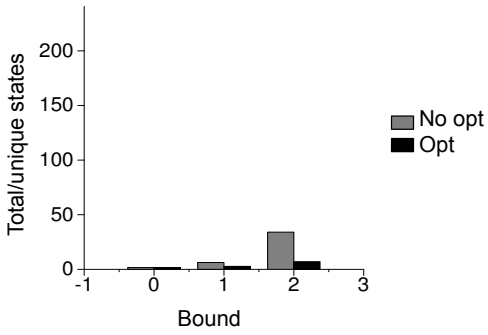


(e) Local depth bound

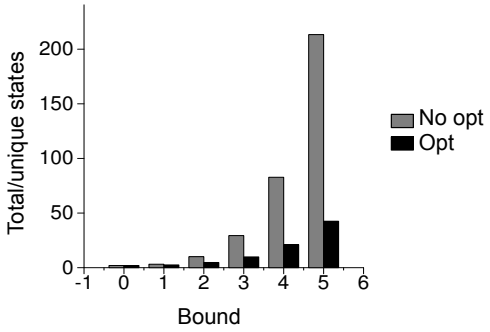
Figure 8.5: Visited over unique visited states with and without optimizations for `MRSE`.



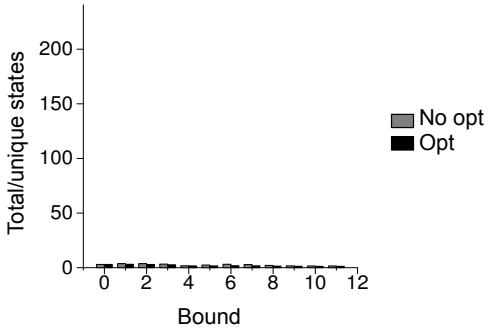
(a) Context bound



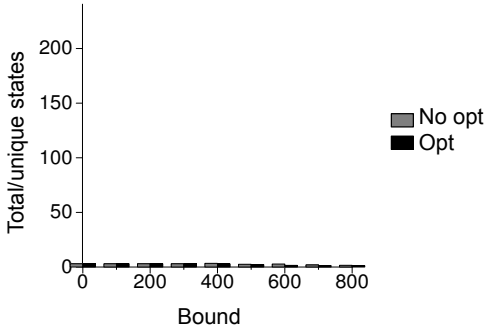
(b) Preemption bound



(c) Delta bound

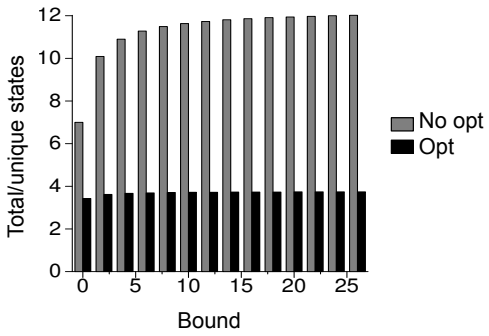


(d) Local context bound



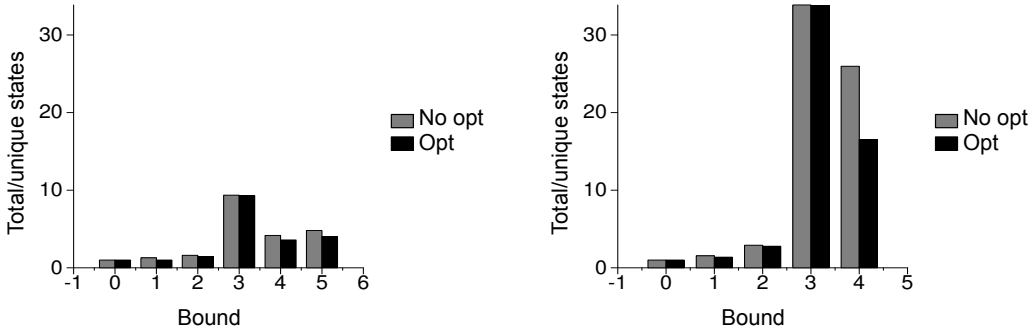
(e) Local depth bound

Figure 8.6: Visited over unique visited states with and without optimizations for `FFT`.



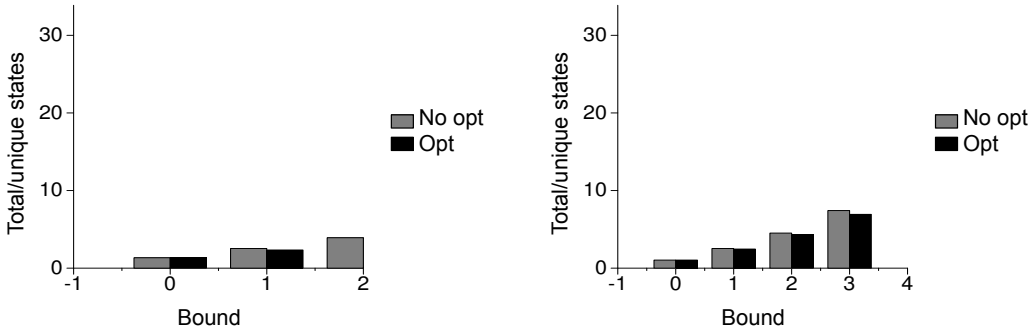
(a) Fair bound

Figure 8.7: Visited over unique visited states with and without optimizations for `Fair`.



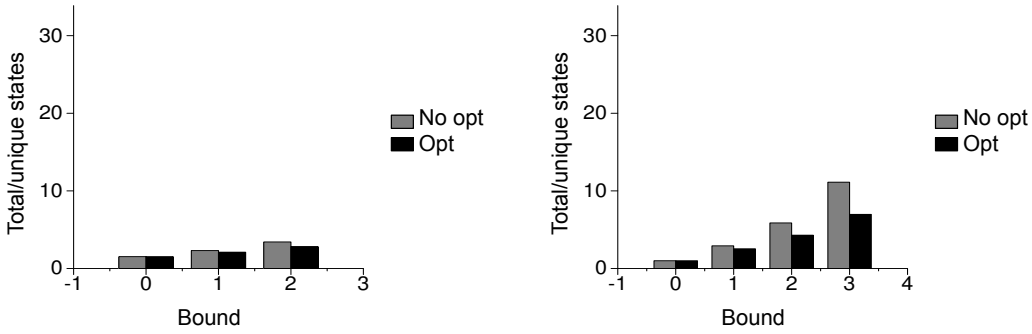
(a) Context bound

(b) Fair/Context bound



(c) Preemption bound

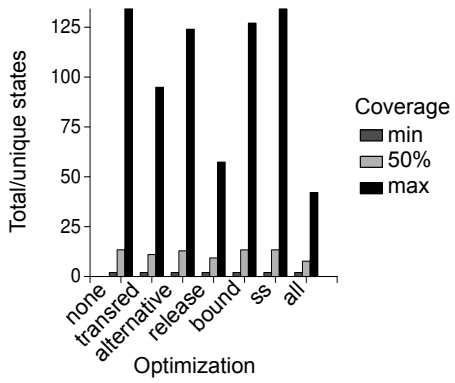
(d) Fair/Preemption bound



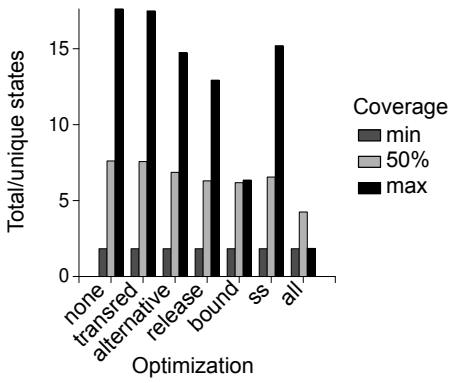
(e) Delta bound

(f) Fair/Delta bound

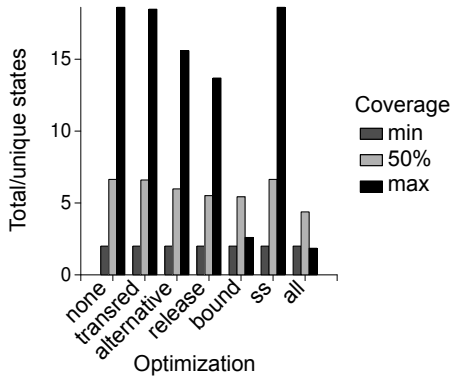
Figure 8.8: Visited over unique visited states with and without optimizations for `Exception`.



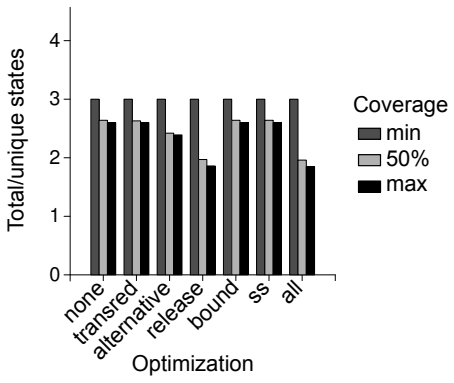
(a) Context bound



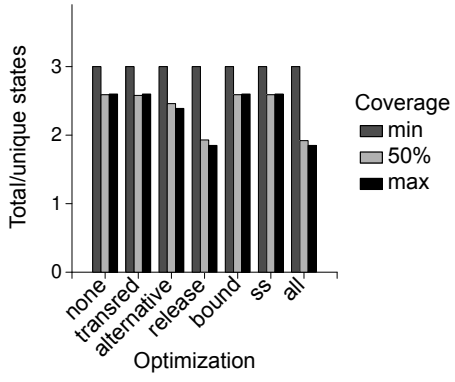
(b) Preemption bound



(c) Delta bound

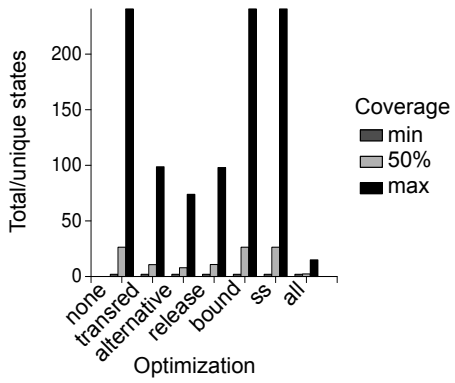


(d) Local context bound

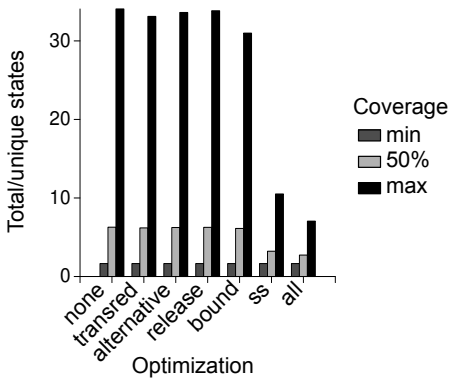


(e) Local depth bound

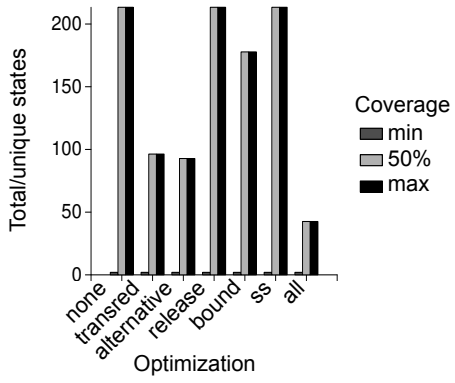
Figure 8.9: Visited over unique visited states with each optimization for `MRSE`.



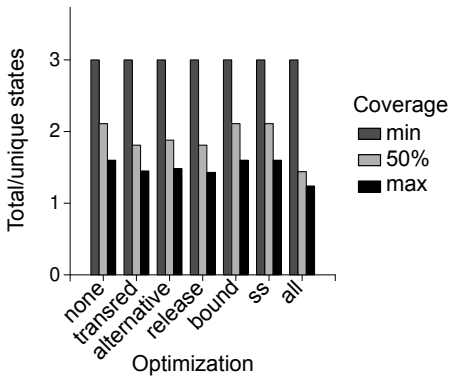
(a) Context bound



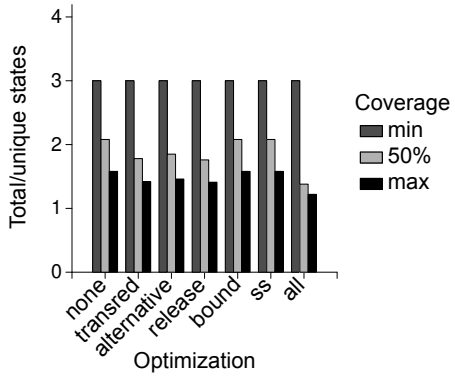
(b) Preemption bound



(c) Delta bound



(d) Local context bound



(e) Local depth bound

Figure 8.10: Visited over unique visited states with each optimization for `FFT`.

Chapter 9

Future Work

This thesis shows that partial-order reduction can be combined with bounded search to provide bounded coverage with a reduced state space. The high-level goal for this work is to make model checking concurrent programs more practical and more useful for testers so that they will be more likely to use it. This section discusses several opportunities these results create, as well as variations on the ideas presented in this thesis that might prove interesting for future exploration.

9.1 Other Bounds

One goal of bounded search is to provide a useful and incremental coverage metric to testers. We explore various bounds in this thesis, but other bounds may also prove useful. The examples in the previous sections show how tool designers might combine other bounds with partial-order reduction. We describe bounds that appear particularly appealing.

9.1.1 Partial-order Bounds

Chapter 7 provides two examples of partial-order bounds and the results are compelling that any bound should be associated with a local state rather than with the total order on the program's transitions. Other intuitive bounds such as the preemption bound would also be useful as partial-order bounds. The preemption and delta bounds both likely introduce additional complexity because they reason about the enabledness of other threads. Handling this problem might be an interesting challenge.

To test cyclic state spaces, the search must include some fairness criterion. Expressing the fairness bound as a partial-order bound would thus prove very useful. The fairness bound, like the preemption and delta bounds, reasons about the enabledness of other threads. The fairness bound we present is just one example of a fairness criterion, however. Other fair bounds might prove more useful, or better able to combine with partial-order reduction. A fair bound that provides *strong fairness* might prove both more useful as a fairness bound, and better able to combine with partial-order reduction [Apt et al., 1988]. An extensible fairness bound would make bounded search far more effective.

In addition to the bounds that we discuss in this thesis, there may exist other bounds that programmers would find particularly useful. One goal of both the context and the preemption bound is that the bound correlates in some way with the complexity of the bug. Other bounds that more explicitly provide this advantage would be useful.

9.1.2 Bug Depth Bound

Prior work defines the *bug depth* for a bug to be the minimum number of constraints that must be placed on the program to guarantee that the bug will manifest [Burckhardt et al., 2010]. Prior work uses the bug depth to provide statistical guarantees,

rather than exhaustively search the state space. Incorporating this same concept into a bound for exhaustive, systematic search would be useful because the bug depth bound is innately tied to the complexity of the bug. The bug depth is a property of the bug, not of the total order on instructions – really, it is a property of a minimal subset of the partial order that is sufficient to guarantee the bug manifests. As a result, this bound may translate well into an extensible bound function.

Bug depth-bounded search has several different properties from regular bounded search. Prior work uses thread priorities to control the order in which transitions execute. These thread priorities could control the schedule in exhaustive search, as well. Rather than backtracking each thread to a given state, however, bug depth-bounded search enumerates new states by lowering the priority of the executing thread to the lowest priority. Thus, bug depth-bounded search allows *all* other threads to execute prior to the executing thread, rather than one particular thread, and as a result it explores more new orderings with each new backtrack point. This search strategy is different from the basic BPOR approach, however, because it does not enumerate all of the threads in a bound persistent set in each state. This bound requires more notable changes to search implementations than other bounds, but it may be worthwhile because it is tied to a fundamental property of the bug, and it may combine well with partial-order reduction.

9.2 Parallel Search

The state space for exhaustive search is exponential even with BPOR, so parallel search is critical for exploring the state space efficiently. In addition, BPOR has very large per-execution overhead, at least two orders of magnitude, so parallelizing each execution might be useful, particularly for testing embarrassingly parallel programs with very large numbers of threads. We discuss each of these approaches to parallelizing BPOR.

9.2.1 Exploring Executions in Parallel

The most straight-forward way to parallelize the search is to run multiple executions in parallel while leaving each individual execution serialized. Because the search space is exponential and each individual execution is independent of each other execution, the problem is embarrassingly parallel. The only shared data among different executions is the set of backtracking points that the search must explore in the future, which should not significantly hinder parallelization because they may be written repeatedly by different executions without affecting correctness.

A parallel search could use many heuristics to divide the state space among processors. By targeting parallel threads at different portions of the state space the search might be able to find bugs more quickly. Prior work shows that randomly assigning parallel threads to different portions of the state space finds bugs effectively with Java PathFinder [Dwyer et al., 2007], so a similar approach would likely be effective for BPOR. Alternatively, by systematically running the search in parallel in disparate parts of the state space, the search might achieve coverage more quickly and also be more likely to find bugs quickly.

9.2.2 Parallelizing Each Execution

In addition to running many different tests in parallel, using concurrency within each individual test might also be possible. For some workloads, particularly those that contain an exceptionally large number of threads, serializing the execution is not practical. In this work, we focus on small unit tests where bugs are as likely to manifest with a small number of threads as they are with a very large number of threads. Some workloads rely on very large numbers of threads for their behavior to be interesting, however, and these programs would be more impractical with BPOR than smaller unit tests.

The bug depth bound [Burckhardt et al., 2010], described in Section 9.1.2,

might allow limited concurrency within each individual test. The bug depth bound requires that certain transitions be ordered, but all other transitions can execute in any order. Because the bug depth bound bounds a *segment* of the partial order on a program's transitions, rather than the entire partial order on those transitions, it does not matter in what order the remaining transitions execute, even if they are dependent with one another. Massively parallel programs with many threads could definitely exploit this advantage to run some large subset of those threads in parallel while exploring smaller bug depth bounds.

As the bug depth bound increases, more threads must be totally ordered and the program becomes less parallel. Prior work suggests that many bugs manifest with relatively small bug depth bounds, however [Burckhardt et al., 2010, Nagarakatte et al., 2012]. Thus, a parallel bug depth bounded search might be very effective for finding bugs while providing limited correctness guarantees. Because these correctness guarantees would correspond with the complexity of the associated bugs, they might be more compelling than other incremental guarantees.

9.3 Exploiting the Bound

In this work, we bound the search primarily to provide an incremental coverage guarantee, but the program or runtime system may be able to leverage these incremental guarantees. The incremental guarantee is a progress metric and a tool for testers to reason about their program's correctness. If the runtime system guarantees these bounds in practice, however, then the incremental guarantee becomes a guarantee for the entire state space that is reachable in practice.

Consider a concurrent system where the thread scheduler guarantees that all executions will be fair, for some definition of fair. By modifying fair-bounded search to explore all such fair executions, fair-bounded search can provide full coverage for the executions that system will see in practice. Similarly, if a system provides

guarantees regarding how frequently it will preempt the executing thread, then BPOR might be able to guarantee correctness while searching only a portion of the state space.

Bounded search might also provide insights regarding the types of restrictions on the system that would make correctness easier to achieve. Prior work makes each execution in a concurrent system deterministic to help prevent concurrency errors and to make them easier to debug [Devietti et al., 2009, Olszewski et al., 2009, Lucia and Ceze, 2013]. These systems inhibit performance as well. Bounded search might shed light on which properties of the program are most correlated with bugs so that the system could restrict those properties and prevent bugs from occurring in practice.

9.4 Exploiting Modularity

We focus on unit tests for concurrency libraries in this work because concurrency libraries should provide a backbone on which future parallel systems can be built. Correctness is thus paramount for these concurrency libraries, and their unit test are relatively small so exhaustive search is practical. We exhaustively reorder all accesses to shared data. We hope systems that leverage these concurrency libraries can exploit their guarantees and that those guarantees will make verifying larger programs more practical.

In prior work, we investigated *preemption sealing*, in which the thread scheduler is disabled from preempting the executing process to eliminate portions of the state space [Ball et al., 2010]. We used preemption sealing to detect multiple errors in programs by sealing the preemption that leads to an error. We also used preemption sealing for compositional testing. If the search can trust lower-level modules to be atomic, then it can seal preemptions in those modules, preserve coverage, and reduce testing time significantly. Extending this modular testing framework and

building it on top of BPOR would make it possible to provide guarantees for much larger programs.

Chapter 10

Conclusions

Concurrent software is notoriously difficult to test and debug, but dynamic, stateless model checkers offer a promising solution to this problem. Dynamic, bounded partial-order reduction for stateless model checking finds bugs quickly and reproducibly, and it incrementally guarantees coverage. This result advances the state of the art for debugging concurrent programs.

This thesis extensively analyzes the space of possible dynamic, bounded partial-order reduction strategies. We show that bounded search alone is insufficient to provide reasonable coverage guarantees. Without partial-order reduction bounded search wastes too much time exploring redundant states. Dynamic partial-order reduction prunes the state space significantly, but it does not combine easily with cyclic state spaces or bounded search, and it does not provide any incremental guarantees. If the search does not terminate, it provides no useful guarantee at all.

Combining bounded search with dynamic partial-order reduction offers clear benefits, as our experiments with best-first search demonstrate. We combine bounded search with dynamic partial-order reduction by identifying dependences that the bound introduces. We sacrifice partial-order reduction to preserve bounded coverage and prove the resulting algorithm correct for each bound function. This algorithm

preserves bounded coverage, but it limits partial-order reduction significantly.

We evaluate the depth, context, preemption, delta, and fair bounds when combined with dynamic partial-order reduction. Each of these bounds offers insight about the nature of the search. These insights motivate *partial-order bounds*, which bound the partial-order on a program's transitions rather than the total order on those transitions. We show that partial-order reduction does not sacrifice bounded coverage for bounds on the partial order. Partial-order bounds that also provide useful coverage metrics for testers make software model checking for concurrent programs more efficient and more useful. The bounds we introduce manifest bugs an order of magnitude more quickly than previous approaches and guarantee incremental coverage in minutes or hours rather than weeks, helping developers find and reproduce concurrency errors. The algorithms we describe advance the state of the art for stateless model checking for concurrent programs by finding bugs quickly and providing systematic, incremental coverage guarantees.

Bibliography

[Aggarwal et al., 1990] Aggarwal, S., Courcoubetis, C. A., and Wolper, P. L. (1990). Adding liveness properties to coupled finite-state machines. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 12, pages 303–339.

[Apt et al., 1988] Apt, K. R., Francez, N., and Katz, S. (1988). Appraising fairness in languages for distributed programming. 2:226–241.

[Ball et al., 2010] Ball, T., Burckhardt, S., Coons, K. E., Musuvathi, M., and Qadeer, S. (2010). Preemption sealing for efficient concurrency testing. In *The* 16^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

[Biere et al., 1999] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. (1999). *Symbolic model checking without BDDs*. Springer.

[Bosnacki et al., 2006] Bosnacki, D., Leue, S., and Lluch-Lafuente, A. (2006). Partial-order reduction for general state exploring algorithms. In *SPIN*, pages 271–287.

[Burch et al., 1990] Burch, J., Clarke, E., McMillan, K., Dill, D., and Hwang, L. (1990). Symbolic model checking: 10^{20} states and beyond.

[Burckhardt et al., 2010] Burckhardt, S., Kothari, P., Musuvathi, M., and Nagarakatte, S. (2010). A randomized scheduler with probabilistic guarantees of

finding bugs. In *Proceedings of the fifteenth annual conference on architectural support for programming languages and operating systems*, pages 167–178, New York, NY, USA. ACM.

[Cimatti et al., 2002] Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. (2002). Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 359–364. Springer.

[Clarke and Emerson, 1981] Clarke, E. M. and Emerson, E. A. (1981). Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, London, UK. Springer-Verlag.

[Coons et al., 2010] Coons, K. E., Musuvathi, M., and Burckhardt, S. (2010). Gambit: Effective unit testing for concurrent libraries. In *The* 15^{th} *ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP 2010)*.

[Devietti et al., 2009] Devietti, J., Lucia, B., Ceze, L., and Oskin, M. (2009). Dmp: Deterministic shared memory multiprocessing. In *In Proceedings of the* 14^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*.

[Dwyer et al., 2007] Dwyer, M. B., Elbaum, S., Person, S., and Purandare, R. (2007). Parallel randomized state-space search. In *International Conference on Software Engineering (ICSE)*, pages 3–12, Washington, DC, USA. IEEE Computer Society.

[Edelkamp and Jabbar, 2006] Edelkamp, S. and Jabbar, S. (2006). Large-scale directed model checking ltl. In *SPIN Workshop on Model Checking of Software*, pages 1–18. Springer.

[Edelkamp et al., 2001] Edelkamp, S., Lafuente, A. L., and Leue, S. (2001). Directed explicit model checking with hsf-spin. In *SPIN Workshop on Model Checking of Software*, pages 57–79. Springer-Verlag.

[Edelkamp et al., 2004] Edelkamp, S., Leue, S., and Lluch-Lafuente, A. (2004). Directed explicit-state model checking in the validation of communication protocols. *International journal on software tools for technology transfer*, 5(2-3):247–267.

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

[Emmi et al., 2011] Emmi, M., Qadeer, S., and Rakamaric, Z. (2011). Delay-bounded scheduling. In *ACM SIGACT-SIGPLAN Principles of Programming Languages (POPL)*.

[Fidge, 1988] Fidge, C. J. (1988). Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, volume 10, pages 56–66.

[Flanagan and Godefroid, 2005] Flanagan, C. and Godefroid, P. (2005). Dynamic partial-order reduction for model checking software. In *ACM SIGPLAN-SIGACT Principles of Programming Languages (POPL)*, pages 110–121.

[Flanagan and Godefroid, 2011] Flanagan, C. and Godefroid, P. (2011). Addendum to dynamic partial-order reduction for model checking software.

[Godefroid, 1990] Godefroid, P. (1990). Using partial orders to improve automatic verification methods. In *Proceedings of the* 2^{nd} *International Workshop on Computer-Aided Verification (CAV '90)*, pages 176–185.

[Godefroid, 1996] Godefroid, P. (1996). *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag.

[Godefroid, 1997] Godefroid, P. (1997). Model checking for programming languages using Verisoft. In *ACM SIGACT-SIGPLAN Principles of Programming Languages (POPL)*, pages 174–186.

[Godefroid and Khurshid, 2002] Godefroid, P. and Khurshid, S. (2002). Exploring very large state spaces using genetic algorithms. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–280. Springer.

[Godefroid and Pirottin, 1993] Godefroid, P. and Pirottin, D. (1993). Refining dependencies improves partial-order verification methods (extended abstract). In *International Conference on Computer Aided Verification (CAV)*, pages 438–449, London, UK. Springer-Verlag.

[Godefroid and Wolper, 1992] Godefroid, P. and Wolper, P. (1992). Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Computer Aided Verification (CAV '91)*, pages 332–342.

[Godefroid and Wolper, 1994] Godefroid, P. and Wolper, P. (1994). A partial approach to model checking. In *Information and Computation*, pages 406–415.

[Groce and Visser, 2002] Groce, A. and Visser, W. (2002). Model checking java programs using structural heuristics. *SIGSOFT Software Engineering Notes*, 27(4):12–21.

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

[Holzmann, 1997] Holzmann, G. (1997). The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295.

[Holzmann et al., 1992] Holzmann, G. J., Godefroid, P., and Pirottin, D. (1992). Coverage preserving reduction strategies for reachability analysis. In *In proceedings of the* 12^{th} *International Symposium on Protocol Specification, Testing, and Verification*, pages 349–363.

[Joshi et al., 2009] Joshi, P., Naik, M., Park, C.-S., and Sen, K. (2009). Calfuzzer: An extensible active testing framework for concurrent programs. In *CAV '09: Proceedings of the* 21^{st} *International Conference on Computer Aided Verification*, pages 675–681, Berlin, Heidelberg. Springer-Verlag.

[Katz and Peled, 1992] Katz, S. and Peled, D. (1992). Defining conditional independence using collapses. In *Theoretical Computer Science*, volume 101, pages 337–359. Elsevier Science Publishers.

[Korf et al., 2005] Korf, R. E., Zhang, W., Thayer, I., and Hohwald, H. (2005). Frontier search. *Journal of the ACM*, 52(5):715–748.

[Leven et al., 2004] Leven, P., Mehler, T., and Edelkamp, S. (2004). Directed error detection in c++ with the assembly-level model checker steam. In *Model Checking Software*, pages 39–56. Springer.

[Lucia and Ceze, 2013] Lucia, B. and Ceze, L. (2013). Cooperative empirical failure avoidance for multithreaded programs. In *In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)*, pages 39–50.

[Mazurkiewicz, 1986] Mazurkiewicz, A. (1986). Trace theory. In *Advances in Petri nets 1986, part II on Petri nets: applications and relationships to other models of concurrency*, pages 279–324. Springer-Verlag.

[McMillan, 1992] McMillan, K. L. (1992). *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, Pittsburgh, PA, USA. UMI Order No. GAX92-24209.

[Musuvathi and Qadeer, 2007a] Musuvathi, M. and Qadeer, S. (2007a). Iterative context bounding for systematic testing of multithreaded programs. In *Programming Language Design and Implementation (PLDI)*, pages 446–455.

[Musuvathi and Qadeer, 2007b] Musuvathi, M. and Qadeer, S. (2007b). Partial-order reduction for context-bounded state exploration. Technical Report MSR-TR-2007-12, Microsoft Research.

[Musuvathi and Qadeer, 2008] Musuvathi, M. and Qadeer, S. (2008). Fair stateless model checking. In *Programming Language Design and Implementation (PLDI)*, pages 362–371.

[Musuvathi et al., 2009] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, A., and Neamtiu, I. (2009). Finding and reproducing heisenbugs in concurrent programs. In *USENIX Symposium on Operating Systems Design and Implementation.*

[Musuvathi et al., 2002] Musuvathi, M. S., Park, D., Park, D. Y. W., Chou, A., Engler, D. R., and Dill, D. L. (2002). Cmc: A pragmatic approach to model checking real code. In *In The Fifth Symposium on Operating Systems Design and Implementation (OSDI)*.

[Nagarakatte et al., 2012] Nagarakatte, S., Burckhardt, S., Martin, M. M. K., and Musuvathi, M. (2012). Multicore acceleration of priority-based schedulers for concurrency bug detection. In *Programming Language Design and Implementation (PLDI)*.

[Olszewski et al., 2009] Olszewski, M., Ansel, J., and Amarasinghe, S. (2009). Kendo: Efficient deterministic multithreading in software. In *In Proceedings of the* 14^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*.

[Overman, 1981] Overman, W. T. (1981). *Verification of concurrent systems: function and timing*. PhD thesis.

[Park et al., 2009] Park, S., Lu, S., and Zhou, Y. (2009). Ctrigger: exposing atomicity violation bugs from their hiding places. In *Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 25–36.

[Pearl, 1984] Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

[Peled, 1993] Peled, D. (1993). All from one, one for all: on model checking using representatives. In *Proceedings of the* 5^{th} *International Conference on Computer Aided Verification*.

[Peled, 1994] Peled, D. (1994). Combining partial order reductions with on-the-fly model checking. pages 377–390. Springer-Verlag.

[Rungta and Mercer, 2009] Rungta, N. and Mercer, E. G. (2009). Guided model checking for programs with polymorphism. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation (PEPM '09)*, pages 21–30, New York, NY, USA. ACM.

[Russell and Norvig, 2003] Russell, S. J. and Norvig, P. (2003). *Artificial intelligence : a modern approach*. Prentice Hall, 2nd edition.

[Sen, 2007] Sen, K. (2007). Effective random testing of concurrent programs. In *ASE*, pages 323–332.

[Sen, 2008] Sen, K. (2008). Race directed random testing of concurrent programs. In *PLDI*, pages 11–21.

[Valmari, 1990] Valmari, A. (1990). A stubborn attack on state explosion. In *Proceedings of the* 2^{nd} *International Workshop on Computer Aided Verification (CAV '90)*, pages 156–165. Springer-Verlag.

[Visser et al., 2000a] Visser, W., Havelund, K., Brat, G., and Park, S. (2000a). Java PathFinder - second generation of a Java model checker. In *Proceedings of Post-CAV Workshop on Advances in Verification*.

[Visser et al., 2000b] Visser, W., Havelund, K., Brat, G., and Park, S. (2000b). Model checking programs. In *Automated Softeware Engineering Journal*, pages 3–12.

[Yang and Dill, 1998] Yang, C. H. and Dill, D. L. (1998). Validation with guided search of the state space. In *DAC '98*, pages 599–604.

Vita

Katherine Elizabeth Coons is the youngest of three children born to Carol and T.A. Coons. She received the Bachelor of Science degree in Computer Science from the University of Virginia in 2005 with a minor in Engineering Business. She entered the Ph.D. program at the University of Texas at Austin in 2005 and earned the Master of Science degree in Computer Science from the University of Texas at Austin in 2008.

Permanent Address: 20 Descanso Drive Unit 1233
San Jose, CA 95134

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.