Bounded Partial-Order Reduction

Katherine E. Coons* Madanlal Musuvathi† Kathryn S. McKinley*†

The University of Texas at Austin* Microsoft Research†

Abstract

Eliminating concurrency errors is increasingly important as systems rely more on parallelism for performance. Exhaustively exploring the state-space of a program's thread interleavings finds concurrency errors and provides coverage guarantees, but suffers from exponential *state-space explosion*. Two prior approaches alleviate state-space explosion. (1) *Dynamic partial-order reduction* (DPOR) provides *full coverage* and explores only one interleaving of independent transitions. (2) *Bounded search* provides *bounded coverage* by enumerating interleavings that do not exceed a bound. In particular, we focus on preemption-bounding. Combining partial-order reduction with preemption-bounding had remained an open problem.

We show that preemption-bounded search explores the same partial orders repeatedly and consequently explores more executions than unbounded DPOR, even for small bounds. We further show that if DPOR simply uses the preemption bound to prune the state space as it explores new partial orders, it misses parts of the state space reachable in the bound and is therefore unsound. The bound essentially induces dependences between otherwise independent transitions in the DPOR state space. We introduce Bounded Partial Order Reduction (BPOR), a modification of DPOR that compensates for bound dependences. We identify properties that determine how well bounds combine with partial-order reduction. We prove sound coverage and empirically evaluate BPOR with preemption and fairness bounds. We show that by eliminating redundancies, BPOR significantly reduces testing time compared to bounded search. BPOR's faster incremental guarantees will help testers verify larger concurrent programs.

OOPSLA'13, October 29–31, 2013, Indianapolis, IN, USA.
Copyright © 2013 ACM 978-1-4503-2374-1/13/10…$15.00.
http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2509136.2509556

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Algoirthms, Verification

Keywords bounded partial-order reduction, dynamic partial-order reduction, concurrency, fairness, liveness, model checking, shared-memory programs, software testing

1. Introduction

Concurrency errors are notoriously difficult to debug because they often occur only under unexpected thread interleavings. One approach to identify and reproduce concurrency errors is *stateless model checking* [8], which systematically drives the program along possible thread interleavings. This approach is limited in practice, however, by *state-space explosion*. The number of possible thread interleavings grows exponentially with the size of the program and number of threads. Two existing approaches alleviate state-space explosion: partial-order reduction and bounded search.

Partial order methods provide full coverage by exploring all unique reachable states, and they alleviate state space explosion by exploring only one interleaving of *independent* events [6, 7]. Independent events commute – their interleavings do not change program behavior. *Dynamic* partial-order reduction (DPOR) observes the program's actual dependences at runtime, rather than a static conservative estimate of dependences, to identify independent events [6]. In practice, the reduced state space is much smaller than the full state space. Searching this state space is sufficient to verify safety properties such as absence of deadlocks and adherence to local assertions [7]. The size of this reduced space still grows exponentially, but with the number of *dependent* events in the program, rather than the total number of events. Nevertheless, for larger programs DPOR often runs for longer than developers are willing to wait.

Bounded search, in contrast, alleviates state-space explosion by pruning executions that exceed a bound [5, 12, 14]. For example, Musuvathi and Qadeer use preemption-bounded search [12] to explore executions involving a small number of preemptions. Their insight is that many concurrency bugs require a small number of preemptions to manifest. Therefore, by focusing on a much smaller portion of

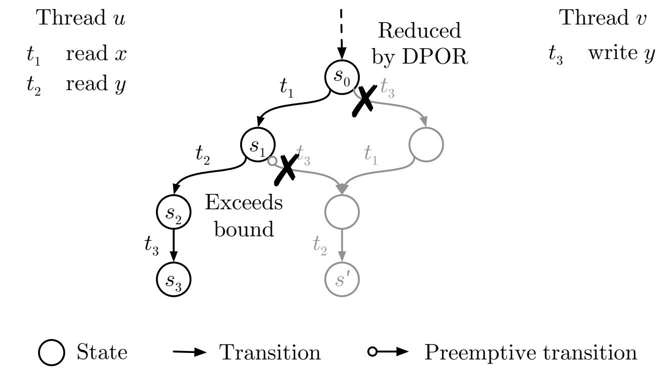


Figure 1. Preemption-bounded search with DPOR. Although s' is reachable within the bound, the search never reaches it.

the state space, preemption-bounded search often finds bugs faster than unbounded search does. At the same time, such a search provides a useful coverage metric to the user — when a search with k preemptions has finished, any remaining bug requires at least $k+1$ preemptions, and as such is more complex to find and less likely to happen in practice [12].

Our experiments show that preemption-bounded search is useful only for very small bounds. For example, searches that allow more than two preemptions for our benchmarks explore so many redundant executions of the same partial order that they explore many more executions than unbounded DPOR. Thus, preemption-bounded search fails to improve performance precisely on those programs where it holds promise - for large programs where DPOR either runs longer than developers are willing to wait, or does not terminate.

This result motivates combining preemption-bounded search with DPOR. Figure 1 illustrates why combining these two approaches has remained an open problem until now. The figure shows the state space for a program with two threads u and v and three instructions. Each transition in the state space corresponds to a thread executing an instruction. Starting from the initial state s_0, the transitions t_1 of thread u and t_3 of thread v are independent as they access different memory locations. Thus, executing the transitions in either order reaches the same state. However, transitions t_2 and t_3 are dependent, leading to two different final states, s_3 and s'. Note, both final states are reachable with no preemptions, respectively through executions t_1, t_2, t_3 and t_3, t_1, t_2. However, a naïve combination of DPOR with preemption bounding can miss exploring s'.

Consider combining DPOR with preemption-bounded search with bound 0. Assume the search first explores execution t_1, t_2, t_3, as illustrated in the figure. The DPOR algorithm achieves state-space reduction by carefully inserting backtrack points only where necessary to explore both orderings of dependent transitions. In the example, because t_3 and t_2 are dependent, DPOR schedules t_3 at s_1, prior to dependent transition t_2, but not at s_0 since t_3 and t_1 are independent. However, executing t_3 in thread v from s_1 requires preempting thread u since it is still enabled in s_1. Since the preemption-bound is 0, the algorithm will not explore this execution – it exceeds the bound. This naïve combination is therefore unsound.

This paper proposes an efficient algorithm called Bounded Partial Order Reduction (BPOR) to soundly combine DPOR with preemption-bounded search. The key idea behind BPOR is to *conservatively* add more backtrack points beyond what DPOR requires. If a backtrack point required by DPOR requires a preemption, then BPOR conservatively backtracks to a previous point in the execution where the backtracked transition does not require a preemption. In the example in Figure 1, when DPOR schedules t_3 at s_1, BPOR additionally schedules t_3 at s_0. Subsequently, BPOR will prune the execution that exceeds the bound at s_1, but not at s_0, eventually exploring both final states.

In this paper, we prove that BPOR with the preemption-bound is sound. In particular, given a preemption-bound, when BPOR terminates it guarantees that it explored all states reachable within the bound in the state space. As the bound increases, BPOR gradually approaches the coverage and time of full DPOR.

Although adding conservative backtrack points decreases the amount of partial-order reduction, we empirically show that BPOR vastly reduces the exploration of redundant executions of the same partial order. We also identify properties of bound functions that do not require conservative backtrack points. Unfortunately, the bounds studied in this paper do not satisfy these properties.

In addition, we recast fair stateless model checking as bounded search [14] and show how to combine BPOR with fair-bounded search. Whereas DPOR cannot explore cyclic state spaces, BPOR with fair-bounded search soundly explores the fair-bounded state space when the state space is cyclic. Most of our test programs have cyclic state spaces that result from the use of lock-free constructs such as spin loops. Fair exploration [14] is essential to scale BPOR to these benchmarks.

Our empirical evaluation shows that BPOR provides bounded coverage far more quickly than preemption-bounded search does. We also show that BPOR finds more bugs faster than DPOR or preemption-bounded search alone.

Section 2 formalizes our representation of concurrent programs, and reviews partial-order reduction and bounded search. Section 3 compares DPOR with bounded search and shows why naïvely combining them sacrifices bounded coverage. The algorithms in Section 4 compensate for bound dependences, computing a sufficient set of transitions to explore in each state. Section 5 introduces bounded partial-order reduction (BPOR) and computes sufficient sets. For brevity here within, we state the theorems and put the proofs in companion materials available from the ACM Digital Library [3]. Coons' dissertation contains a fuller treatment [2].

Section 6 evaluates BPOR performance on concurrent unit tests and Section 7 concludes.

2. Background and related work

This section provides necessary background and describes our formalism for concurrent programs, partial-order reduction, and bounded search.

2.1 Stateless model checking

We use stateless model checking to systematically explore the state space of multithreaded programs [8]. In such an exploration, a runtime scheduler systematically forces concurrent programs down different thread interleavings starting from the same initial state.

There are two useful exploration modes for an implementation of stateless model checking. In *synchronization mode*, the model checker interleaves threads only at synchronization primitives such as locks, events, and volatile variables. In this mode, the checker finds all concurrency errors that do not require a data race to trigger, and it finds all data races.

In *data-race mode*, the checker additionally interleaves threads at every shared memory location. This mode is useful only if the programmer intentionally introduced data races in the program. The model checker determines whether these data races are indeed benign – that is, do not trigger concurrency errors. Usually, data-race mode is more expensive because it requires additional overhead to instrument shared-memory accesses, and an (exponential) increase in state space resulting from fine-grained interleaving. BPOR is applicable to both modes.

2.2 Multithreaded programs and semantics

A concurrent program contains a fixed set *Tid* 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 represents an operation $t.op$ on variable(s) $t.var$ performed by a thread $t.tid \in Tid$. Example operations include fork, join, lock acquire, lock release, and load/store operations.

In stateless model checking, a program state is conservatively identified by the sequence of transitions executed to reach that state from the initial state. Note that it is possible for different sequences of transitions to lead to the same program "state" represented by the contents of the program variables (including registers, stack, and heap). One such possibility is when the two sequences differ only in the order of independent transitions, such as two threads accessing different memory locations. Partial-order reduction seeks to eliminate such redundancies.

Accordingly, we define a *state* as the directed acyclic graph $\langle T, H\rangle$ where $T \subseteq \mathcal{T}$ is a set of transitions and H (for "happens-before") is an irreflexive partial order on T. H captures all the dependences between the transitions such that reordering transitions not related by H behaviorally leads to the same program state. In other words, two transition sequences containing the same set of transitions T

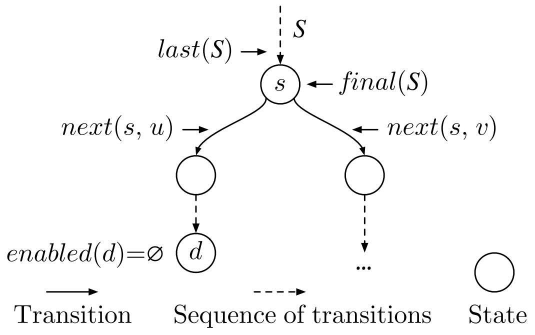


Figure 2. Program representation: S is a sequence of transitions such that state $s = \mathit{final}(S)$. Threads $u, v \in \mathit{enabled}(s)$.

but different linearizations of the partial-order H lead to the same program state $\langle T, H\rangle$. In this paper, we assume that all transitions of a single thread are dependent on each other. Accordingly, we will assume 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.

The *local state* for a thread u in a state $s = \langle T, H\rangle$, denoted by $\mathit{local}(s, u)$, is defined as the subgraph of s that contains only those transitions that happen-before the most recent transition by u. Essentially, these are the transitions that are guaranteed to drive the thread to said local state. Formally, if u has no transitions in T then $\mathit{local}(s, u) = \langle \emptyset, \emptyset\rangle$. Otherwise, if t is the most recent transition by u, then $\mathit{local}(s, u) = \langle T_L, H_L\rangle$, where

$$T_L = \{t' \in T \mid (t', t) \in H\} \cup \{t\}$$
$$H_L = \{(t'', t') \in H \mid t'' \in T_L \text{ and } t' \in T_L\}$$

Figure 2 illustrates the following terms. 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 *deadlock state*.

The expression $s \xrightarrow{t} s'$ indicates that transition t leads from state s to state s'. Using Flanagan and Godefroid's notation [6], 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 $\mathit{dom}(S)$ returns the *domain* of S, the set $\{1 \ldots n\}$, and the length of S is $\mathit{len}(S) = n$. The term $\mathit{final}(S)$ refers to s_{n+1}, the final state reached after executing all transitions in S. Transition S_i is the ith transition in

S, $i \in dom(S)$, and last(S) refers to S_n, the last transition in S. Greek symbols α, β, ω, γ represent any 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 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 [6]. 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 initial state of the system. Bounded search and DPOR each explore only a subset of A_G [5, 6, 12].

A Mazurkiewicz *trace* is an equivalence class of sequences of transitions that can be obtained from one another by permuting adjacent independent transitions [11]. A Mazurkiewicz trace, or trace for concision, is uniquely defined by one of its members. We use $[\omega]$ to denote the trace that contains ω.

Formally, using Godefroid's definition of traces [7], 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 $I_\Lambda = \mathcal{T}^2 \setminus D$ is independence 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 2.1. Traces [7].
Equivalence classes of $\equiv_\Lambda$ are *traces* over Λ. The term $[\omega]$ denotes the trace that contains the sequence of transitions ω.

Intuitively, a trace is a set of sequences of transitions where each sequence in the trace can be derived from each other sequence in the trace by permuting independent transitions.

2.3 Dependence relation

A *dependence relation*, D, identifies transitions whose interleavings may lead to new states. The dependence relation is critical for partial-order methods because it determines which interleavings may be pruned. The following definition characterizes "valid" dependence relations for the transitions of a concurrent system.

Definition 2.2. Valid dependence relation [6, 10].
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 iff 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 $t \in$ *enabled*(s) and $s \xrightarrow{t} s'$, then $t' \in$ *enabled*(s) iff $t' \in$ *enabled*(s')
2. if $t, t' \in$ *enabled*(s), then there is a unique state s' such that $s \xrightarrow{t.t'} s'$ and $s \xrightarrow{t'.t} s'$

We implement the valid dependence relation in Definition 2.3 and use $t \leftrightarrow t'$ to denote that t is independent with t', and $t \not\leftrightarrow t'$ to denote that t is dependent with t'. These operators apply to sequences of transitions, as well.

Definition 2.3. Dependence relation.
Transitions $t, t' \in \mathcal{T}$ are *dependent*, $t \not\leftrightarrow t'$, iff

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

This relation differentiates read operations from write operations because multiple read operations to the same variable commute. We consider only constant dependence relations in this work and assume that the dependence relation is not conditional [9]. Next, we use these definitions to provide background for partial-order reduction and bounded search.

2.4 Partial-order reduction

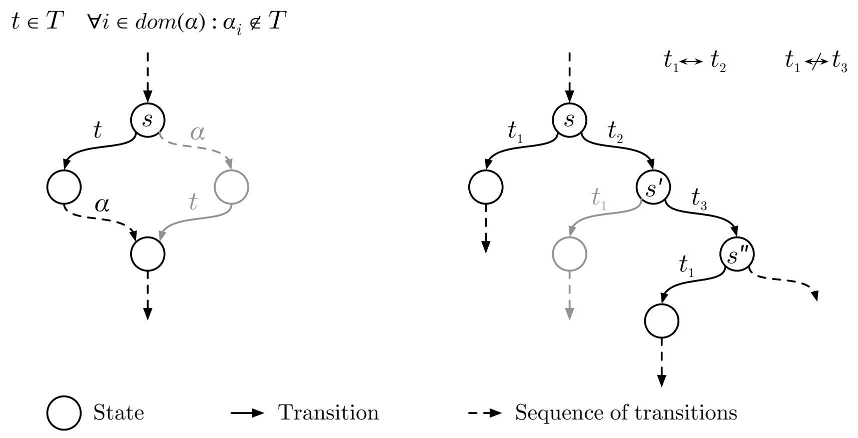
A search using partial-order reduction provides *full coverage*: it explores all relevant, reachable states [7]. A state is relevant if it must be explored to provide the required safety guarantee. We focus on *local state reachability*, which guarantees absence of deadlocks and user-specified local assertion failures.

A state is a partial order on a program's dependent transitions. A sequence of transitions is a total order on those transitions. Many total orders may correspond to a single partial order. Whenever possible, partial-order methods explore only one total order per partial order. The dependence relation in Definition 2.3 identifies transitions whose interleavings do not affect the partial order on transitions. We define two classes of partial-order reduction algorithms, persistent sets and sleep sets, then describe DPOR.

2.4.1 Persistent sets

A *persistent set* in a state s is a sufficient set of transitions to explore from s while maintaining local state reachability for acyclic state spaces [9]. A *selective search* using persistent sets explores a persistent set of transitions from each state s where *enabled*$(s) \neq \emptyset$ and prunes enabled transitions that are not persistent in s. Godefroid defines a persistent set as follows.

Definition 2.4. Persistent sets [9].
A set $T \subseteq \mathcal{T}$ of transitions enabled in a state s is *persistent in* s iff for all nonempty sequences α of transitions from s


$t \in T \quad \forall i \in dom(\alpha) : \alpha_i \notin T$
t
s
α
α
t
$t_1 \leftrightarrow t_2$
$t_1 \not\leftrightarrow t_3$
t_1
s
t_2
t_1
s'
t_3
t_1
s''
State
Transition
Sequence of transitions

(a) T is persistent in s.

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

Figure 3. Persistent sets and sleep sets. Transitions in gray may be pruned.

in A_G such that $\forall i \in dom(\alpha) : \alpha_i \notin T$ and for all $t \in T$, $t \leftrightarrow \text{last}(\alpha)$.

Intuitively, any transitions reachable via transitions not in T are independent with respect to all transitions in T. Figure 3(a) illustrates a transition t in the persistent set T in a state s, and a sequence of transitions α, none of which are in T. The interleaving in gray need not be explored because it is equivalent to the interleaving in black.

2.4.2 Sleep sets

Sleep sets prohibit visited transitions from executing again until the search explores a dependent transition. Assume that the search explores transition t from state s, backtracks t, then explores t' from s instead. Unless the search explores a transition that is dependent with t, no states are reachable via t' that were not already reachable via t from s. Thus, t "sleeps" unless a dependent transition is explored.

Figure 3(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''.

Many algorithms reduce the state space with persistent sets and sleep sets [7, 16, 17]. Most of these algorithms use static analysis to determine which transitions may be dependent with one another. As a result, these algorithms must be conservative. Unless two transitions must always be independent, the search must assume that they may be dependent. In the next section, we review using dynamic information to reduce the search space.

2.4.3 Dynamic partial-order reduction

Dynamic Partial-Order Reduction (DPOR) computes persistent sets on-the-fly [6]. Unlike static conservative dependence detection [4, 8, 18], DPOR detects dependences accurately at runtime. DPOR performs a depth-first search of the state space and keeps track of the most recent access to each variable. When a conflicting access occurs, DPOR inserts a backtrack point to reverse the order of the dependent accesses in a future execution. Flanagan and Godefroid prove that DPOR explores a persistent set of transitions from each state and show that it significantly reduces the search space for a few sample programs [6].

The DPOR algorithm is very effective – it maintains coverage guarantees while significantly reducing search time. DPOR does not provide any incremental guarantees, however. If the search space is large and the test does not terminate sufficiently quickly, then the tester has no guarantees. Additionally, DPOR works only with acyclic state spaces. We wrote unit tests for a set of concurrent data structures using publicly-available source code for the .NET 4.0 framework and found that all of the concurrent data structures internally use spin loops that result in cyclic state spaces and therefore cannot benefit from DPOR.

We would like bounded search to provide incremental guarantees, and to prune cyclic state spaces so that DPOR can explore them. Combining DPOR with bounded search is not straightforward, however. We address this problem to make DPOR more widely applicable. Next, we introduce bounded search, including several bound functions.

2.5 Bounded search

Bounded search explores only executions that do not exceed a bound [5, 12, 14]. The bound may be any property of a sequence of transitions. A *bound evaluation function* $Bv(S)$ computes the *bounded value* for a sequence of transitions S. A bound evaluation function Bv and bound c are inputs to

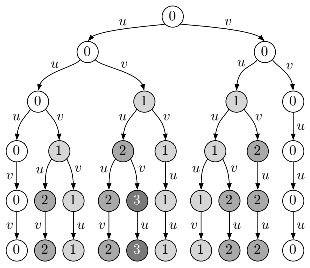


Figure 4. Preemption-bounded search explores executions that contain fewer preemptions first. The u and v labels are threads. Lighter-colored states require fewer preemptions. Numbers indicate bound values.

bounded search. Bounded search may not visit all relevant reachable states; it visits only those that are reachable within the bound. If a search explores all relevant states reachable within the bound, then it provides *bounded coverage*.

Prior work bounds the depth [8], number of context switches [12], number of preemptive context switches [12], and the number of delays that an otherwise deterministic scheduler is allowed [5]. We focus on preemption-bounded search because it is more useful than the depth bound [12], and it has been more widely tested in practice than the other bounds. We also recast fair stateless model checking as bounded search [14].

Testers find preemption-bounded search useful for several reasons. First, it provides an incremental coverage metric when searching the entire state space in a reasonable time period is infeasible. Second, preemption-bounded search finds bugs effectively – many bugs manifest with few preemptions [12]. Testers find fair-bounded search very useful because most realistic state spaces contain cycles [14].

2.5.1 Preemption-bounded search

Preemption-bounded search limits the number of *preemptive* context switches that occur in an execution [12]. The preemption bound is defined recursively as follows.

Definition 2.5. Preemption bound [13].

$$Pb(t) = 0$$

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

Figure 4 illustrates preemption-bounded search, in which the *executing thread* – the thread that performed the previous transition – never requires a preemption. If the executing thread is blocked, then no thread requires a preemption. The preemption bound increases slowly when the same thread executes repeatedly, so the search may explore deep into the state space with a small bound.

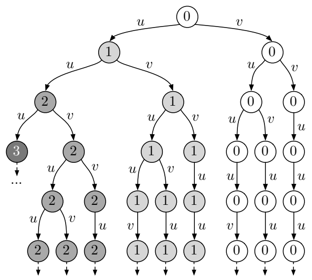


Figure 5. Fair-bounded search explores executions that contain fewer trips through cycles in the state space first. The u and v labels are threads. Lighter-colored states explore fewer trips. Numbers indicate bound values.

2.5.2 Fair-bounded search

Fair-bounded search limits the number of cycles in a cyclic state space. We adapt a fairness criterion from prior work to identify cycles in the state space given two assumptions [14]:

1. Threads always yield the processor when not making progress
2. Threads never yield the processor when making progress

Fair-bounded search tracks the number of yield operations that each thread u has performed after each sequence of transitions S, $Y(S,u)$, and ensures that this value never differs among enabled threads by more than the bound. Formally, we define the fair bound recursively as follows.

Definition 2.6. Fair bound (*Fb*).
Let $Y(S,u)$ return Thread u's yield count in *final*(S).

$$Fb(t) = 0$$
$$Fb(S.t) = \mathit{max}(Fb(S),$$
$$\max_{u \in \mathit{enabled}(\mathit{final}(S))}(Y(S, t.tid) - Y(S,u)))$$

Figure 5 illustrates fair-bounded search for a cyclic state space. Although detecting cycles is impossible in stateless search, prior work argues that counting yield operations is a good approximation [14]. Essentially, a thread executing a bounded number of yield operations is considered to be stuck in a cycle in the state space and thus unable to make progress. The tester sets the bound such that the search neglects spurious yield operations that would otherwise cause the search to miss relevant states. We selected this fairness criterion primarily for its simplicity, but other fairness criteria can be expressed as bounds, as well. Unlike prior work, this bound does not provide strong fairness [1, 14].

Both the preemption and the fairness bound prune portions of the state space and may fail to detect bugs in programs. DPOR, in contrast, explores all relevant states and detects all bugs that violate the correctness guarantee. If the state space is so large that the program does not terminate sufficiently quickly, however, then DPOR does not provide any guarantees. If the state space is cyclic, then

DPOR cannot search it. Ideally, combining these techniques would provide incremental coverage guarantees for a reduced state space. The next section compares these techniques and shows why combining them is not trivial.

3. Partial-order reduction with bounds

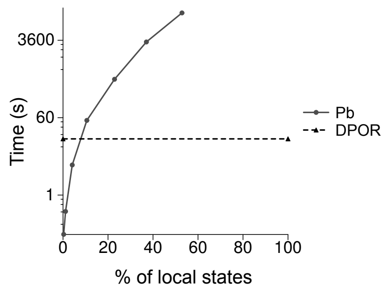
Combining partial-order methods with bounded search should provide incremental coverage guarantees while reducing redundant work. We compare DPOR with bounded search for small unit tests based on each technique's state space coverage over time. We implement DPOR in CHESS, a publicly available model checker for concurrent programs that performs bounded search. We run these tests on a 2.2 GHz Intel Core 2 Duo processor with 4 GB of RAM. We show that bounded search provides little benefit without partial-order reduction, except when the bound is very small.

Figure 6 compares DPOR to bounded search. The x-axis shows the percent of local states that the search visits or, if the search never explores the entire state space, then the total number of local states that the search visits. The y-axis shows the time in seconds that the search requires. Each data point represents an invocation of CHESS with a particular value for the bound, which we iteratively increase. The single dot at 100% of local states for `MRSE` and `FFT` represents an invocation of CHESS with unbounded DPOR. We provide the dotted line for easier comparison. DPOR does not complete within our 3 hour limit on `Exception` and DPOR cannot explore the cyclic state space in `Fair`.

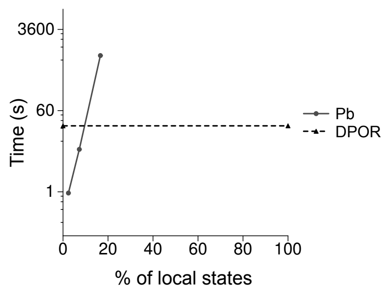
By default, CHESS preempts only prior to synchronization variable accesses, and uses a data-race detector to identify accesses to shared variables that it may have missed as a result [12]. When a test does contain a data-race, CHESS therefore does not explore portions of the state space reachable by re-ordering the data-racy accesses. As a result, in our tests we force CHESS to preempt at all shared variable accesses because otherwise the state spaces that CHESS searches and that BPOR searches are not comparable. All of our test programs contain data races, so CHESS provides insufficient coverage on these programs if it preempts only at synchronization variable accesses. An important advantage of BPOR is that it allows preemption-bounded search to scale without sacrificing coverage for data-racy programs.

These results show the limitations of bounded search and DPOR. For `MRSE` and `FFT`, 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. Bounded search could thus benefit greatly from partial-order reduction. We enforced a three-hour time limit per test. With this limit, the `Exception` test does not terminate within 3 hours with DPOR, so DPOR provides no guarantees. `Fair` contains a cyclic state space and thus DPOR never terminates in Figure 6(d).

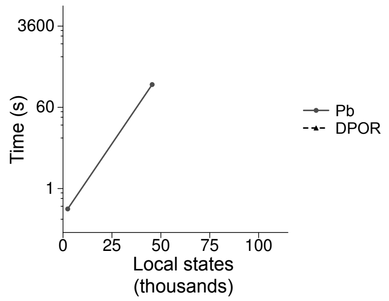
Bounded search makes large state spaces tractable, but it wastes too much time exploring redundant states. DPOR ex-



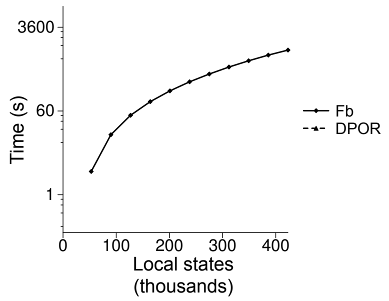
(a) `MRSE`



(b) `FFT`



(c) `Exception`



(d) `Fair`

Figure 6. Coverage vs. time for DPOR and preemption-bounded (Pb) and fair-bounded (Fb) search. DPOR explores the entire state space faster than preemption-bounded search explores only a subset for `MRSE` and `FFT`. `Fair` never terminates with DPOR because its state space is cyclic. `Exception` exceeds our three-hour time limit with DPOR.

plores the *entire* state space in less time than bounded search requires to explore a small subset of that state space, provided that the state space is acyclic and relatively small. The time DPOR requires scales exponentially with the number of successive dependent accesses. As this number grows, the state space quickly becomes intractable. Without a bound, if the state space is intractably large, then DPOR runs for hours, days, or longer without providing any guarantees. Likewise, if the state space is cyclic, DPOR provides no guarantees.

Practical bounded search requires DPOR's aggressive state space reduction. Unfortunately, as Figure 1 illustrates, sequences of transitions that lead to the same local state may have different bounded values, so the search cannot guarantee that it has taken the cheapest path to each state. DPOR may prune transitions that make new states reachable within the bound, sacrificing bounded coverage.

This unsoundness arises 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 dependent with it, by Definition 2.2 of valid dependence relations.

The dependences that the bound introduces are different from dependences in 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 meet the safety criteria. If we treat bound dependences equivalently to dependences in the program under test, then the search must explore each state with *all* possible bounded values, which is an enormous waste of time.

Instead, we must differentiate bound dependences from dependences in the program under test. Bounded search need not explore both orders of bound dependent transitions – it must explore only the cheapest order. 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. In the next section, we show how to compensate for these dependences to combine DPOR with bounded search while maintaining bounded coverage.

4. Bound persistent sets

To compensate for the dependences that bounded search creates, we introduce *bound persistent sets*, which reduce the size of the state space while guaranteeing bounded coverage. First, we establish sufficient conditions to guarantee absence of local assertion failures among executions that do not exceed the bound. For concision, we put all the proofs in companion materials in the ACM Digital Library [3]. Coons' dissertation contains a fuller treatment [2].

Algorithm 1 Bounded selective search

```
1: Initially, **Explore**($\emptyset$)
2: **procedure Explore**($S$) **begin**
3:     $T =$ **Sufficient_set**($\mathit{final}(S)$)
4:     **for all** ($t \in T$) **do**
5:         **if** ($Bv(S.t) \leq c$) **then**
6:             **Explore**($S.t$)
```

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 in the sufficient set. A 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 enable more state space reduction.

Selective search explores only a sufficient set of transitions from each state [7]. Algorithm 1 performs bounded selective search. Line 3 returns a nonempty sufficient set in each state $\mathit{final}(S)$, and Lines 4-6 recursively explore only the transitions in that sufficient set that do not exceed the bound. Requirements for this sufficient set vary with the bound evaluation function and with the desired safety guarantee. We identify constraints on a sufficient set such that Algorithm 1 guarantees absence of local assertion failures among sequences of transitions that do not exceed the bound. We use $A_{G(Bv,c)}$ to refer to a generic bounded state space with bound function *Bv* and bound c.

Definition 4.1. ***Prefix***$([\omega])$ [7].
Prefix$([\omega])$ returns the set containing all prefixes of all sequences in the Mazurkiewicz trace defined by ω.

Definition 4.2. 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' from s in $A_{G(Bv,c)}$ such that $\omega \in$ *Prefix*$([\omega'])$ and $\omega'_1 \in T$.

Let $A_{R(Bv,c)}$ be the reduced state space that Algorithm 1 explores if Line 3 returns a nonempty local sufficient set in each state.

Theorem 1. *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 $A_{R(\mathrm{Bv},c)}$.*

Intuitively, there must exist a sequence ω' such that $\omega \in$ *Prefix*$([\omega'])$ and $\omega'_1 \in T$. Thus, there exists a sequence β such that $\omega.\beta \in [\omega']$. Any two sequences of transitions that lead to the same global state reach all of the same local states, so ω' must lead to l as well.

Thus, if Algorithm 1 returns a local sufficient set of transitions in each state at Line 3, then Algorithm 1 explores all local states reachable within the bound. In unbounded search, persistent sets are local sufficient, provided that the state space is acyclic [7]. Persistent sets may not be local sufficient in bounded search, however. In the next section, we identify properties of bound functions that determine how conservative their local sufficient sets must be.

4.2 Properties of bound functions

Two properties of bound functions enable bounded partial-order reduction. We first define each property for a generic bound function *Bv*.

Definition 4.3. 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, in a stable bound function, any two sequences of transitions that lead to the same global state cost the same amount. This property is desirable because independent transitions remain commutative with stable bound functions. 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 global 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 it must also explore the cheaper path. This redundant execution sacrifices partial-order reduction. The search prunes the state space most efficiently if it explores the cheapest sequence of transitions first.

Definition 4.4. Extensible bound functions.
Bound function *Bv* is *extensible* if and only if for all sequences of transitions S in $A_{G(Bv,c)}$, 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 require that independent transitions not affect one another's cost. 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. Bounds from prior work [5, 12, 14] bound the total order on a program's transitions and are thus neither stable nor extensible – they introduce artificial dependences that partial-order reduction must accommodate.

4.3 Bound sufficient sets

We introduce *bound sufficient sets* to compensate for dependences imposed by the bound and thus guarantee bounded coverage. We show that unfortunately the preemption and fair bounds are neither stable nor extensible. However, we can and do define sufficient sets for preemption-bounded and fair-bounded search and prove that these sets are sufficient to explore the bounded state space soundly, but they reduce partial order reduction in some cases.

4.3.1 Preemption-bounded (Pb) search

Preemption-bounded search limits the number of *preemptive* context switches in each execution [14]. The preemption bound is neither stable nor extensible. Each transition t's cost depends upon whether or not the prior transition t' is enabled, even if $t \leftrightarrow t'$. To compensate for this dependence, a *preemption-bound persistent set* T requires that each transition $t \in T$ be independent with the *next* transition by each thread that is not in T, even if that transition is disabled.

Preemption-bounded search with DPOR reduces the state space most effectively if it visits new states via the cheapest path first. When the executing thread is enabled in a state s, its next transition is cheaper than all other enabled transitions in s. Exploring the executing thread first is thus a good heuristic for reaching new states as cheaply as possible.

If the executing thread executes until it blocks, then *any* transition can execute for free. We exploit this property to perform limited partial-order reduction even when transitions increment the bound. 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. We introduce *preemption-bound persistent sets* to permit limited partial-order reduction with local state reachability for preemption-bounded search.

Definition 4.5. ***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. $\forall i \in \mathit{dom}(\beta) : \beta_i.tid = t.tid$
2. $t.tid \notin \mathit{enabled}(\mathit{final}(S.\beta))$

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

Definition 4.6. 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 iff 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
   $t \leftrightarrow \mathit{next}(\mathit{final}(S.\alpha), \text{last}(\alpha).tid)$

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

Assume that in each state of the reduced state space $A_{R(Pb,c)}$, Algorithm 1 returns a preemption-bound persistent set. We provide two lemmas to manage the bound, and a theorem stating that a nonempty preemption-bound persistent set is local sufficient.

Lemma 2. *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. $\forall i \in \text{dom}(\beta) : \beta_i.tid = \beta_1.tid$
4. $\beta \leftrightarrow \text{next}(\text{final}(S.\alpha_1 \dots \alpha_i), \alpha_i.tid), 1 \leq i < \text{len}(\alpha)$
5. *if* $\text{Pb}(S.\beta_1) = \text{Pb}(S.\alpha_1)$*, then* $\beta_1.tid \notin \text{enabled}(\text{final}(S.\beta))$

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

Intuitively, β does not contain any context switches and $\beta \leftrightarrow \alpha$, so placing β prior to α does not modify the program's behavior or increase the cost of the transitions in α. If β contains a release operation, then it still cannot increase the cost of any transition in α because β is independent with the *next* transition by each thread in α. Thus, β can never enable or disable the executing thread in α, and cannot affect whether any transitions in α require a preemption.

Lemma 3. *Let T be a nonempty preemption-bound persistent set in a state $s = \text{final}(S)$ in $A_{R(\text{Pb},c)}$ and let $\alpha.\beta.\gamma$ be a sequence of transitions from s in $A_{G(\text{Pb},c)}$ such that α and β are nonempty and*

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

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

The intuitions for Lemma 3 are similar to the intuitions for Lemma 2. To account for preemptions γ_1 may incur, we require that $\gamma_1.tid \neq \beta_1.tid$.

Theorem 4. *If T is a nonempty preemption-bound persistent set in a state s in $A_{R(\text{Pb},c)}$, then T is local sufficient in s.*

The proof of Theorem 4 follows the correctness proof for persistent sets, but leverages Lemmas 2 and 3 to show that relevant sequences of transitions do not exceed the bound [2, 3]. By Theorems 4 and 1, if Algorithm 1 explores a nonempty preemption-bound persistent set in each state, then it reaches all local states reachable in $A_{G(Pb,c)}$. Next, we define local sufficient sets for fair-bounded search.

4.3.2 Fair-bounded (Fb) search

Fair-bounded search limits the maximum difference between the executing thread's yield count in each state s and the yield count of other enabled threads in s. This bound prunes executions in which a thread yields the processor repeatedly, and thus prunes cycles from cyclic state spaces under the assumption that threads that are not making progress always yield the processor [14].

The fair bound is neither stable nor extensible due to release operations. A release operation may enable threads with a lower yield count, and thus increase the cost of an independent, enabled transition. To provide local state reachability for fair-bounded search, we introduce *fair-bound persistent sets*. Fair-bound persistent sets compensate for dependences that the fair bound introduces by conservatively scheduling all threads prior to release operations.

Definition 4.7. Fair-bound persistent sets.
A set $T \subseteq \mathcal{T}$ of transitions enabled in a state $s = final(S)$ is *fair-bound persistent* in s if and only if for all nonempty 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. $t \leftrightarrow \text{last}(\alpha)$

Requirement 1 requires that transitions in the fair-bound persistent set not exceed the bound. Requirement 2 requires that all enabled threads be scheduled prior to release operations. This requirement is conservative – a less restrictive requirement might permit more partial-order reduction while still providing coverage guarantees. We choose this requirement to compensate for the dependences that release operations introduce in fair-bounded search because it is simple and intuitive yet still provides partial-order reduction. Requirement 3 requires that transitions reachable via transitions not in the fair-bound persistent set be independent with transitions in the fair-bound persistent set.

Let $A_{R(Fb,c)}$ be the reduced state space explored by Algorithm 1 with bound function *Fb* and bound c. Assume that in each state, Algorithm 1 returns a fair-bound persistent set. We provide two lemmas to manage the bound, and a lemma stating that a nonempty fair-bound persistent set in a state s is local sufficient in s.

Lemma 5. *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 $A_{G(\text{Fb},c)}$.

Algorithm 2 BPOR with bound function *Bv* and bound c

```
1:  Initially, **Explore**($\epsilon$) from $s_0$
2:  procedure Explore($S$) begin
3:    Let $s = \mathit{final}(S)$
      # Add backtrack points
4:    for all ($u \in \mathit{Tid}$) do
5:      for all ($v \in \mathit{Tid} \mid v \neq u$) do
          # Find most recent dependent transition
6:        if ($\exists i = \mathbf{max}(\{i \in \mathit{dom}(S) \mid S_i \not\leftrightarrow \mathit{next}(s,u)$ and $S_i.\mathit{tid} = v\})$) then
7:          Backtrack($S, i, u$)
      # Continue the search by exploring successor states
8:    Initialize($S$)
9:    Let $\mathit{visited} = \emptyset$
10:   while ($\exists u \in (\mathit{enabled}(s) \cap \mathit{backtrack}(s) \setminus \mathit{visited})$) do
11:     add $u$ to $\mathit{visited}$
12:     if ($\mathit{Bv}(S.\mathit{next}(s,u)) \leq c$) then
13:       Explore($S.\mathit{next}(s,u)$)
```

Intuitively, t can increase the cost of a transition in α only if t is a release operation that enables a transition t' with a lower yield count than one of the transitions in α, yet t is not a release operation.

Lemma 6. *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 $A_{G(\text{Fb},c)}$.*

The intuitions for Lemma 6 are similar to the intuitions for Lemma 5. If a transition in γ exceeds the fair bound in $S.t.\alpha.\gamma$ then it also exceeds the fair bound in $S.\alpha.t.\gamma$.

Theorem 7. *If T is a nonempty fair-bound persistent set in a state s in $A_{R(\text{Fb},c)}$, then T is local sufficient in s.*

The proof of Theorem 7 follows the correctness proof for persistent sets, but it leverages Lemmas 5 and 6 to show that each relevant sequence of transitions does not exceed the bound [2, 3]. By Theorems 7 and 1, if Algorithm 1 explores a nonempty fair-bound persistent set in each state then it reaches all local states reachable in $A_{G(Fb,c)}$.

We have identified constraints on sufficient sets that permit limited partial-order reduction while providing bounded coverage for preemption-bounded and fair-bounded search. In the next section, we present an algorithm to dynamically compute bound persistent sets for these bound functions at runtime. We generalize DPOR to search a bounded state space, then specialize this algorithm for each bound function and prove the resulting algorithm correct.

5. Computing bound persistent sets

Algorithm 2 presents bounded, dynamic partial-order reduction (BPOR), a modified version of DPOR that computes a bound persistent set in each state. We specialize BPOR to compute preemption-bound persistent and fair-bound persistent sets and prove the resulting algorithms correct. First, we summarize Algorithm 2 and highlight differences from DPOR. The procedure **Explore** in Algorithm 2, which is common to all bound evaluation functions, recursively explores the bounded state space from a state $s = \mathit{final}(S)$. Lines 4-7 create backtrack points.

For each thread u, Line 6 computes the most recent transition in S by each thread v that is dependent with $\mathit{next}(\mathit{final}(S), u)$. For each such dependence, Line 7 creates a backtrack point to reverse the order of the dependent transitions in a future execution. The original DPOR algorithm places a backtrack point only prior to the most recent dependent transition, rather than the most recent dependent transition by each thread. We make this change because we differentiate read and write operations, so we must consider read operations by each thread. This change also simplifies the proofs and allows us to simplify the algorithm.

Lines 8-13 recursively explore the state space from s. Line 12 ensures that the next transition does not exceed bound c. Line 13 recursively explores Thread u's next transition. This recursive search may add additional threads to the backtrack set in s.

The **Backtrack** and **Initialize** procedures are specific to each bound evaluation function. The **Backtrack** procedure adds backtrack points to compensate for dependences that the bound introduces. **Initialize** initializes the bound persistent set 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 transitions first. We specialize versions of **Backtrack** and **Initialize** for the preemption and fair bounds.

Algorithm 2 is simplified in comparison to the original DPOR algorithm to make it easier to describe and to make its proofs more intuitive. We provide results for this simplified algorithm and also for the original DPOR algorithm, which we denote as "optimized." The optimizations we omit in the unoptimized version include the following [2].

1. Backtrack only dependences that are in the transitive reduction on the program's partial order.
2. If the backtracked thread is disabled, backtrack any enabled thread that is transitively dependent with it.
3. Do not backtrack release operations – backtrack only prior to the matching acquire instead (DPOR only).

The last optimization does not apply to preemption-bounded or fair-bounded BPOR because both of these algorithms must specifically backtrack release operations. We include these optimizations for DPOR in Figure 6 when we compare

Algorithm 3 BPOR for preemption-bounded search

```
1:  procedure Initialize($S$) begin
2:      if (last($S$)$.tid \in$ enabled(final($S$))) then
3:          add last($S$)$.tid$ to backtrack(final($S$))
4:      else
5:          add any $u \in$ enabled(final($S$)) to backtrack(final($S$))
6:  procedure Backtrack($S, i, u$) begin
7:      AddBacktrackPoint($S, i, u$)
8:      if ($j = \mathbf{max}(\{j \in dom(S) \mid j = 0$ or $S_{j-1}.tid \neq S_j.tid$ and $j < i\})$) then
9:          AddBacktrackPoint($S, j, u$)
10: procedure AddBacktrackPoint($S, i, u$) begin
11:     if ($u \in$ enabled(pre($S, i$))) then
12:         Add $u$ to backtrack(pre($S, i$))
13:     else
14:         backtrack(pre($S, i$)) = enabled(pre($S, i$))
```

Algorithm 4 BPOR procedures for fair-bounded search

```
1:  procedure Initialize($S$) begin
2:      if (len($S$) $>$ MAX) then
3:          report livelock and exit
4:      Backtrack($S$, len($S$), $u$) where $u$ is a lowest cost enabled thread in final($S$)
5:  procedure Backtrack($S, i, u$) begin
6:      if ($u \in$ enabled(pre($S, i$)) and next(pre($S, i$), $u$) is not a release operation) then
7:          add $u$ to backtrack(pre($S, i$))
8:      else
9:          backtrack(pre($S, i$)) = enabled(pre($S, i$))
```

DPOR to bounded search. In Section 6, however, we separate these optimizations out because we do not include these optimizations in our formal proofs [2, 3].

5.1 Computing preemption-bound persistent sets

Algorithm 3 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 because all threads cost the same in *final*(S).

The **Backtrack** procedure adds two backtrack points: at Line 7 it adds one prior to the most recent dependent transition S_i, and at Line 9 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 4.6 and the second backtrack point satisfies Requirement 3 of Definition 4.6. The backtrack point at Line 9 is *conservative* – it may lead to precisely the same states that the backtrack point at Line 7 leads to, but with fewer preemptions. The search cannot know whether these backtrack points lead to the same states yet, however, because it has not yet searched the intervening state space. Thus, the search must add both backtrack points. 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 prove that Algorithm 2 computes a preemption-bound persistent set in each state, we create postconditions for preemption-bounded search, which we derive from the postconditions that DPOR satisfies in each state [6].

Definition 5.1. ***PC* for Explore**(S) **– Preemption bound.**
$\forall u \forall \omega :$ **if** $Pb(S.\omega) \leq c$ **then** *Post*$(S.\omega, len(S), u)$

Definition 5.2. ***Post***(S, k, u) **– Preemption bound.**
$\forall v :$ **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 5.1 requires that *Post* hold for all threads and for all sequences of transitions that are reachable within the preemption bound. Definition 5.2 requires an additional backtrack point to guarantee that *ext*(*pre*(S, j), *next*(*pre*$(S, j), u$)) is independent with transitions not in the local sufficient set. Requirement 3 of Definition 4.6 of preemption-bound persistent sets requires this backtrack point.

Theorem 8. *Whenever a state $s =$* final(S) *is backtracked during the search performed by Algorithm 2 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.*

This proof leverages Lemmas 2 and 3 to show that each sequence of transitions is reachable within the preemption bound [2, 3].

5.2 Computing fair-bound persistent sets

Algorithm 4 contains the **Initialize** and **Backtrack** procedures to compute fair-bound persistent sets. Assume that Algorithm 2 calls these procedures. **Initialize** adds any minimum-cost enabled thread to the backtrack set. **Backtrack** checks whether S_i is a release operation, or u is disabled in *pre*(S, i). In either case, the search conservatively adds all enabled threads to the backtrack set. Otherwise, Line 7 adds u to the backtrack set.

To prove that Algorithm 2 computes a fair-bound persistent set in each state, we define postconditions that Algorithm 2 guarantees prior to backtracking each state [6].

Definition 5.3. ***PC* for Explore(S) - Fair bound.**
$\forall u \forall \omega :$ **if** $Fb(S.\omega) \leq c$ **and** $len(S.\omega) \leq MAX$ **then** $Post(S.\omega, len(S), u)$

Definition 5.4. ***Post*(S, k, u) - Fair bound.**

```
∀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 then
        $u \in backtrack(pre(S, i))$
    else $backtrack(pre(S, i)) = enabled(pre(S, i))$
```

Definition 5.3 requires that *Post* hold for all threads and for all sequences of transitions that are reachable within the fair bound and within a maximum depth parameter, *MAX*. This maximum depth bound should be very large such that it constrains the search only when the state space contains a cycle that the fair bound cannot break. In such a case, the search reports a livelock to indicate that a thread likely yielded the processor when it should not have, or failed to yield the processor when it should have. If u is enabled in $pre(S, i)$ and S_i is not a release operation, then Definition 5.4 requires that u be in $backtrack(pre(S, i))$. Otherwise, Definition 5.4 conservatively requires that all enabled threads be in the backtrack set in $pre(S, i)$.

Theorem 9. *Whenever Algorithm 2 backtracks a state $s = \text{final}(S)$, postcondition* Post *for* ***Explore***(S) *is satisfied, and the set T of transitions explored from s is fair-bound persistent in s.*

This proof is similar to DPOR's correctness proof, but it leverages the **Initialize** procedure in Algorithm 4 and Lemma 6 to show that each sequence of transitions is within the fair bound [2, 3].

5.3 Combining bounds

Combining bound functions may be advantageous if the bounds serve fundamentally different purposes, as the preemption and fairness bounds do. Combining these bounds provides the incremental coverage guarantees of preemption-bounded search but also prunes cyclic state spaces. Combining these bounds sacrifices partial-order reduction, however. If a fair-blocked transition does not allow a non-preemptive context switch, then the search may lose coverage.

When using multiple bounds, it is also more likely that the search will reach states in which all transitions exceed a bound. For example, a preemption-bounded, fair-bounded search may 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.

5.4 Optimizations

In the performance results, we report in the next section, we always apply sleep sets to DPOR and BPOR. The sleep sets algorithm is complementary to the partial-order reduction algorithm and ensures that previously visited transitions are not visited again until after the search explores a dependent transition. To compensate for the bound, we never place transitions in the sleep set if they were conservatively added by BPOR due to the bound.

The optimized results also include one additional optimization that is specific to the bound. When all states are reachable within the bound from a given state, then the search does not add any conservative backtrack points. This optimization ensures that with a sufficiently large bound, BPOR will behave exactly like DPOR and explore the entire state space with no conservative overhead due to the bound.

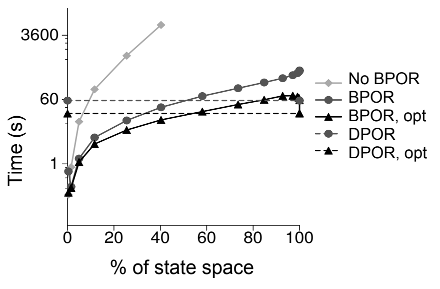
6. Results

We evaluate BPOR by measuring its state space reduction and time required to manifest known bugs. We measure state space coverage over time to ensure that the per-transition overhead of BPOR's bookkeeping is not lost. We compare to bounded search without partial-order reduction, and if the DPOR search terminates, then we compare to DPOR as well.

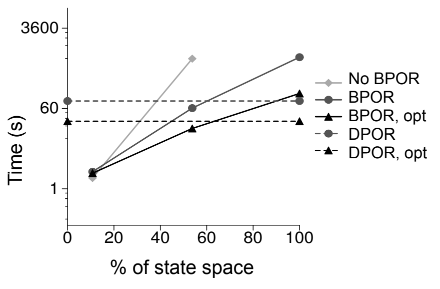
Methodology We compare DPOR, bounded search, and BPOR in CHESS, a publicly available, stateless, dynamic model checker for concurrent software. CHESS places a thin wrapper between the program under test and the Win32 and .NET APIs using binary instrumentation [15]. This wrapper intercepts calls into 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.

We validate our implementation in several ways. First, we hash local and deadlock states to track the unique states that the search visits. We compare these states with and without partial-order reduction to ensure that they are the same. Second, we automatically generate random concurrent programs and compare the states that DPOR, bounded search, and BPOR explore for these programs. Third, we explicitly verify that the lemmas and postconditions in Section 5 are true at runtime.

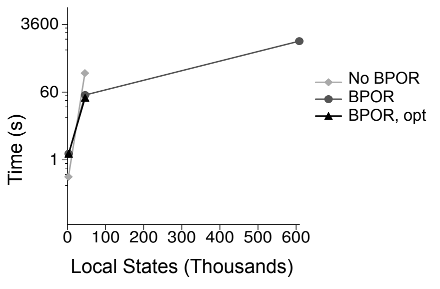
We test four concurrent unit tests developed by testers for concurrent software and libraries at Microsoft. `Exception` tests a concurrent exception for the Concurrency Coordination Runtime (CCR) library. `MRSE` tests a manual reset event for the .NET 4.0 concurrency libraries. `FFT` is a parallel fast Fourier transform with eight threads. We also provide results for a microbenchmark that we created explicitly to test fair-bounded search without partial-order reduction. Because fair-bounded search prunes only cycles in the



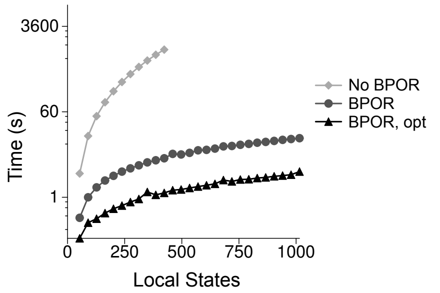
(a) `MRSE` - Preemption bound



(b) `FFT` - Preemption bound



(c) `Exception` - Preemption bound



(d) `Fair` - Fair bound

Figure 7. Coverage vs. time as the bound increases.

state space, its state space is intractably large without partial-order reduction or other bound functions, so we needed a test that was small enough that we could explore the entire fair-bounded state space.

Coverage time Figure 7 shows coverage over time for each benchmark. If at least one test explores the entire state space, then we show % of local states, otherwise we show the raw number of visited local states on the x-axis. We measure and report time on the y-axis as a function of explored states and the bound value to show the overhead of executing BPOR compared to the other approaches. Each point on the graphs in Figure 7 represents an invocation of CHESS with a particular bound function and bound. Lines connect points only for visual clarity – we test only integer bounds. Dashed lines represent DPOR, which is a single invocation of CHESS that searches the entire state space. We show both optimized and unoptimized results because our proofs explicitly cover only the unoptimized algorithms. In practice, the optimized results explore all of the same local states, but we have not proved the optimizations.

BPOR explores new states significantly more quickly than bounded search without partial-order reduction, particularly as the bound increases. Note that the y-axis is logarithmic in Figure 7. We show results for `MRSE` to compare BPOR with DPOR when the search space is small enough that DPOR terminates. We include `FFT` because it is pathological for the preemption bound. `FFT` contains eight threads that access a shared lock. The lock release operations all require conservative backtrack points with preemption-bounded BPOR. Each acquire operation also requires a conservative backtrack point that turns out to be unnecessary. Still, BPOR reduces search time. Note that we terminate tests if they do not complete after 3 hours – without BPOR, most tests time out.

DPOR does does not terminate within 3 hours for the `Exception` test, and neither does preemption-bounded search with a bound of two or greater. BPOR search does terminate within three hours with a bound of two. Figure 7(d) shows fair-bounded search with and without partial-order reduction. There are no DPOR results for this program because it has a cyclic state space, and thus DPOR crashes with a stack overflow. BPOR improves the rate at which fair-bounded search explores new states considerably over fair-bounded search without partial-order reduction.

Finding Bugs Table 1 shows time required to find several previously known bugs with DPOR, without any partial-order reduction, and with BPOR. Tests marked with "-" either do not find the bug within three hours, or die after the first execution due to a stack overflow as a result of the cyclic state space. All of the DPOR and BPOR tests in Table 1 use the optimized search settings described in Section 5, and each bounded search uses the minimum bound required to detect the bug. This result is a best-case scenario for bounded search because the ideal bound cannot be known beforehand.

DPOR cannot explore the `NQueens`, `Matrix`, or `RegOwn` tests because they have cyclic state spaces. Fair-bounded BPOR finds each bug, and preemption/fair-bounded BPOR finds each bug more quickly than fair-bounded BPOR de-

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 1. 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.

spite the conservative approach we chose to combine these bounds.

7. Conclusions

This paper exploits properties of bounds to combine preemption and fair-bounded search with dynamic partial-order reduction (DPOR). We show that bounded search alone is insufficient to reduce the time to find many bugs without partial-order reduction. DPOR is also insufficient without a bound if the search space is large or the test program is cyclic. Bounded, dynamic partial-order reduction (BPOR) provides incremental coverage guarantees for a reduced state space. We specialize this algorithm for preemption and fair-bounded search, prove its coverage guarantees, and show that it reduces search time considerably in practice. We describe two desirable bound properties: stability and extensibility. We show that neither preemption or fair-bounds have these properties. However, these properties do point to ways to choose better bound functions [2]. Bounded partial-order reduction gives testers a more effective tool for verifying the correctness of concurrent programs with bounded guarantees.

Acknowledgments

This research was supported in part by NSF SHF-0910818 and CCF-1018271. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors. We thank Bert Maher for endless discussions, proof reading, and graphs.

References

[1] Apt, K. R., Francez, N., and Katz, S. Appraising fairness in languages for distributed programming. In *Distributed Computing* (1988), vol. 2, pp. 226–241.

[2] Coons, K. E. *Fast Error Detection with Coverage Guarantees for Concurrent Software*. PhD thesis, The University of Texas at Austin, 2013.

[3] Coons, K. E., Musuvathi, M., and McKinley, K. S. Bounded partial order reduction (Proof source material in the ACM Digital Library). In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (2013).

[4] Dwyer, M. B., Hatcliff, J., and Ranganath, R. V. P. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. In *Formal Methods in System Design* (2004), vol. 25, pp. 199–240.

[5] Emmi, M., Qadeer, S., and Rakamaric, Z. Delay-bounded scheduling. In *ACM SIGACT-SIGPLAN Principles of Programming Languages (POPL)* (2011), pp. 411–422.

[6] Flanagan, C., and Godefroid, P. Dynamic partial-order reduction for model checking software. In *ACM SIGPLAN-SIGACT Principles of Programming Languages (POPL)* (2005), pp. 110–121.

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

[8] Godefroid, P. Model checking for programming languages using Verisoft. In *ACM SIGACT-SIGPLAN Principles of Programming Languages (POPL)* (1997), pp. 174–186.

[9] Godefroid, P., and Pirottin, D. Refining dependencies improves partial-order verification methods (extended abstract). In *International Conference on Computer Aided Verification (CAV)* (1993), pp. 438–449.

[10] Katz, S., and Peled, D. Defining conditional independence using collapses. In *Theoretical Computer Science* (1992), vol. 101, Elsevier Science Publishers, pp. 337–359.

[11] Mazurkiewicz, A. Trace theory. In *Advances in Petri nets 1986, part II on Petri nets: Applications and relationships to other models of concurrency* (1986), Springer-Verlag, pp. 279–324.

[12] Musuvathi, M., and Qadeer, S. Iterative context bounding for systematic testing of multithreaded programs. In *ACM Conference on Programming Language Design and Implementation* (2007), pp. 446–455.

[13] Musuvathi, M., and Qadeer, S. Partial-order reduction for context-bounded state exploration. Tech. Rep. MSR-TR-2007-12, Microsoft Research, 2007.

[14] Musuvathi, M., and Qadeer, S. Fair stateless model checking. In *ACM Conference on Programming Language Design and Implementation* (2008), pp. 362–371.

[15] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, A., and Neamtiu, I. Finding and reproducing heisenbugs in concurrent programs. In *USENIX Symposium on Operating Systems Design and Implementation* (2009).

[16] Overman, W. T. *Verification of concurrent systems: Function and timing*. PhD thesis, University of California, Los Angeles, 1981.

[17] VALMARI, A. A stubborn attack on state explosion. In *International Workshop on Computer Aided Verification (CAV '90)* (1990), Springer-Verlag, pp. 156–165.

[18] VISSER, W., HAVELUND, K., BRAT, G., AND PARK, S. Model checking programs. In *Automated Softeware Engineering Journal* (2000), pp. 3–12.