Partial-Order Reduction for Context-Bounded State Exploration

Madanlal Musuvathi Shaz Qadeer

February 8, 2007

Technical Report
MSR-TR-2007-12

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

Partial-Order Reduction for Context-Bounded State Exploration

Madanlal Musuvathi and Shaz Qadeer

Microsoft Research

Abstract. Iterative context-bounding is a technique for performing prioritized search of the state-space of multithreaded programs. A context switch occurs in a concurrent execution when a thread temporarily stops and a different thread resumes. Iterative context-bounding gives priority to executions with fewer context switches during state-space search, exploring for a given context-bound c only those executions in which the number of context switches is at most c. Prior work has shown that this search algorithm is effective in finding many subtle concurrency errors in large programs.
Partial-order reduction has traditionally been applied in complete state-space search and depth-bounded search for reducing the cost of state exploration; however, these techniques have not been applied to context-bounded search. As we show in our paper, it is difficult to perform partial-order reduction during a context-bounded search because of subtle interactions between the two techniques. The main contribution of our paper is an algorithm for performing partial-order reduction for context-bounded state exploration.

1 Introduction

Multithreaded programs are difficult to get right. Unexpected thread interleavings lead to crashes that occur late in the software development cycle or even after the software is released. Empirical evidence clearly demonstrates that traditional methods for testing multithreaded programs are inadequate. Model checking [3, 19] or systematic exploration of program behavior is a promising alternative for verifying such programs. However, it is challenging to apply systematic exploration to large programs because the number of possible executions of a program increases exponentially with the length of the execution.

In recent work [14], we introduced the technique of iterative context-bounding for effectively searching the state space of a multithreaded program. A *context switch* occurs in an execution when a thread temporarily stops even though it is enabled and a different thread resumes. Iterative context-bounding gives priority to executions with fewer context switches during state-space search. For a given context-bound c, the search explores only those executions in which the number of context switches is at most c.

There are three important reasons for the efficacy of iterative context-bounding in state-space search. First, for a fixed number of context switches, the total number of executions in a (terminating) program is *polynomial* in the number of steps

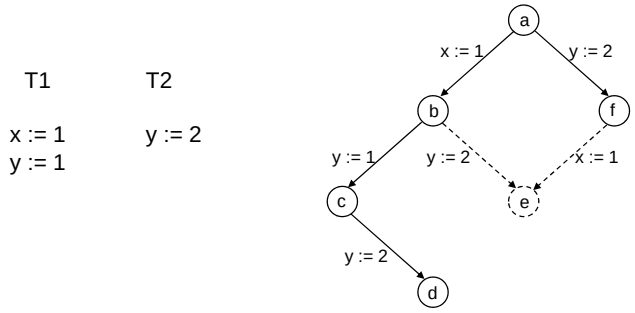

Fig. 1. Example

taken by each thread [14]. This theoretical upper bound makes it practically feasible to scale systematic exploration to large programs without sacrificing the ability to go deep in the state space. Second, we have observed empirically that many subtle concurrency errors are manifested in executions with few context switches. In fact, our implementations of context-bounded search in the KISS [18] and CHESS [14] model checkers have revealed many subtle bugs in real-world multithreaded programs. Each of these errors was exposed by an execution with a *small* number of context switches. Finally, if context-bounded search terminates with c context switches without reporting any errors, it is guaranteed that any error in the program requires at least $c+1$ context switches. Not only does context-bounded search provide a valuable coverage metric but also it is sound in the limit as the search parameter is increased.

Partial-order reduction [21, 8, 16] is a class of techniques orthogonal to context-bounding for reducing the complexity of state-space search. These techniques have proved to be very effective in improving the scalability of model checking on message-passing software [7]. In this paper, we address the problem of incorporating partial-order reduction in context-bounded search.

The context-bound of a *totally-ordered* execution is the number of context-switches in it. The context-bound of a *partially-ordered* execution is the minimum among the context-bound of all totally-ordered executions equivalent to it. Given a context-bound c, we would like to explore each partially-ordered execution with a context-bound of c or less exactly once. In Section 3, we present an algorithm to solve precisely this problem.

A partial-order reduction algorithm is characterized by the rules it provides for pruning a subset of the enabled transitions in each state. During depth-first search, context-bounding can be trivially added by pruning a transition either if the partial-order reduction algorithm requires it or if by executing the transition the number of context switches of the current execution exceeds the context-bound. We now present an example to motivate why this naive approach is unsound.

The program in Figure 1 has two threads, `T1` and `T2`. Let us perform depth-first search with context-bound 0 while using the sleep set algorithm [8] for performing partial-order reduction. We use the sleep set algorithm to illustrate the difficulty; we believe that a similar issue would arise even with other partial-order reduction algorithms. The state-space graph explored during the algorithm is shown on the right. In each state, thread `T1` is executed before thread `T2`. Context-bounding does not prune the transition $c \rightarrow d$ since the previously executing thread `T1` is disabled in state c. The transition $b \rightarrow e$ is pruned due to context-bounding. When the transition $a \rightarrow f$ of thread `T2` is executed the sleep set $\{$`T1`$\}$ is passed along because the action `x := 1` of `T1` is locally independent with the action `y := 2` of `T2`. Consequently, the transition $f \rightarrow e$ is pruned. As a result, the execution in which the final value of `y` is 1 is never explored although the execution `y := 2`, `x := 1`, `y := 1` has context-bound 0. Thus, we find that the simple approach for integrating context-bounding and partial-order reduction is unsound.

In Section 3, we present Algorithm 1 which soundly incorporates partial-order reduction in context-bounded search. For any given context-bound c, Algorithm 1 explores precisely once each partial-order, including all of its prefixes, with context-bound c or less. For detecting safety violations such as data-races, local assertion failures, and deadlocks, it suffices to visit only the terminal partial-orders with no successors. We present Algorithm 2, an improved algorithm that visits each terminal partial-order precisely once and significantly reduces the number of visited prefixes.

The key innovation that enables Algorithms 1 and 2 is that we represent a state as a happens-before graph [10] and use efficient and incremental algorithms (described in Section 3) for storing these graphs in a hashtable. Context-bounded exploration provides a polynomial bound on the number of happens-before graphs and consequently makes it feasible to scale this approach to large programs. Our algorithms need to compute the context-bounds of each partial-order encountered during the search. We first show that finding the context-bound of a partially-ordered execution is NP-complete. We then present an algorithm based on dynamic programming for solving this problem. By using memoization, we are able to reuse the work performed in this computation across various executions.

We present the asymptotic complexity of our algorithms on the class of *terminating* multithreaded programs. This class of programs is interesting because the testing harnesses of concurrent software components in the industry are invariably terminating. Suppose P is a terminating multithreaded program with a maximum of n threads, where each thread performs a maximum of k steps. Then, the total number of executions of P can be as large as $\Omega(n^k)$. Even though the number of threads is usually small, the number of steps can be very large and consequently the number of executions with its exponential dependence on k is huge. If c is the context-bound provided to our algorithms, the asymptotic complexity of state exploration is $O(n^{c+3}.k^{n+c+1}.(n+c)!)$, which although ex-

ponential in c and n is polynomial in k, which is crucial for scaling to large programs.

2 Multithreaded programs

In this paper, we are interested in systematic state exploration of terminating multithreaded programs. We identify a program state with the partially-ordered set of actions that happened since the beginning of the program. This notion (formally defined below) is different from the usual notion of a state being a vector of variables. Our notion of a state is appealing in the context of model checking of software implementations [7, 4, 9, 22, 13], for which it is often extremely difficult to collect all the variables defining the current state. For example, the state of a concurrent user-mode WIN32 process comprises both its address space as well as the state corresponding to any kernel resources allocated by it. However, it is difficult for a model checker to access or modify the internal state of the kernel.

Let us fix a set *Tid* of *thread identifiers*, a set *Act* of actions, and a function $T : Act \to Tid$ that provides for each action in *Act* the thread that performs it.

A *state* is a finite graph $\langle A, H\rangle$, where $A \subseteq Act$ is a set of actions and H is an irreflexive partial order on A that for each $t \in Tid$ is total on the set $\{a \in A \mid T(a) = t\}$ of actions in A performed by t. A set $B \subseteq A$ is *prefix-closed* if whenever $(a, b) \in H$ and $b \in B$ then $a \in B$. A state $\langle A', H'\rangle$ is a *prefix* of state $\langle A, H\rangle$ iff A' is a *prefix-closed* strict subset of A and H' is the restriction of H on A'. We write $s' < s$ if s' is a prefix of s. If $\langle A', H'\rangle < \langle A, H\rangle$ and $A \setminus A'$ is some singleton set $\{a\}$, then we write $\langle A', H'\rangle \xrightarrow{T(a)} \langle A, H\rangle$. Intuitively, thread $T(a)$ takes a step to evolve the state from $\langle A', H'\rangle$ to $\langle A, H\rangle$. In this case, $\langle A', H'\rangle$ is a *predecessor* of $\langle A, H\rangle$ and $\langle A, H\rangle$ is a *successor* of $\langle A', H'\rangle$. If the action a is a *read* of variable v by thread t, then H' contains extra edges to a from the last action performed by t and from the last write action on v. Similarly, if the action a is a *write* of variable v by thread t, then H' contains extra edges to a from the last action performed by t and from the last read or write action on v. In Section 3.3, we will formally define how these edges are added.

A thread that can take a step in a state s is said to be *enabled* in s. We denote the set of enabled threads in s by $E(s)$. We assume that if a thread t is enabled in a state s, then the state resulting from a step of t is unique; this state is denoted by $next(s, t)$. We denote the set of all successors and predecessors of a state s by $succ(s)$ and $pred(s)$, respectively.

A *multithreaded program* starts execution in the state $s_I = \langle A_I, H_I\rangle$ where A_I is the empty set and H_I is the empty relation. An *execution* is a sequence $s_I \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots \xrightarrow{t_{n-1}} s_{n-1} \xrightarrow{t_n} s_n$ for some $n > 0$. The *length* of an execution α, denoted by $len(\alpha)$, is the number of steps in it. We write $\alpha \xrightarrow{t} s$ to denote the extension of the execution α by a single step of thread t. The final state of α is denoted by $final(\alpha)$. The thread that performs the last step in α is denoted by $last(\alpha)$.

Informally, a *context-bound* of an execution is the number of preempting context-switches in the execution. Such context-switches occur when a thread

executing an action is different from the thread that executed the previous action, despite the latter being enabled. Formally, the context-bound of an execution α, denoted by $CB(\alpha)$, is defined recursively as follows:

$$\begin{aligned} CB(s_I \xrightarrow{t} s) &= 0 \\ CB(\alpha \xrightarrow{t} s) &= CB(\alpha) + ite(t = last(\alpha) \vee last(\alpha) \notin E(final(\alpha)), 0, 1) \end{aligned}$$

We extend the function CB to apply to program steps. We define $CB(s,t)$, when $t \in E(s)$, to be the least value of $CB(\alpha)$ over all α whose last step is $s \xrightarrow{t} next(s,t)$. Finally, we extend the function CB to apply to program states. We define $CB(s_I) = 0$. For any state $s \neq s_I$, we define $CB(s)$ to be the least value of $CB(\alpha)$ over all executions α such that $final(\alpha) = s$. Obviously, $CB(s)$ is also the least value of $CB(s',t)$ over all s' and t such that $next(s',t) = s$.

Given a state s, define the *frontier* $F(s) = \{t \mid \exists s'.s' \xrightarrow{t} s\}$. If $t \in F(s)$, then let $s - t$ denote the unique predecessor of s such that $s - t \xrightarrow{t} s$. From the definitions above, it is easy to show that the following recursive function computes $CB(s,t)$ for a $t \in E(s)$.

$$\begin{aligned} CB(s_I, t) &= 0 \\ CB(s, t) &= min_{t' \in F(s)}\, CB(s - t', t') + ite(t' = t \vee t' \notin E(s), 0, 1) \end{aligned}$$

3 Algorithm

In this section, we describe algorithms for performing partial-order reduction during context-bounded state-space search. We fix for the rest of this section a multithreaded program P and a context-bound $csb \geq 0$. We wish to search the state space of P and discover all reachable states s such that $CB(s) \leq csb$.

One complication in designing such a search algorithm is that the context-bound is not monotonic across transitions. Figure 2 shows an example of two states c and d such that $c \xrightarrow{T1} d$ but $CB(c) > CB(d)$. At first sight, this seems to imply that a model checker should visit states with arbitrary context-bounds in order to reach those within a smaller context bound. However, we show that every reachable state s of P such that $CB(s) \leq csb$ is indeed reachable via an execution in which every state has a context-bound of csb or less. This is a direct result of Theorem 1 presented below.

An execution $s_I = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-2}} s_{n-1} \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$ is *monotonic*, if $CB(s_{i-1}, t_{i-1}) \leq CB(s_i, t_i)$ for all $1 \leq i \leq n$.

Theorem 1 (Monotonicity). *For each execution α there is a monotonic execution α' such that the last step of α and α' are identical.*

Proof. We will prove the theorem by induction on the length of α. The base case when $len(\alpha) = 1$ is trivial. For the inductive case, let $n > 1$ and suppose the theorem holds for all α such that $len(\alpha) = n-1$. Suppose β is an execution such that $len(\beta) = n$ and the last step of β is $s_n \xrightarrow{t_n} s_{n+1}$. Let β' be an execution

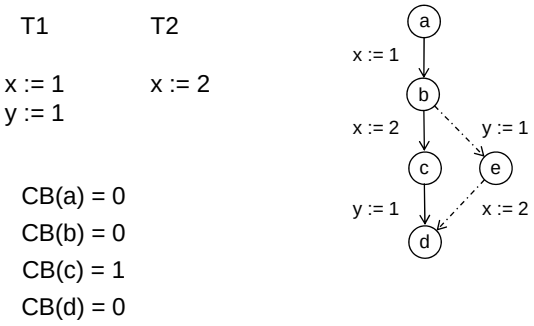

Fig. 2. This figure shows a state c such that executing thread $T1$ from c results in state d with a smaller context-bound. However, the state d is also reachable through an equivalent monotone execution through e.

$s_I = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-2}} s_{n-1} \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$ such that $CB(s_n, t_n) = CB(s_{n-1}, t_{n-1}) + ite(t_n = t_{n-1} \vee t_{n-1} \notin E(s_n), 0, 1)$. Let α' be the execution obtained by removing the last step from β'. Since $len(\alpha') = n - 1$, the theorem holds for α'. Therefore, there is another monotonic execution α'' such that the last steps of α'' and α' are identical. Since $CB(s_{n-1}, t_{n-1}) \leq CB(s_n, t_n)$, we get $\alpha'' \xrightarrow{t_n} s_{n+1}$ as a monotonic execution sequence whose last step is identical to β. □

An execution $s_I = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-2}} s_{n-1} \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$ is *csb*-bounded if $CB(s_i, t_i) \leq csb$ for all $0 \leq i \leq n$. The following observation is a simple corollary of Theorem 1.

Corollary 1. *For each execution α such that $CB(final(\alpha)) \leq csb$, there is a csb-bounded execution sequence α' such that $final(\alpha') = final(\alpha)$.*

Corollary 1 forms the basis for the correctness of Algorithm 1. This algorithm performs a depth-first search using the hashtable H to store the visited states. The guard on line 6 prunes a transition from state s either if the successor has already been visited or if the context-bound of the transition is greater than *csb*. The computation of the context-bound of a state required to implement the test $CB(s, t) > csb$ is described in Section 3.2. The design of a canonical state representation required to implement the hashtable-lookup operation $next(s, t) \in H$ is described in Section 3.3. We postpone a discussion of the computational complexity of Algorithm 1 to Section 3.4.

To prove the correctness of Algorithm 1, we introduce a few definitions. We define $Reach(s)$ to be the set of states reachable from s by following steps whose context-bound is *csb* or less. Formally, the set $Reach(s)$ is $\emptyset$ if $CB(s) > csb$ and is the least fixpoint of the following equation if $CB(s) \leq csb$:

$$Reach(s) \equiv \{s\} \cup \{s' \mid \exists x \in Reach(s), t \in E(x). CB(x, t) \leq csb \wedge s' = next(x, t)\}$$

```
1  Hashtable H;
2  Search (s_I);
   requires  : ¬(s ∈ H ∨ CB(s) > csb)
   ensures   : s ∈ H
3  Search(state s) begin
4      H = H ∪ {s} ;
5      for t ∈ E(s) do
6          if ¬(next(s,t) ∈ H ∨ CB(s,t) > csb) then
7              Search(next(s,t))
8          end
9      end
10 end
```

Algorithm 1: Basic algorithm

We now define the predicate $Closure(u)$ for any state u.

$$Closure(u) \equiv CB(u) \leq csb \wedge \forall v \in Reach(u) : v \in H$$

In our proof, we refer to an auxiliary variable $S \subseteq \mathsf{state}$ that represents the set of states on the stack of Search. The proof depends on the precondition and postcondition of Search given in the figure and the global invariant

$$CI \;\equiv\; \forall u \in H : u \in S \vee Closure(u).$$

Theorem 2 (Correctness). *Algorithm 1 terminates. At termination, for every state s, we have $s \in H$ iff there is an execution α such that $final(\alpha) = s$ and $CB(s) \leq csb$.*

Proof. Termination is straightforward for terminating programs.

Clearly CI holds initially because H is empty. The precondition holds at the call on line 2 because H is empty and $CB(s_I) = 0 \leq csb$. At line 4, when s is added to H it is added to S simultaneously and therefere CI is preserved. The precondition to Search for each call on line 7 is satisfied because $CB(next(s,t)) \leq CB(s,t) \leq csb$. Moreover, CI is preserved by each call to Search (by induction). We show that $Closure(s)$ holds at line 10. By the precondition of Search, we have $CB(s) \leq csb$. Suppose $v \in Reach(s)$. Then either $v = s$ or there is $t \in E(s)$ such that $CB(s,t) \leq csb$ and $v \in Reach(next(s,t))$. In the first case, we have $v = s \in H$ because s was added to H on line 4 and states are never removed from H. In the second case, we know that $next(s,t) \in H$ because either $next(s,t) \in H$ already on line 6 or Search$(next(s,t))$ was called on line 7 and the postcondition of Search guarantees that $next(s,t) \in H$. Since CI holds, we get $next(s,t) \in S \vee Closure(next(s,t))$. Clearly, $next(s,t) \notin S$ because S contains precisely the states on the stack and at line 10 $next(s,t)$ has been popped. Therefore, we get $Closure(next(s,t))$ and since $v \in Reach(next(s,t))$ we conclude that $v \in H$. Thus, $Closure(s)$ holds at line 10 and CI is preserved by the removal of s from S at line 10. Finally, the postcondition of Search is clearly satisfied because s was added to H at line 4 and not removed thereafter.

If $s \in H$, clearly there is an execution α with $final(\alpha) = s$ that was present on the stack when $\mathsf{Search}(s)$ was called and we get $CB(s) \leq csb$ from the precondition of Search. If there is an execution α with $final(\alpha) = s$ and $CB(s) \leq csb$, then Corollary 1 gives us that $s \in Reach(s_I)$. From the postcondition of Search, $s_I \in H$ when Algorithm 1 terminates. At the same time, S is empty at termination because the stack is empty. Therefore $Closure(s_I)$ holds at termination and we get $s \in H$. □

3.1 Improved algorithm

Algorithm 1 has the property that it visits each reachable state with context-bound at most *csb* precisely once. However, if the goal of the state-space search is to detect safety violations such as data-races, local assertion failures, and deadlocks, then it is sufficient but not necessary to visit each reachable state. In fact, it suffices to visit only the terminal states, which are states with no successors. Since the number of prefixes of a non-terminal state may be exponential in the size of the state, an algorithm that minimizes the number of explored prefixes may offer significant savings in practice. In this section, we present Algorithm 2, an improved algorithm in which the number of visited states is equal to the sum of the sizes of all terminal states with context-bound at most *csb*. In the worst case, Algorithm 2 still *processes* all the prefixes of visited states, but does not require driving the program to these non-terminal states. Thus, even though the two algorithms have similar asymptotic complexities, we expect Algorithm 2 to be more efficient for large programs.

The Search procedure in Algorithm 2 does not put any state in the hashtable H directly; rather it calls the procedure Mark to do it. If $E(s) \neq \emptyset$, the behavior of Search is the same as before. If $E(s) = \emptyset$, then Mark is invoked on s. Such a state s with no successors is called a terminal state. The goal of Mark is to add s to H and then further add all states from which it is impossible to execute the program and arrive at a *csb*-bounded terminal state that has not been visited before. Mark achieves this task by a depth-first search of all prefixes of s.

To understand the invariants behind the correctness of Algorithm 2, we define the set $TerminalReach(s) = \{s' \in Reach(s) \mid E(s') = \emptyset\}$. We now define the predicate $TerminalClosure(s)$ for any state s.

$$TerminalClosure(s) \equiv CB(s) \leq csb \wedge \forall v \in TerminalReach(s) : v \in H$$

The correctness of the algorithm relies on the following invariant, ensured by Mark when it returns.

$$Inv \;\equiv\; \forall u \in \mathsf{state} : u \in H \Leftrightarrow TerminalClosure(u).$$

Informally, this invariant states a state is in H iff all *csb*-bounded terminal successors of that state are also present in H. When Search calls Mark this invariant is temporarily violated but re-established by the time the call returns.

We will assume that Mark ensures *Inv* (which will be proved later) and use *Inv* to prove the following theorem that states the correctness theorem for Algorithm 2.

```
1  Hashtable H;
2  Search (s_I);
   requires : (SR) ¬(s ∈ H ∨ CB(s) > csb)
   ensures  : (SE) s ∈ H
3  Search(state s) begin
4      if succ(s) = ∅ then
5          Mark(s)
6      end
7      else
8          for t ∈ E(s) do
9              if ¬(next(s,t) ∈ H ∨ CB(s,t) > csb) then
10                 Search(next(s,t))
11             end
12         end
13     end
14 end
   requires : (MRa) ¬(s ∈ H ∨ CB(s) > csb)
   requires : (MRb) ∀t ∈ E(s) : next(s,t) ∈ H ∨ CB(s,t) > csb
   ensures  : (ME) s ∈ H
15 Mark(state s) begin
16     H = H ∪ {s} ;
17     for x ∈ {x | ∃t.next(x,t) = s ∧ CB(x,t) ≤ csb} do
18         if ∀t' ∈ E(x) : next(x,t') ∈ H ∨ CB(x,t') > csb then
19             Mark(x)
20         end
21     end
22 end
```

Algorithm 2: Improved algorithm

Theorem 3 (Correctness). *Algorithm 2 terminates. At termination, for every state s, we have $s \in H$ iff there is an execution α such that $final(\alpha) = s$ and $CB(s) \leq csb$.*

Proof. Termination is straightforward for terminating programs. We first show that the precondition SR holds before each call to Search and postcondition SE holds when Search returns.
[SR at Line 2] The precondition of Search is satisfied at the call on line 2 because H is empty and $CB(s_I) = 0 \leq csb$.
[SE at Line 6] The postcondition of Search follows from ME at line 6.
[SR at Line 10] We have $next(s,t) \notin H$ from the guard at line 9. Similarly, $CB(next(s,t)) \leq CB(s,t) \leq csb$. The precondition of Search follows.
[SE at Line 13] If line 8 is executed, then we first show that $TerminalClosure(s)$ holds at line 13. Clearly $CB(s) \leq csb$ from the precondition of Search. Consider an arbitrary $v \in TerminalReach(s)$. Since $E(s) \neq \emptyset$, we have $v \neq s$ and there exists a thread $t \in E(s)$ such that $CB(s,t) \leq csb$ and $v \in TerminalReach(next(s,t))$. Since $CB(s,t) \leq csb$, either $next(s,t) \in H$ at line 9 or Search$(next(s,t))$ is called

at line 10. From the postcondition of Search and the fact states are never removed from H, we get that in either case $next(s,t) \in H$ at line 13. From Inv, we get that $TerminalClosure(next(s,t))$ holds at line 13 and therefore $v \in H$. Thus we get that $TerminalClosure(s)$ holds at line 13. Consequently, from Inv we get that $s \in H$ at line 13 and the postcondition of Search is satisfied.

Now, we prove the statement of the theorem. If $s \in H$, then s was added to H by Mark whose precondition gives us that $CB(s) \leq csb$. Moreover, Mark is called either on a terminal state or a prefix of a terminal state. In either case, there is an execution α with $final(\alpha) = s$. If there is an execution α with $final(\alpha) = s$ and $CB(s) \leq csb$, then Corollary 1 gives us that $s \in Reach(s_I)$. From the postcondition of Search, $s_I \in H$ when Algorithm 2 terminates. Therefore, at termination $TerminalClosure(s_I)$ holds which implies that $TerminalClosure(s)$ also holds. From Inv we get $s \in H$. □

Next, we show that the search is never blocked until execution reaches a terminal state not visited before.

Theorem 4 (Nonblocking search). *Whenever* Search(s) *is invoked either* $E(s) = \emptyset$ *or there is a successor* $next(s,t)$ *of* s *such that* $\neg(next(s,t) \in H \vee CB(s,t) > csb)$.

Proof. Suppose Search(s) is invoked on a state s such that $E(s) \neq \emptyset$. From SR, we get that $s \notin H$ and $CB(s) \leq csb$. Therefore, from Inv, there exists $x \in TerminalReach(s)$ such that $x \notin H$. Since s is not a terminal state, there is a thread t such that $CB(s,t) \leq csb$ and $x \in TerminalReach(next(s,t))$. From Inv, we get $next(s,t) \notin H$. □

Theorem 4 guarantees that the number of times Search is called is proportional to the sum of the sizes of all terminal states with context-bound at most csb.

Finally, we show the correctness of Mark by showing that it ensures Inv when it returns. We need the following lemmas for this proof.

Lemma 1 (Local monotonicity). *For all states* s *and threads* $t \in E(s)$*, we have* $CB(s) \leq CB(s,t)$.

Proof. Let $t \in E(s)$ for some state s. If $s = s_I$, then $CB(s) = 0$ and $CB(s,t) = 0$ and therfore $CB(s) \leq CB(s,t)$. Otherwise, there is $x \in pred(s)$ such that $CB(s,t) = CB(x,u) + ite(t = u \vee u \notin E(s), 0, 1)$, which implies that $CB(x,u) \leq CB(s,t)$. By the definition of $CB(s)$, we have $CB(s) \leq CB(x,u)$. Thus we get $CB(s) \leq CB(s,t)$. □

Lemma 2 (Progress). *For all states* s*, either* $E(s) = \emptyset$ *or there exists* $t \in E(s)$ *such that* $CB(s,t) = CB(s)$.

Proof. Suppose $E(s) \neq \emptyset$. If $s = s_I$, then for all $t \in E(s)$ we have $CB(s,t) = CB(s) = 0$. Otherwise, there is a predecessor x of s and a thread t such that $s = next(x,t)$ and $CB(x,t) = CB(s)$. If $t \in E(s)$, then $CB(s,t) \leq CB(x,t) =$

$CB(s)$ and from Lemma 1 we get $CB(s,t) = CB(s)$. If $t \notin E(s)$, then there exists $u \in E(s)$ and $CB(s,u) \leq CB(x,t) = CB(s)$ and from Lemma 1 we get $CB(s,u) = CB(s)$. □

To state the correctness invariant of Mark, we need an auxiliary variable S that precisely captures the set of states on the stack of Mark. We show that Mark maintains the following invariant MI at all instants, which by Lemma 3 implies Inv when Mark returns.

$$
\begin{aligned}
&MI \equiv MIa \wedge MIb \wedge MIc\\
&MIa \equiv \forall u \in \mathsf{state} : u \in H \Rightarrow CB(u) \leq csb\\
&MIb \equiv \forall u \in \mathsf{state} : u \in H \Rightarrow \forall t \in E(u) : next(u,t) \in H \vee CB(u,t) > csb\\
&MIc \equiv \forall u \in \mathsf{state} : \begin{pmatrix} E(u) \neq \emptyset \\ \wedge\ CB(u) \leq csb \\ \wedge\ \forall t \in E(u) : next(u,t) \in H \setminus S \vee CB(u,t) > csb \end{pmatrix} \Rightarrow u \in H
\end{aligned}
$$

Lemma 3. $S = \emptyset \wedge MI \Rightarrow Inv$.

Proof. The proof is by well-founded induction on the on the inverse of the prefix partial-order $<$. That is, we show for any state u that if $\forall v \in succ(u) : v \in H \Leftrightarrow TerminalClosure(v)$ then $u \in H \Leftrightarrow TerminalClosure(u)$.

For the base case, consider a state u such that $E(u) = \emptyset$. If $u \in H$, then MIa gives us that $CB(u) \leq csb$. Moreover, $TerminalReach(u) = \{u\}$. Thus, we have $TerminalClosure(u)$. If $TerminalClosure(u)$ holds then $u \in TerminalReach(u)$ and therefore $u \in H$.

For the inductive case, suppose u is a state such that $E(u) \neq \emptyset$ and $\forall v \in succ(u) : v \in H \Leftrightarrow TerminalClosure(v)$. Suppose $u \in H$. From MIa, $CB(u) \leq csb$. We now show for an arbitrary $x \in TerminalReach(u)$ that $x \in H$. Since $E(u) \neq \emptyset$, we have $x \neq u$. Therefore, there is a thread t and a state v such that $CB(u,t) \leq csb$, $next(u,t) = v$, and $x \in TerminalReach(v)$. From MIb, we get $v = next(u,t) \in H$ and the induction hypothesis gives us $x \in H$. Suppose $TerminalClosure(u)$ holds. We show that $\forall t \in E(u) : next(u,t) \in H \vee CB(u,t) > csb$. Consider an arbitrary $t \in E(u)$ such that $CB(u,t) \leq csb$. Since $TerminalClosure(u)$ holds $TerminalClosure(next(u,t))$ also holds. By the induction hypothesis, we have $next(u,t) \in H$. Thus, MIc and $S = \emptyset$ gives us that $E(u) = \emptyset$ or $u \in H$. Since we are in the inductive case, we get $u \in H$. □

Lemma 4. *MI holds initially and* Mark *satisfies its preconditions and postcondition and preserves MI.*

Proof. [MI at Line 2] Since H is empty initially, MIa and MIb hold trivially. We now show that MIc holds initially. Consider a state u such that $E(u) \neq \emptyset$ and $CB(u) \leq csb$. By Lemma 2, there exists $t \in E(u)$ such that $CB(u,t) = CB(u) \leq csb$. Therefore MIc holds.

Since Search does not modify H or S, MI trivally holds whenever the control is in Search. We only need to show that MI is preserved whenever the control is in Mark.

[*MRa* at Line 5] $SR \Rightarrow MRa$
[*MRb* at Line 5] From the guard at line 4, $E(s) = \emptyset$. This trivially implies *MRb*.
[*MRa* at Line 19] Let t be such that $next(x,t) = s$. We know that at line 16, both *MRa* and *MIb* hold. *MRa* gives us $s \notin H$ at line 16. Since $next(x,t) = s \notin H$ and $CB(x,t) \leq csb$, *MIb* gives us that $x \notin H$ at line 16. Furthermore, a call $\mathsf{Mark}(x')$ at line 19 for some predecessor x' of s different from x cannot add x to H. Therefore, $x \notin H$ at line 19. The set constructor on line 17 ensures $CB(x,t) \leq csb$. From Lemma 1, we get that $CB(x) \leq CB(x,t) \leq csb$ holds at line 19.
[*MRb* at Line 19] The final precondition of Mark is ensured by the guard on line 18.

We now argue that *MI* is preserved by each statement in the body of Mark.
[*MIa* at Line 17] The statement on line 16 adds s to H. $MRa \Rightarrow MIa$
[*MIb* at Line 17] $MRb \Rightarrow MIb$
[*MIc* at Line 17] Since s is added to S and H simultaneously at line 16, *MIc* trivally follows.
[$MIa \wedge MIb$ at Line 22] At line 22, state s is removed from S. Both *MIa* and *MIb* are not affected by S and thus preserved.
[*MIc* at Line 22] Let u be an arbitrary state such that $\forall t \in E(u).next(u,t) \in H \setminus (S \setminus \{s\}) \vee CB(u,t) > csb$ holds at line 22. We now prove that $u \in H$. Suppose for all $t \in E(u)$, we had that $next(u,t) = s$ implied $CB(u,t) > csb$. Then the condition $\forall t \in E(u).next(u,t) \in H \setminus (S \setminus \{s\}) \vee CB(u,t) > csb$ simplifies to $\forall t \in E(u).next(u,t) \in H \setminus S \vee CB(u,t) > csb$ and we are done. Otherwise there exists $t' \in E(u)$ such that $next(u,t') = s$ and $CB(u,t') \leq csb$. Therefore u will be considered on line 18. Moreover, the existence of such a t' means that the condition $\forall t \in E(u).next(u,t) \in H \setminus (S \setminus \{s\}) \vee CB(u,t) > csb$ simplifies to $\forall t \in E(u).next(u,t) \in H \vee CB(u,t) > csb$. If the condition $\forall t \in E(u).next(u,t) \in H \vee CB(u,t) > csb$ holds at line 22, then it also holds at line 18 because once s is added to H at line 16, every other state added to H during the execution of this call to Mark is different from $next(u,t)$. Therefore, $\mathsf{Mark}(u)$ is called on line 19. The postcondition of Mark ensures that $u \in H$ after this call. Since states are never removed from H, we get that $u \in H$ at line 22.
[*ME* at Line 22] The postcondition trivally follows from the fact that line 16 adds s to H. □

3.2 Context-bound computation

In this section, we present a method to calculate the context-bound of a state. This computation is required in both Algorithms 1 and 2. Unfortunately, the following theorem shows that estimating the context-bound of a state is an NP-complete problem.

Theorem 5. *Given a state s and an integer $c \geq 0$, the problem of determining whether $CB(s) \leq c$ is NP-complete.*

Proof. The proof is by reduction from the minimum feedback-vertex set problem: Given a directed graph $G(V,E)$, find a subset V' of V of size c such that every

directed cycle of G contains at least one vertex in V'. This problem is known to be NP-complete.

Given a directed graph $G(V, E)$ build a state $\langle A, H\rangle$ as follows. For every vertex $v \in V$, create a thread that contains two actions v_{src} followed by v_{dst} and add the intra-thread edge (v_{src}, v_{dst}) to H. For every edge $(u, v) \in E$, create an inter-thread edge (u_{src}, v_{dst}) to H.

We claim that G contains a feedback-vertex set of size c iff there is a linearization of $\langle A, H\rangle$ with context-bound c. Suppose V' is a feedback-vertex set of G of size c. Then $G - V'$ is a directed acyclic graph. We can construct a linearization of $\langle A, H\rangle$ with a context-bound c as follows. The linearization starts by scheduling v_{src} for every $v \in V'$. Then, we schedule (v_{src}, v_{dst}) for every $v \in V \setminus V'$ in the order of some linearization of $G - V'$. Finally, we schedule v_{dst} for every $v \in V'$. The linearization contains c context switches for every vertex in V'. Similarly, we can show that starting from a linearization of $\langle A, H\rangle$ with context-bound c, we can construct a feedback-vertex set of size c by adding all the threads that got preempted in V'. □

Given our NP-completeness result, it is unlikely that there is a polynomial-time algorithm for calculating the context-bound of a state. We use the recursive definition of $CB(s, t)$ presented in Section 2 to derive an algorithm based on dynamic programming and memoization that could take exponential time and space in the worst case.

Let p be the number of prefixes of s and f be the maximum frontier size among all prefixes of s. Then, the complexity of computing $CB(s)$ is proportional to $p \cdot f$. While f is bounded by the number of threads involved in s, p may be exponential in the number of actions in s.

In Algorithms 1 and 2, the context-bound is computed for all prefixes of the visited states. By memoizing the value returned by a call to CB, work performed during its computation can be reused across context-bound computations. As a result, the amount of performed work amortized across all context-bound calculations is proportional to the number of prefixes of visited states.

3.3 Canonical representation of states

In this section, we present an efficient method for implementing the hashtable operations needed in Algorithms 1 and 2. Towards this end, we introduce more structure into the actions of Act. An action $a \in A$ is a tuple $\langle o, t, n, v, m\rangle$, where $o \in \{R, W\}$, $t \in Tid$, $n \in \mathcal{N}$, $v \in Var$, and $m \in \mathcal{N}$. If $o = W$, then this action is the m-th write to the variable v and is the n-th action performed by thread t. If $o = R$, then this action is a read of the value written by the m-th write to variable v and is the n-th action performed by thread t. For consistency, we require that $T(\langle o, t, n, v, m\rangle) = t$.

A state $\langle A, H\rangle$ is now canonically represented by the set A; the edge information in H is automatically encoded in the structure of the actions in A. We now show how we can recover H given the set A. Let $OP(\langle o, t, n, v, m\rangle) = o$, $TC(\langle o, t, n, v, m\rangle) = n$, and $VC(\langle o, t, n, v, m\rangle) = m$. Then, H is the transitive

closure of the set $\bigcup_{t \in Tid} TE_t \cup \bigcup_{v \in Var} VE_v$ of edges, where TE_t and VE_v are defined below.

$$\begin{array}{rcl} TE_t & = & \{(a,b) \subseteq A \times A \mid TC(a) < TC(b)\} \\ VE_v & = & \{(a,b) \subseteq A \times A \mid OP(a) = W \wedge VC(a) \leq VC(b)\} \cup \\ & & \{(a,b) \subseteq A \times A \mid OP(b) = W \wedge VC(a) < VC(b)\} \end{array}$$

Let h be a universal hashing function that maps an action $\langle o, t, n, v, m \rangle$ to a 32-bit integer. We lift h to a set of actions A by applying h to each element of A and then performing the bitwise XOR operation $\oplus$ on the results. Since $\oplus$ is commutative and associative, this lifting procedure defines a function. Note that this hash function is incremental by design. During program execution, suppose a thread takes a step to extend the state s by a single action a into s'. Then $h(s') = h(s) \oplus h(a)$ and $h(s) = h(s') \oplus h(a)$. These observations are used in Search and Mark for incremental hash computation.

3.4 Complexity analysis

This section presents the asymptotic complexity of Algorithm 1 and Algorithm 2. Given a terminating multithreaded program with n threads, each executing at most k steps, let $NCB(n, k, c)$ be the number of terminal executions of the program with a context-bound c. Each of these executions has at most $n.k$ steps, resulting in at most $n.k.NCB(n, k, c)$ states that have a context-bound less than or equal to c.

When the Search procedure is called on a state s in Algorithm 1, the guard in line 6 ensures that the recursive call of Search happens only when there is an enabled thread t such that $CB(s, t) \leq c$. By definition, $CB(next(s, t)) \leq CB(s, t)$. Also, $CB(s_I) = 0 \leq c$. Therefore, Algorithm 1 calls Search(s) only when $CB(s) \leq c$. Moreover, maintaining the hash table ensures that Search is called for each state at most once. Thus, the number of calls to Search is $O(n.k.NCB(n, k, c))$. Each such call to a state s involves at most n operations on the hashtable H and the computation of $CB(s, t)$ for every enabled thread t. While the hash computation of a state can be done incrementally in constant time, we need to check equality of two states. Assuming that hash-collisions are rare, the cost of the hash table operations is $O(n^2.k)$. In addition, the computation of $CB(s, t)$ can visit all prefixes of s, requiring a total of $O(k^n)$ cost, which dwarfs the cost of hash table operations.

Thus, Algorithm 1 has a complexity of $O(n.k^{n+1}.NCB(n, k, c))$. Finally, in our previous work [14], we show that $NCB(n, k, c) = O((n.k)^c.(n + c)!)$ for nonblocking programs. This results in a bound of $O(n^{c+1}.k^{n+c+1}.(n + c)!)$.

The complexity analysis for Algorithm 2 is similar. The additional observation required is that the Mark procedure is only called for prefixes of a terminal state with context-bound less than or equal to c. Thus there are $k^n.NCB(n, k, c)$ calls to Mark. Also, as the calls to CB are amortized over prefixes, the cost of CB computation is constant for each call to Mark. There are at most n^2 hash computations in Mark resulting in a complexity of $n^3.k^{n+1}.NCB(n, k, c)$. In contrast

Context Bound	Number of terminal executions					
	size = 3		size = 4		size = 5	
	cb-dfs	cb-por	cb-dfs	cb-por	cb-dfs	cb-por
0	2	2	2	2	2	2
1	46	46	54	54	67	67
2	756	713	898	853	1197	1136
3	8918	7838	12711	8458	19460	14563
4	66374	48828	95935	52572	161637	95068

Table 1. Experimental evaluation of Algorithm 2 for a work-stealing queue concurrent data structure implementation.

to Algorithm 1, the number of calls to Search is bounded by $n.k.NCB(n,k,c)$, which is small compared to the complexity of Mark. Thus the complexity of Algorithm 2 is $O(n^{c+3}.k^{n+c+1}.(n+c)!)$.

4 Experimental Evaluation

We have implemented the algorithms discussed in Section 3 in the CHESS model checker. CHESS is a stateless model checker akin to Verisoft [7], and is designed for systematically exploring the behaviors of a multi-threaded software implementations.

For the evaluation in this section, we used an implementation [12] of the work-stealing queue for the Cilk multithreaded programming system [6]. The implementation consists of a queue of work items in a bounded circular buffer and is around 1000 lines of C++ code. The test driver used in our experiments involves a main thread that enqueues jobs into the queue and later dequeues them. Concurrently, a stealer thread that accesses the queue to *steal* jobs for processsing.

Table 1 shows the results of our experiments for various sizes of the work-stealing queue. We compared Algorithm 2, labeled `cb-por` in the table, with a basic context-bounding algorithm that prunes executions exceeding the bound while performing a depth-first search. The latter is labeled `cb-dfs` in the table. For each experiment, the table reports the number of terminal executions of the program for each context-bound.

From Table 1, we see that for small context-bounds, there is negligible difference between the number of executions with and without partial-order reduction. This reflects a nice property of context-bounded search — for small context-bounds, the search does not explore redundant behaviors of the program. For larger context-bounds however, we see the benefit of performing partial-order reduction. For instance, with a context-bound of 4 around 40% of the executions performed by `cb-dfs` are redundant. The `cb-por` algorithm, in contrast, is guaranteed to never produce two partial-order equivalent executions.

Encouraged by our initial results, we are currently evaluating the algorithms presented in this paper on larger programs. We believe that incorporating partial-

order reduction is very crucial to scaling context-bounded explorations for large context-bounds.

5 Related work

Context-bounded verification. Context-bounding was introduced by Qadeer and Wu [18] as a technique for transforming the problem of verifying multithreaded programs to that of verifying sequential programs. Later, Qadeer and Rehof [17] showed that context-bounded verification of concurrent boolean programs is decidable; Bouajjani et al. [1] improved their result. Musuvathi and Qadeer [14] showed that context-bounding increases the efficacy of explicit-state model checking in finding concurrency errors. None of the aforementioned papers have considered the problem of incorporating partial-order reduction in context-bounded reachability analysis. Context-bounding for multithreaded software is also analogous to bounded model checking [2] for synchronous hardware.

Partial-order reduction. A variety of partial-order reduction techniques have been proposed in the literature. The guarantees provided by Algorithm 1 is closest in spirit to that of the sleep set algorithm [8] in that each prefix of a partially-ordered execution is visited precisely once. However, the algorithmic mechanisms are completely different. Our algorithm performs caching and context-bound calculation for happens-before graphs, whereas the sleep set algorithm only needs to maintain a sleep set for each element in the search stack. Other algorithms [21, 16, 15] attempt more aggressive state-space reduction by avoiding visiting many prefixes. However, these algorithms are based on statically computed independence relations which are usually imprecise for software implementations. On the other hand, our algorithm is based on the happens-before graphs of dynamic executions. In this respect, Algorithm 2 is similar in spirit to recent work on dynamic partial-order reduction for multithreaded software [5, 11, 20]. As far as we know, none of these previous works have considered the subtle issues that arise when partial-order reduction is performed in context-bounded search, which is the main focus of this paper.

References

1. A. Bouajjani, J. Esparza, S. Schwoon, and J. Strejcek. Reachability analysis of multithreaded software with asynchronous communication. In *FSTTCS 05: Foundations of Software Technology and Theoretical Computer Science*, volume 3821 of *Lecture Notes in Computer Science*. Springer, 2005.
2. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
3. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.
4. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE 00: International Conference on Software Engineering*, pages 439–448. ACM, 2000.

5. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL 05: Principles of Programming Languages*, pages 110–121. ACM Press, 2005.
6. Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI 98: Programming Language Design and Implementation*, pages 212–223, 1998.
7. P. Godefroid. Model checking for programming languages using Verisoft. In *POPL 97: Principles of Programming Languages*, pages 174–186, 1997.
8. Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. LNCS 1032. Springer-Verlag, 1996.
9. Gerard J. Holzmann and Margaret H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Transactions on Software Engineering*, 28(4):364–377, 2002.
10. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
11. Yu Lei and Richard H. Carver. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, 32(6):382–403, 2006.
12. Daan Leijen. Futures: a concurrency library for C#. Technical Report MSR-TR-2006-162, Microsoft Research, 2006.
13. M. Musuvathi, D. Park, A. Chou, D. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI 02: Operating Systems Design and Implementation*, 2002.
14. Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI 07: Programming Language Design and Implementation*, 2007.
15. Ratan Nalumasu and Ganesh Gopalakrishnan. An efficient partial order reduction algorithm with an alternative proviso implementation. *Formal Methods in System Design*, 20(3):231–247, May 2002.
16. Doron Peled. Partial order reduction: Model-checking using representatives. In *MFCS 96: Mathematical Foundations of Computer Science*, pages 93–112. Springer-Verlag, 1996.
17. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS 05: Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005.
18. S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *PLDI 04: Programming Language Design and Implementation*, pages 14–24. ACM, 2004.
19. J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Fifth International Symposium on Programming*, Lecture Notes in Computer Science 137, pages 337–351. Springer-Verlag, 1981.
20. Koushik Sen and Gul Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Haifa Verification Conference*, 2006.
21. Antti Valmari. A stubborn attack on state explosion. In *CAV 91: Computer Aided Verification*, pages 156–165. Springer-Verlag, 1991.
22. W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ASE 00: Automated Software Engineering*, pages 3–12, 2000.