Iterative Context Bounding for Systematic Testing of Multithreaded Programs

Madan Musuvathi Shaz Qadeer
Microsoft Research
{madanm,qadeer}@microsoft.com

Abstract

Multithreaded programs are difficult to get right because of unexpected interaction between concurrently executing threads. Traditional testing methods are inadequate for catching subtle concurrency errors which manifest themselves late in the development cycle and post-deployment. Model checking or systematic exploration of program behavior is a promising alternative to traditional testing methods. However, it is difficult to perform systematic search on large programs as the number of possible program behaviors grows exponentially with the program size. Confronted with this state-explosion problem, traditional model checkers perform iterative depth-bounded search. Although effective for message-passing software, iterative depth-bounding is inadequate for multithreaded software.

This paper proposes iterative context-bounding, a new search algorithm that systematically explores the executions of a multithreaded program in an order that prioritizes executions with fewer *context switches*. We distinguish between preempting and nonpreempting context switches, and show that bounding the number of preempting context switches to a small number significantly alleviates the state explosion, without limiting the depth of explored executions. We show both theoretically and empirically that context-bounded search is an effective method for exploring the behaviors of multithreaded programs. We have implemented our algorithm in two model checkers and applied it to a number of real-world multithreaded programs. Our implementation uncovered 9 previously unknown bugs in our benchmarks, each of which was exposed by an execution with at most 2 preempting context switches. Our initial experience with the technique is encouraging and demonstrates that iterative context-bounding is a significant improvement over existing techniques for testing multithreaded programs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification — formal methods, validation; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs — mechanical verification, specification techniques; D.2.5 [*Software Engineering*]: Testing and Debugging — debugging aids, diagnostics, monitors, tracing

General Terms Algorithms, Reliability, Verification

Keywords Concurrency, context-bounding, model checking, multithreading, partial-order reduction, shared-memory programs, software testing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI '07 June 11–13, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-633-2/07/0006. . . $5.00.

1. Introduction

Multithreaded programs are difficult to get right. Specific thread interleavings, unexpected even to an expert programmer, lead to crashes that occur late in the software development cycle or even after the software is released. The traditional method for testing concurrent software in the industry is *stress-testing*, in which the software is executed under heavy loads with the hope of producing an erroneous interleaving. Empirical evidence clearly demonstrates that this form of testing is inadequate. Stress-testing does not provide any notion of coverage with respect to concurrency; even after executing the tests for days the fraction of explored schedules remains unknown and likely very low.

A promising method to address the limitations of traditional testing methods is *model checking* [2, 21] or systematic exploration of program behavior. A model checker systematically executes each thread schedule, while verifying that each execution maintains desired properties of the program. The fundamental problem in applying model checking to large programs is the well-known *state-explosion problem*, i.e., the number of possible program behaviors grows explosively (at least exponentially) with the size of the program.

To combat the state-explosion problem, researchers have investigated reduction techniques such as partial-order reduction [9] and symmetry reduction [13, 12]. Although these reduction techniques help in controlling the state explosion, it remains practically impossible for model checkers to fully explore the behaviors of large programs within reasonable resources of memory and time. For such large programs, model checkers typically resort to heuristics to maximize the number of errors found before running out of resources. One such heuristic is *depth-bounding* [22], in which the search is limited to executions with a bounded number of steps. If the search with a particular bound terminates, then it is repeated with an increased bound. Unlike other heuristics for partial state-space search, depth-bounded search provides a valuable coverage metric—if search with depth-bound d terminates then there are no errors in executions with at most d steps.

Since the number of possible behaviors of a program usually grows exponentially with the depth-bound, iterative depth-bounding runs out of resources quickly as the depth is increased. Hence, depth-bounding is most useful when interesting behaviors of the program, and therefore bugs, manifest in small number of steps from the initial state. The state space of message-passing software has this property which accounts for the success of model checking on such systems [10, 16]. In contrast, depth-bounding does not work well for multithreaded programs, where the threads in the program have fine-grained interaction through shared memory. While a step in a message-passing system is the send or receive of a message, a step in a multithreaded system is a read or write of a shared variable. Typically, several orders of magnitude more steps

are required to get interesting behavior in a multithreaded program than in a message-passing program.

This paper proposes a novel algorithm called *iterative context-bounding* for effectively searching the state space of a multithreaded program. In an execution of a multithreaded program, a *context switch* occurs when a thread temporarily stops execution and a different thread starts. The iterative context-bounding algorithm bounds the number of context switches in an execution. However, a thread in the program can execute an arbitrary number of steps between context switches, leaving the execution depth unbounded.

Furthermore, the iterative context-bounding algorithm distinguishes between two kinds of context switches — preempting and nonpreempting. A *preempting* context switch, or simply a *preemption*, occurs when the scheduler suspends the execution of the running thread at an arbitrary point. This can happen, for instance, at the expiration of a time slice. On the other hand, a *nonpreempting* context switch occurs when the running thread voluntarily yields its execution, either at termination or when it blocks on an unavailable resource. The algorithm bounds the number of preemptions while leaving the number of nonpreempting context switches unbounded.

Limiting the number of preemptions has many powerful and desirable consequences for systematic state-space exploration of multithreaded programs. First, bounding the number of preemptions does not restrict the ability of the model checker to explore deep in the state space. This is due to the fact that, starting from any state, it is always possible to drive a terminating program to completion (or to a deadlock state) without incurring a preemption. As a result, a model checker is able to explore interesting program behaviors, even with a bound of zero!

Second, we show (Section 2) that for a fixed number of preemptions, the total number of executions in a program is *polynomial* in the number of steps taken by each thread. 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.

Finally, iterative context-bounding has the important property that it finds a trace with the smallest number of preemptions exposing the error. As most of the complexity of analyzing a concurrent error-trace arises from the interactions between the threads, the algorithm naturally seeks to provide the simplest explanation for the error. Moreover, when the search runs out of resources after exploring all executions with *c* preemptions, the algorithm guarantees that any error in the program requires at least *c+1* preemptions. In addition to providing a valuable coverage metric, it also provides the programmer with an estimate of the complexity of bugs remaining in the system and the probability of their occurrence in practice.

We present our iterative context-bounding algorithm in Section 3. To evaluate our algorithm, we implemented it in two model checkers, ZING and CHESS. ZING is an explicit-state model checker for concurrent programs specified in the ZING modeling language. CHESS is a stateless model checker that executes the program directly, much along the lines of Verisoft [10], but designed for shared-memory multithreaded programs.

An important aspect of the CHESS implementation is its dynamic partitioning of the set of program variables into data and synchronization variables. Typical programs use synchronization variables, such as locks, events, and semaphores, to ensure that there are no data-races on the data variables. Motivated by this observation, CHESS introduces context switches only at accesses to synchronization variables and verifies that accesses to data variables are ordered by accesses to synchronization variables in each explored execution. In Section 3.1, we provide theoretical justification for the soundness of this approach.

Our evaluation (Section 4) provides empirical evidence that a small number of preemptions is sufficient to expose nontrivial concurrency bugs. Our implementation uncovered 9 previously unknown bugs in several real-world multithreaded programs. Each of these bugs was exposed by an execution with at most 2 preemptions. Also, for a set of programs for which complete search is possible, we show that few preemptions are sufficient to cover most of the state space. This empirical evidence strongly suggests that when faced with limited resources, which is invariably the case with model checkers, focusing on the polynomially-bounded and potentially bug-yielding executions with a small preemption bound is a productive search strategy.

In summary, the technical contributions of the paper are as follows:

- The notion of iterative context-bounding and the concomitant argument that bounding the number of preemptions is superior to bounding the depth as a strategy for systematic exploration of multithreaded executions.
- A combinatorial argument that for a fixed number of preemptions, the number of executions is polynomial in the total number of steps executed by the program.
- An iterative context-bounding algorithm that systematically enumerates program executions in increasing order of preemptions.
- Empirical evidence that context-bounded executions expose interesting behavior of the program, even when the number of preemptions is bounded by a small number.

2. Iterative context-bounding

In the view of this paper, model checking a multithreaded program is analogous to running the system on a nondeterministic scheduler and then systematically exploring each choice made by the scheduler. Each thread in the program executes a sequence of steps with each step involving exactly one access to a shared variable. After every step of the currently running thread, the scheduler is allowed to choose the next thread to schedule. As a result, the number of possibilities explodes exponentially with the number of steps. To make this point concretely, suppose P is a *terminating* multithreaded program. Let P have n threads where each thread executes at most k steps of which at most b are potentially-blocking. Then the total number of executions of P may be as large as $\frac{(nk)!}{(k!)^n} \geq (n!)^k$, a dependence that is exponential in both n and k. For most programs, although the number of threads may be small, the number of steps performed by a thread is very large. Therefore, the exponential dependence on k is especially problematic. All previous heuristics for partial state-space search, including depth-bounding, suffer from this problem.

The fundamental and novel contribution of context-bounding is that it *limits the number of scheduler choices without limiting the depth of the execution*. A context switch occurs at a schedule point if the scheduler chooses a thread different from the current running thread. This context switch is preempting if the running thread is enabled at the schedule point, otherwise it is nonpreempting.

In context-bounding, we bound the number of preempting context switches (or preemptions) but leave the number of nonpreempting context switches unconstrained. It is very important to note that the scheduler has a lot more choices in inserting preemptions — it can choose any one of the $n.k$ steps to preempt, and for each choice the scheduler can choose any of the enabled threads to run. In contrast, a nonpreempting context is forced on the scheduler when the running thread yields — its choice is limited to deciding the next enabled thread to run.

There are two important facts to note about context-bounding. First, the number of steps within each context remains unbounded. Therefore, unlike depth-bounding there is no bound on the execution depth. Second, since the number of nonprempting context switches remains unbounded it is possible to get a complete terminating execution even with a bound of zero. For instance, such a terminating execution can be obtained from any state by scheduling each thread in a round-robin fashion without preemption. These two observations clearly indicate that context bounding does not affect the ability of the search to go deep into the state space.

We show below that the number of executions of P with at most c preemptions is polynomial in k but exponential in c. An exponential dependence on c is significantly better than an exponential dependence on k because k is much greater than c. Moreover, many concurrency bugs are manifested when threads are preempted at unexpected places. With this polynomial bound, it becomes feasible to apply context-bounded search to large programs, at least for small values of c.

Let ${}^{x}\mathcal{C}_{y}$ denote the number of ways of choosing y objects out of a set of x objects.

THEOREM 1. *Consider a terminating program P with n threads, where each thread executes at most k steps of which at most b are potentially-blocking. Then there are at most ${}^{nk}\mathcal{C}_{c}(nb+c)!$ executions of P with c preemptions.*

PROOF. The length of each execution of P is bounded by nk. Therefore, there are are at most nk points where a preemption can occur and at most ${}^{nk}\mathcal{C}_{c}$ ways of selecting c preemptions from these nk points. Once the c preemptions have been chosen, we have a maximum of $nb+c$ execution contexts which can be arranged in at most $(nb+c)!$ ways. Thus, we get the upper bound of ${}^{nk}\mathcal{C}_{c}(nb+c)!$ executions with c preemptions. □

Assuming that c is much smaller than both k and nb, the bound given in the theorem above is simplified to $(nk)^{c}(nb)^{c}(nb)! = (n^{2}kb)^{c}(nb)!$. This bound remains exponential in c, n, and b, but each of these values is significantly smaller than k, with respect to which this bound is polynomial. It is also interesting to simplify this bound further for non-blocking multithreaded programs. In such programs, the only blocking action performed by a thread is the fictitious action representing the termination of the thread. Therefore $b = 1$ and the bound becomes $(n^{2}k)^{c}n!$.

2.1 Empirical argument

To evaluate the efficacy of iterative context-bounding in exposing concurrency errors, we have implemented the algorithm and used it to test several real-world programs. We describe our evaluation in detail in Section 4. Here we give a brief preview of the performance of our algorithm on an implementation [15] of a work-stealing queue algorithm [8]. This implementation represents the queue using a bounded circular buffer which is accessed concurrently by two threads in a non-blocking manner. The implementor gave us the test harness along with three variations of his implementation, each containing what he considered to be a subtle bug. The test harness has two threads that concurrently call functions in the work-stealing queue API. Our model checker based on iterative context-bounding found each of those bugs within a context-switch bound of two.

We plotted the coverage graph for this implementation of the work-stealing queue. Unlike syntactic notions of coverage such as line, branch or path coverage, we have chosen the number of distinct visited states as our notion of coverage. We believe that state coverage is the most appropriate notion of coverage for semantics-based safety checkers such as our model checker. Figure 1 plots the fraction of reachable states covered on the y-axis against the context-switch bound on the x-axis. There are several interesting facts about this coverage graph. First, full state coverage is achieved

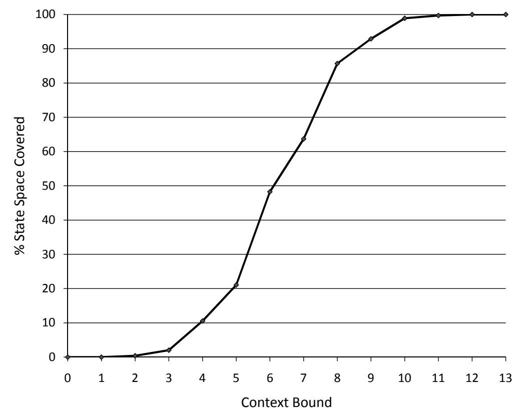

Figure 1. Coverage graph

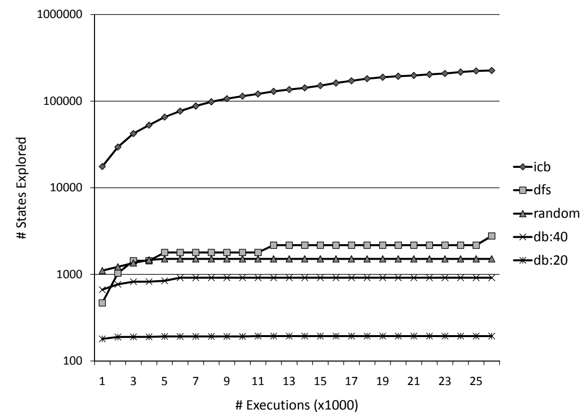

Figure 2. Coverage growth

with eleven preemptions although the program has executions with *at least* 35 preemptions (see Table 1). Second, 90% state coverage is achieved within a context-switch bound of eight. These observations indicate that iterative context-bounding is good at achieving high coverage within bounds that are significantly smaller than the maximum number of possible preemptions.

Finally, we also compared the variation of coverage with time for various methods of state-space search. Figure 2 plots the number of distinct visited states on the y-axis against the number of executions explored by different methods. Note that the y-axis is on a logarithmic scale. There are five curves in the graph corresponding to iterative context-bounding (icb), unbounded depth-first search (dfs), random search (random), depth-first search with depth-bound 40 (db:40), and depth-first search with depth-bound 20 (db:20). As is evident from the graph, iterative context-bounding achieves significantly better coverage at a faster rate compared to the other methods. In Section 4, we present a more detailed discussion of the various graphs presented here.

3. Algorithm

In this section, we describe an algorithm that effectively searches the state space of a program by systematically bounding the number of preemptions. The algorithm takes as input the initial state s_0, and iteratively explores executions with increasing preemptions. In

```
Input: initial state s0 ∈ State
1  struct WorkItem { State state; Tid tid; }
2  Queue⟨WorkItem⟩ workQueue;
3  Queue⟨WorkItem⟩ nextWorkQueue;
4  WorkItem w;
5  int currBound:= 0;
6  for t ∈ enabled(s0) do
7      workQueue.Add(WorkItem (s0, t));
8  end
9  while true do
10     while ¬workQueue.Empty() do
11         w := workQueue.Front();
12         workQueue.Pop();
13         Search(w);
14     end
15     if nextWorkQueue.Empty() then
16         Exit();
17     end
18     currBound:= currBound+ 1;
19     workQueue:= nextWorkQueue;
20     nextWorkQueue.Clear();
21 end

22 Search(WorkItem w) begin
23     WorkItem x;
24     State s;
25     s := w.state.Execute(w.tid);
26     if w.tid ∈ enabled(s) then
27         x := WorkItem(s, w.tid);
28         Search(x);
29         for t ∈ enabled(s) \ {w.tid} do
30             x := WorkItem(s, t);
31             nextWorkQueue.Push(x);
32         end
33     else
34         for t ∈ enabled(s) do
35             x := WorkItem(s, t);
36             Search(x);
37         end
38     end
39 end
```

Algorithm 1: Iterative context bounding algorithm

other words, for any $i \geq 0$, the algorithm explores every execution with i preemptions before exploring any execution with $i + 1$ preemptions. This algorithm can be trivially modified to stop when a particular preemption bound is reached.

We now present a detailed description of the algorithm. The algorithm maintains two queues of work items. Each work item w contains a state and a thread identifier and notifies the model checker to schedule the thread $w.tid$ from the state $w.state$. The variable $workQueue$ contains work items that can be explored within the current preemption bound set in the variable $currBound$. During this exploration, the model checker inserts work items requiring an extra preemption into $nextWorkQueue$, postponing the processing of such work items after the exploration of the states within the current preemption bound.

In lines 6–8, $workQueue$ is initialized with work items corresponding to the initial state. One work item is created for each thread enabled in the initial state. The loop in lines 10–14 removes a work item from the queue, and invokes the procedure Search on it. Whenever control reaches line 15, the algorithm guarantees that all executions with at most $currBound$ preemptions have been executed. In lines 15–20, the algorithm continues the execution of work items in $nextWorkQueue$, if any, after incrementing the $currBound$.

The recursive procedure Search processes a work item w and recursively explores all states reachable without introducing any preemptions. In line 25, the procedure executes the thread $w.tid$ in $w.state$ till the next scheduling point. In order to explore every behavior of the program, it is necessary to insert a scheduling point after each access to a shared variable. Essentially, this forces a thread to execute at most one shared-variable access in every step. Section 3.1 provides an improved strategy for introducing scheduling points.

If $w.tid$ is enabled in the state s (line 26), the algorithm schedules $w.tid$ for another step by calling Search recursively in line 28. At the same time, scheduling some other thread enabled in s results in a preemption of $w.tid$. In lines 29–32, the algorithm creates a work item for every such thread and inserts the item in the $nextWorkQueue$.

If the thread $w.tid$ is not enabled in s, then $w.tid$ voluntarily yielded control in s. Therefore, the algorithm is free to schedule any enabled thread without incurring the cost of a preemption. The loop in lines 34–36 accomplishes this by creating a work item for every enabled thread in s and calling Search on each one of them.

State caching is orthogonal to the idea of context-bounding; our algorithm may be used with or without it. In fact, we have implemented our algorithm in two different model checkers—ZING, which caches states and CHESS, which does not cache states. The description in this section has ignored the issue of state caching. It is easy enough to add that feature by introducing a global variable:

```
Set⟨WorkItem⟩ table;
```

The variable $table$ is initialized to the empty set. We also add the following code at the very beginning of Search to prune the search if a state is revisited.

```
if table.Contains(w) then
    return;
end
table.Add(w);
```

3.1 Strategy for introducing preemptions

During program execution, the scheduler can preempt the current running thread at an arbitrary point. Subtle concurrency errors arise when such preemptions occur exactly when the running thread temporarily violates a global program invariant and subsequent threads require this invariant for correct execution. To find such errors, the algorithm presented above schedules each thread for a single step, enabling a preemption opportunity after every access to a shared variable.

In this section, we show that it is sufficient to insert a scheduling point before a *synchronization* operation in the program, provided the algorithm also checks for data-races [1]. By scheduling all variable access between two synchronization operations atomically, the algorithm significantly reduces the state space explored. In addition, exploring this reduced state space is sound and the algorithm does not miss any errors in the program. This strategy is essentially a kind of partial-order reduction [9, 18] and was first proposed in the form above by Bruening and Chapin [1]. Our contribution here is in showing that this reduction is sound when performing a context-bounded search. The formal soundness proof is

fairly involved and is provided in Appendix A. We only provide a high-level description of the proof in this section.

Let us fix a multithreaded program for the remainder of this section. All the definitions and theorems that follow are with respect to this program.

An *execution* α is a nonempty sequence of steps $\alpha(1), \alpha(2), \ldots$, where $\alpha(i)$ is the identifier of the thread executing the ith step, for $i \geq 1$. We assume that each step in an execution accesses exactly one variable. We denote by $|\alpha|$ the length of α. We assume that thread scheduling is the only source of nondeterminism in the program. Therefore, by executing α from the initial state of the program, we arrive at a unique state. Let $enabled(\alpha)$ denote the set of threads enabled in this state. The execution α is *terminating* if $enabled(\alpha) = \emptyset$. Let $L(\alpha)$ be the thread executing the last step of α, and let $V(\alpha)$ be the variable accessed by $L(\alpha)$ at the last step. Also, for $t \in enabled(\alpha)$, let $NV(\alpha, t)$ be the variable thread t will access if scheduled from the state obtained by executing α. For all $i \in [1, |\alpha|]$, we define $\alpha|_i$ to be the prefix of α whose length is i. Note that a prefix of an execution is also an execution. Given an execution α, the execution $\alpha \cdot \beta$ is obtained by executing steps in β from the state obtained by executing α.

Let $SyncVar$ be the set of synchronization variables that the threads in the program use to communicate with one another. All variables that are not in $SyncVar$ belong to $DataVar$, the set of data variables. Our implementation dynamically infers the variables in $SyncVar$. We also assume that a thread in the program blocks only on accesses to synchronization variables.

Given an execution $\alpha \cdot t$, we say that a preemption occurred at $\alpha \cdot t$ if the last thread in α is enabled in α and is different from t. In this case, the scheduler preempted the execution of $L(\alpha)$ and scheduled the thread t. Let the number of preemptions in α be denoted by $NP(\alpha)$. A preemption at $\alpha \cdot t$ occurs at an access to a synchronization variable if $NV(\alpha, L(\alpha)) \in SyncVar$. In other words, the thread $L(\alpha)$ was preempted right before an access to a synchronization variable. An execution is *observable* if all preemptions occur at accesses to synchronization variables. Note that if a model checker introduces preemptions only at accesses to synchronization variables, then it can explore only observable executions and detect observable races.

Two steps $\alpha(i)$ and $\alpha(j)$ in an execution α are *dependent* if they are either executed by the same thread or if they access the same synchronization variable. Otherwise the two steps are *independent*. Given two dependent steps $\alpha(i)$ and $\alpha(j)$, $\alpha(i)$ *happens before* $\alpha(j)$ if $i < j$. The *happens-before relation* of an execution α, denoted by $HB(\alpha)$, is the transitive closure of the happens-before ordering of all dependent steps in the execution. It is easy to see that $HB(\alpha)$ defines a partial-order on the steps of α.

An execution is *race-free* if any two accesses to the same data variable in α are ordered by $HB(\alpha)$. Two race-free executions α and β are *equivalent* if $HB(\alpha) = HB(\beta)$. Intuitively, two equivalent executions differ only on the order of independent steps and therefore result in the same final state. A pair (α, t) is a *race* if α is race-free but $\alpha \cdot t$ is not. A race (α, t) is observable if α is an observable execution and $L(\alpha) = t$.

THEOREM 2. *A terminating race-free execution α is equivalent to an observable terminating race-free execution β such that $NP(\beta) \leq NP(\alpha)$.*

PROOF. Starting from α, the proof constructs a linearization β of the partial-order $HB(\alpha)$ such that all preemptions in β occur at accesses to synchronization variables. By construction, β is equivalent to α. The key difficulty in the proof is in showing that $NP(\beta) \leq NP(\alpha)$. To do so, the proof constructs β iteratively by changing the order of two independents steps in α in such a way that the number of preemptions does not increase in each iteration. The details of this construction is presented in Appendix A. □

THEOREM 3. *If there is a race (α, t), then there is an observable race (β, u) such that $NP(\beta) \leq NP(\alpha)$.*

PROOF. The proof of this theorem is similar to the proof of Theorem 2 and relies on iteratively constructing β from α without increasing the number of preemptions. The details are presented in Appendix A. □

Theorem 3 allows us to conclude that for any context-bound c, if all observable executions α with $NP(\alpha) \leq c$ are race-free then all executions β (both observable and otherwise) with $NP(\beta) \leq c$ are also race-free. Thus, if no races are reported on observable executions up to the bound c, then indeed the program is race-free up to the bound c. Finally, Theorem 2 allows us to to verify the absence of all errors expressible as predicates on terminating states by evaluating only the observable executions. This is a broad class of errors including both deadlocks and assertion failures.

4. Empirical evaluation

We implemented the iterative-context bounding algorithm in two model checkers and evaluated the algorithm on a few realistic benchmarks. This section describes our evaluation and the results.

We now give brief descriptions of these two model checkers. ZING has been designed for verifying models of concurrent software expressed in the ZING modeling language. The models may be created manually or automatically using other tools. Currently, there exist translators from subsets of C# and X86 assembly code into the ZING modeling language. ZING is an explicit-state model checker; it performs depth-first search with state caching. It maintains the stack compactly using state-delta compression and performs state-space reduction by exploiting heap-symmetry.

CHESS is meant for verifying concurrent programs directly and does not require a model to be created. Similar to the Verisoft [10] model checker, CHESS is stateless and runs program executables directly. However, Verisoft was designed for message-passing software whereas CHESS is designed to verify shared-memory multithreaded software. Since CHESS does not cache states, it expects the input program to have an acyclic state space and terminate under all possible thread schedules. The ZING model checker described earlier has no such restriction and can handle both cyclic and acyclic state spaces. CHESS introduces context switches only at accesses to synchronization variables, while using the Goldilocks algorithm [4] to check for data-races in each execution. As shown in Section 3.1, this methodology is sound while significantly increasing the effectiveness of the state space exploration.

4.1 Benchmarks

We evaluated the iterative context-bounding algorithm on a set of benchmark programs. Each program is an open library, requiring a test driver to close the system. The test driver allocates threads that concurrently call interesting sequences of library functions with appropriate inputs. The input program together with the test driver forms a closed system that is given to the model checker for systematically exploring the behaviors. For the purpose of our experiments, we assume that the only nondeterminism in the input program and the test driver is that induced by the scheduler, which the model checker controls.

Obviously, a model checker can only explore behaviors of the program triggered by the test driver. The quality of the state space search, and thus the bugs found depends heavily upon good test drivers. When available, we used existing concurrent test cases for our experiments. For programs with no existing test cases, we wrote our own drivers that, to our best knowledge, explored interesting

Programs	LOC	Max Num Threads	Max K	Max B	Max c
Bluetooth	400	3	15	2	8
File System Model	84	4	20	8	13
Work Stealing Q.	1266	3	99	2	35
APE	18947	4	247	2	75
Dryad Channels	16036	5	273	4	167

Table 1. Characteristics of the benchmarks. For each benchmark, this table reports the number of lines, the number of threads allocated by the test driver. For an execution, K is the total number of steps, B is the number of blocking instructions, and c is the number of preemptions. The table reports the maximum values of K, B, and c seen during our experiments.

Programs	Total Bugs	Bugs with Context Bound			
		0	1	2	3
Bluetooth	1	0	1	0	0
Work Stealing Queue	3	0	1	2	0
Transaction Manager	3	0	0	2	1
APE	4	2	1	1	0
Dryad Channels	5	1	4	0	0

Table 2. For a total of 14 bugs that our model checker found. this table shows the number of bugs exposed in executions with exactly c preemptions, for c ranging from 0 to 3. The 7 bugs in the first three programs were previously known. Iterative context-bounding algorithm found the 9 previously *unknown* bugs in Dryad and APE.

behavior in the system. Comprehensively closing an open system to expose most of the bugs in the system is a challenging problem, beyond the scope of this paper.

We provide a brief description of the programs used for our evaluation below.

Bluetooth: This program is a sample Bluetooth Plug and Play (PnP) driver modified to run as a library in user space. The sample driver does not contain hardware-specific code but captures the synchronization and logic required for basic PnP functionality. We wrote a test driver with three threads that emulated the scenario of the driver being stopped when worker threads are performing operations on the driver.

File system model: This is a simplified model of a file system derived used in prior work (see Figure 7 in [7]). The program emulates processes creating files and thereby allocating inodes and blocks. Each inode and block is protected by a lock.

Work-stealing queue: This program is an implementation [15] of the work-stealing queue algorithm originally designed for the Cilk multithreaded programming system [8]. The program has a queue of work items implemented using a bounded circular buffer. Our test driver consists of two threads, a victim and a thief, that concurrently access the queue. The victim thread pushes work items to and pops them from the tail of the queue. The thief thread steals work items from the head of the queue. Potential interference between the two threads is controlled by means of sophisticated non-blocking synchronization.

APE: APE is an acronym for Asynchronous Processing Environment. It contains a set of data structures and functions that provide logical structure and debugging support to asynchronous multithreaded code. APE is currently used in the Windows operating system. For our experiments, we compiled APE in user-mode and used a test driver provided by the implementor of APE. In the test, the main thread initializes APE's data structures, creates two

```
// Function called by a worker thread
// of RChannelReaderImpl
void RChannelReaderImpl::AlertApplication(
                                RChannelItem* item)
{
   // Notify Application

   // XXX: Preempt here for the bug
   // Note: this == channel variable in TestChannel
   EnterCriticalSection(&m_baseCS);
   // process before exit
   LeaveCriticalSection(&m_baseCS);
}

// Function called by the main thread
void TestChannel(WorkQueue* workQueue, ...)
{
    // Creating a channel allocates worker threads
    RChannelReader* channel =
        new RChannelReaderImpl(..., workQueue);

    // ... do work here

    channel->Close();
    // wrong assumption that channel->Close() waits
    // for worker threads to be finished

    delete channel;
    // BUG: deleting the channel when
    // worker threads still have a valid reference
    // to the channel
}
```

Figure 3. Use after free bug in Dryad. The bug requires a context switch to happen right before the call to EnterCriticalSection in AlertApplication. This is the only preempting context switch. The bug trace CHESS found involves 6 nonpreempting context switches.

worker threads, and finally waits for them to finish. The worker threads concurrently exercise certain parts of the interface provided by APE.

Dryad channels: Dryad is a distributed execution engine for coarse-grained data-parallel applications [14]. A Dryad application combines computational ”vertices” with communication ”channels” to form a data-flow graph. Dryad runs the application by executing the vertices of this graph on a set of available processors communicating as appropriate through files, TCP pipes, and shared-memory FIFOs. The test harness for Dryad for our experiments was provided by its lead developer. The test has 5 threads and exercises the shared-memory channel library used for communication between the nodes in the data-flow graph.

Transaction manager: This program provides transactions in a system for authoring web services on the Microsoft .NET platform. Internally, the in-flight transactions are stored in a hashtable, access to which is synchronized using fine-grained locking. We used existing test harnesses written by our colleagues for our experiments. Each test contains two threads. One thread performing an operation —create, commit, or delete— on a transaction. The second thread is a timer thread that periodically flushes from the hashtable all pending transactions that have timed out.

Except for the transaction manager, all the benchmarks used above are written in a combination C and C++. Table 1 enumerates the characteristics of these benchmarks. The transaction man-

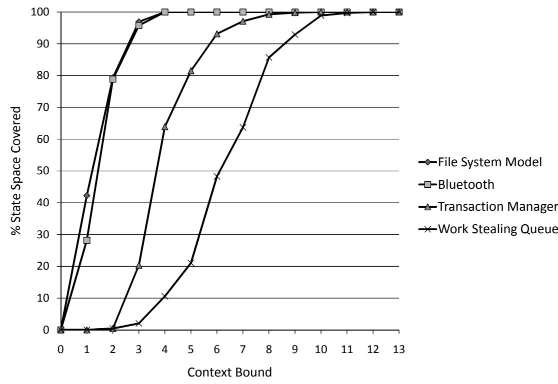

Figure 4. Figures shows the percentage of the entire state space (y axis) covered by executions with bounded number of preemptions (x axis). For state spaces of programs small enough for our model checkers to completely search, the graph shows that more than 90% of the state space is covered with executions with at most 8 preemptions.

ager is a ZING model constructed semi-automatically from the C# implementation, and has roughly 7000 lines of code.

In the rest of the section, we will show that bounding the number of preemptions is an effective method of exploring interesting behaviors of the system, while alleviating the state space explosion problem. Note, as described in Section 2, bounding the number of preemptions results in a state space polynomial in the number of steps in an execution. This allows us to scale systematic exploration techniques to larger programs.

Specifically, we will use our experiments to demonstrate the following two hypotheses:

1. Many subtle bugs manifest themselves in executions with very small preemptions.
2. Most states can be covered with few preemptions

4.2 Small context bounds expose concurrency bugs

Context bounding relies on the intuition that many errors occur due to *few* context switches happening at the *right* places. To substantiate this intuition, we ran the iterative context-bounding program for the five programs shown in Table 2. For the first three programs, namely Bluetooth, work-stealing queue, and the transaction manager, we introduced 7 known bugs that the respective developers considered subtle concurrency errors. The iterative context bounding algorithm was able to find all such errors within a bound of 3.

We also ran the iterative context-bounding algorithm on APE and Dryad, the largest programs currently handled by our model checker. We found a total of 9 previously *unknown* concurrency errors. To provide the reader with an idea of the complexity of these errors, we describe one of the errors we found in Dryad below in detail. This error could not be found by a depth-first search, even after running for a couple of hours.

Dryad use-after-free bug: When deallocating a shared heap object, a concurrent program has to ensure that no existing thread in the system has a live reference to that object. This is a common concurrency problem that is very hard to get right. Figure 3 describes an error that requires only one preempting context switch, but 6 nonpreempting context switches.

The error involves a message channel, which contains a few worker threads that process messages in the channel. When the

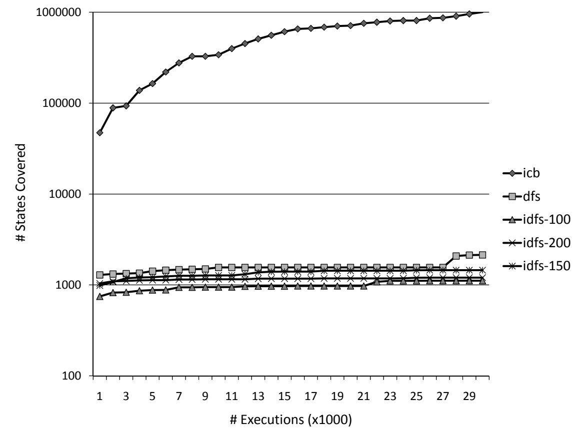

Figure 5. Coverage growth for APE

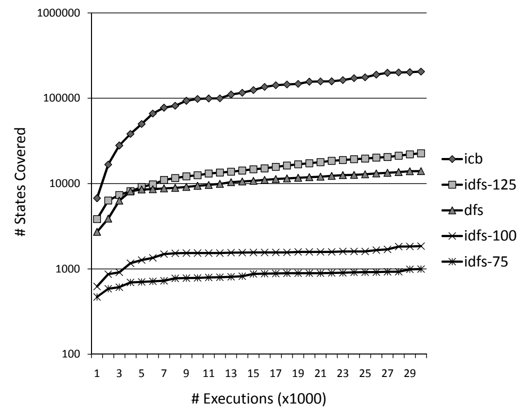

Figure 6. Coverage growth for Dryad

function `TestChannel` calls the `close` function on the channel, each worker thread gets a STOP message, in response to which a worker thread calls the `AlertApplication` function, as part of its cleanup process. However, when there is an preempting context switch right before the thread enters the `m_baseCS` critical section, the main thread is able to return from the `close` function and subsequently delete the channel, which in this case is the current `this` pointer for the worker thread. The use-after-free bug occurs when the worker thread is subsequently scheduled.

When run with a context bound one, the iterative context-switch algorithm systematically tried its budgeted preempting context switch at every step, and eventually found the small window in `AlertApplication` that found the error. In contrast, a depth-first search is flooded with an unbounded number of preemptions, and is thus unable to expose the error within reasonable time limits.

4.3 Few context bounds cover most states

In the previous section, we empirically showed that a small number of preemptions are sufficient to expose concurrency errors. In this section, we show that a fair percentage of state space is reached through executions with few preemptions. Obviously, we are only able to demonstrate this on programs for which our model checkers are able to *complete* the state space search.

Figure 4 shows the cumulative percentage of the entire state space covered by executions with increasing context bounds. The results for transaction manager benchmark is from the ZING model checker, which is an explicit-state model checker. Thus, counting states is straightforward for this program. The remaining three programs are executables run directly by the CHESS model checker. These programs make numerous calls to the synchronization primitives provided by the kernel. Capturing the state in this case would require accounting for this kernel state, apart from the global variables, the heap, and the stack. In fact, this difficulty in capturing states of program executables is the main reason for designing CHESS as a stateless model checker. For these programs, we use the happens-before relation of an execution, described in Section 3.1 and formally defined in Appendix A, as a representation for the state at the end of the execution.

Figure 4 shows that for both Bluetooth and the filesystem model, 4 preemptions are sufficient to completely explore the entire state space. For the relatively larger transaction manager and the work-stealing queue benchmark, a context-bound of 6 and 8 respectively are sufficient to cover more than 90% of the state space. This strongly suggests the advantage of iterative context bounding — when systematically exploring the behavior of multithreaded programs, model checkers can maximize state space coverage by focusing on the polynomial number of executions with few preemptions.

For programs on which the model checker is unable to complete the state space search, we report the increase in the states visited by different search strategies. Figure 5 shows the number of states covered in the y axis with the number of complete executions of the program in the x axis for the APE benchmark. Figure 6 shows corresponding graph for the Dryad benchmark. These two graphs compare the iterative context bounding algorithm with the depth-first (dfs) search strategy and the iterative depth-bounding (idfs) strategy. For the idfs search, we selected different depth bounds and selected the the depth bound with maximum, minimum, and median coverage. From the graph, it is very evident that context bounding is able to systematically achieve better state space coverage, even in the first 1000 executions.

5. Related work

Context-bounding: The notion of context-bounding was introduced by Qadeer and Wu [20] as a method for static analysis of concurrent programs by using static analysis techniques developed for sequential programs. That work was followed by the theoretical result of Qadeer and Rehof [19] which showed that context-bounded reachability analysis for concurrent boolean programs is decidable. Our work exploits the notion of context-bounding for systematic testing in contrast to these earlier results which were focused on static analysis. The combinatorial argument of Section 2 and the distinction between preempting and nonpreempting context switches is a direct result of our focus on dynamic rather than static analysis.

State-space reduction techniques: Researchers have explored the use of partial-order reduction [9, 18, 17, 3] and symmetry reduction [13, 5, 12] to combat the state-space explosion problem. These optimizations are orthogonal and complementary to the idea of context-bounding. In fact, our preliminary experiments indicate that state-space coverage increases at an even faster rate when partial-order reduction is performed during iterative context-bounding.

Analysis tools: Researchers have developed many dynamic analyses, such as data-race detection [23] and atomicity-violation detection [6], for finding errors in multithreaded software. Such analyses are again orthogonal and complementary to context-bounding. They are essentially program monitors which can be applied to each execution explored by iterative context-bounding.

Heuristic search: Confronted with limited computational resources and large state spaces, researchers have developed heuristics for partial state-space search. Groce and Visser [11] proposed the heuristic of prioritizing states with more enabled threads. Sivaraj and Gopalakrishnan [24] proposed the use of a random walk through the search space. Unlike these heuristics, iterative context-bounding provides an intuitive notion of coverage and a polynomial guarantee on the number of context-bounded executions.

6. Conclusions

Model checking or systematic exploration of program behavior is a promising alternative to traditional testing methods for multithreaded software. However, it is difficult to perform systematic search on large programs because the number of possible program executions grows exponentially with the length of the execution. Confronted with this state-explosion problem, traditional model checkers perform partial state-space search using techniques such as iterative depth-bounding. Although effective for message-passing software, iterative depth-bounding is inadequate for multithreaded software because several orders of magnitude more steps are required to get interesting behavior in a multithreaded program than in a message-passing program.

This paper proposes a novel algorithm called *iterative context-bounding* for effectively searching the state space of a multithreaded program. Unlike iterative depth-bounding which gives priority to executions with shorter length, iterative context-bounding gives priority to executions with fewer preemptions. We show that that by bounding the number of preemptions, the number of executions becomes a polynomial function of the execution depth. Therefore, context-bounding allows systematic exploration to scale to large programs without sacrificing the ability to go deep in the state space.

We implemented iterative context-bounding in two model checkers and used our implementation to uncover 9 previously unknown bugs in realistic multithreaded benchmarks. Each of these bugs required at most 2 preemptions. Our experience with these benchmarks and other benchmarks with previously known bugs indicates that many bugs in multithreaded code are manifested in executions with a few preemptions. Our experiments also indicate that state coverage increases faster with iterative context-bounding than with other search methods. Therefore, we believe that iterative context-bounding significantly improves upon existing search strategies.

In future work, we would like to make our model checker even more scalable. We believe that incorporating complementary state-reduction techniques, such as partial-order reduction, could improve scalability. Yet another interesting direction for our work is to extend CHESS, which currently handles user-mode programs written against the WIN32 API, to kernel-mode programs.

Acknowledgements

We would like to thank Michael Isard, Joseph Joy, and Daan Leijen for providing the benchmarks, and Iulian Neamtiu for helping with CHESS. We would like to thank Tom Ball and the anonymous reviewers for their feedback on a prior version of this paper.

References

[1] Derek Bruening and John Chapin. Systematic testing of multithreaded Java programs. Technical Report LCS-TM-607, MIT/LCS, 2000.

[2] 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.

[3] Matthew B. Dwyer, John Hatcliff, Robby, and Venkatesh Prasad Ranganath. Exploiting object excape and locking information in partial-order reductions for concurrent object-oriented programs. *Formal Methods in System Design*, 25:199–240, 2004.

[4] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: Efficiently computing the happens-before relation using locksets. In *FATES/RV 06: Formal Approaches to Testing and Runtime Verification*, volume 4262 of *Lecture Notes in Computer Science*, pages 193–208. Springer-Verlag, 2006.

[5] F. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1/2):105–131, August 1996.

[6] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *POPL 04: Principles of Programming Languages*, pages 256–267. ACM Press, 2004.

[7] 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.

[8] 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. ACM Press, 1998.

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

[10] Patrice Godefroid. Model checking for programming languages using Verisoft. In *POPL 97: Principles of Programming Languages*, pages 174–186. ACM Press, 1997.

[11] Alex Groce and Willem Visser. Model checking Java programs using structural heuristics. In *ISSTA 02: Software Testing and Analysis*, pages 12–21, 2002.

[12] Radu Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *ASE 01: Automated Software Engineering*, pages 254–261, 2001.

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

[14] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. Technical Report MSR-TR-2006-140, Microsoft Research, 2006.

[15] Daan Leijen. Futures: a concurrency library for C#. Technical Report MSR-TR-2006-162, Microsoft Research, 2006.

[16] Madanlal Musuvathi, David Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI 02: Operating Systems Design and Implementation*, pages 75–88, 2002.

[17] 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.

[18] Doron Peled. Partial order reduction: Model-checking using representatives. In *MFCS 96: Mathematical Foundations of Computer Science*, pages 93–112. Springer-Verlag, 1996.

[19] 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-Verlag, 2005.

[20] S. Qadeer and D. Wu. KISS: Keep it simple and sequential. In *PLDI 04: Programming Language Design and Implementation*, pages 14–24. ACM Press, 2004.

[21] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Fifth International Symposium on Programming*, Lecture Notes in Computer Science 137, pages 337–351. Springer-Verlag, 1981.

[22] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice Hall, 2002.

[23] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[24] Hemanthkumar Sivaraj and Ganesh Gopalakrishnan. Random walk based heuristic algorithms for distributed memory model checking. *Electronic Notes in Theoretical Computer Science*, 89(1), 2003.

A. Appendix

A.1 Definitions

The number of preemptions in α, denoted by $NP(\alpha)$, is defined recursively as follows:

$$NP(t)=0$$
$$NP(\alpha.t)=\begin{cases} NP(\alpha) & \text{if } t = L(\alpha) \vee L(\alpha) \notin \mathit{enabled}(\alpha) \\ NP(\alpha)+1 & \text{otherwise} \end{cases}$$

The happens-before relation of an execution α, denoted by $HB(\alpha)$, is the transitive closure of the relation

$$\{ (i,j) \in \{1,\ldots,|\alpha|\} \times \{1,\ldots,|\alpha|\} \mid (i < j \wedge t_i = t_j) \vee (i < j \wedge V(\alpha|_i) = V(\alpha|_j) \wedge V(\alpha|_j) \in \mathit{SyncVar}) \}.$$

The execution α is race-free if and only if for all $i, j \in [1, |\alpha|]$, if $i < j$ and $V(\alpha|_i) = V(\alpha|_j)$ then $(i,j) \in HB(\alpha)$.

We assume that there is a special synchronization event variable e_t corresponding to each thread t. The first operation of t blocks on e_t until e_t is signaled by the parent of t in the operation that creates t. Thus, it is guaranteed that in any execution the first operation of any thread accesses a synchronization variable. Furthermore, we also assume that a thread terminates by performing as its final operation a block on e_t that is never signaled. Thus, it is also guaranteed that in the final state of any terminating execution all threads are accessing a synchronization variable.

Consider an execution α, a thread t, and a data variable d. We define $\mathit{last}(\alpha,t)$ to be 0 if $\alpha(i) \neq t$ for all $i \in [1,|\alpha|]$. Otherwise, we define $\mathit{last}(\alpha,t)$ to be the number $i \in [1,|\alpha|]$ such that (1) $\alpha(i) = t$, (2) $V(\alpha|_i) \in \mathit{SyncVar}$, and (3) for all $j \in (i,|\alpha|]$, if $\alpha(j) = t$ then $V(\alpha|_j) \in \mathit{DataVar}$. We define $\mathit{last}(\alpha,d)$ to be 0 if $V(\alpha|_i) \neq d$ for all $i \in [1,|\alpha|]$. Otherwise, we define $\mathit{last}(\alpha,d)$ to be the number $i \in [1,|\alpha|]$ such that $V(\alpha|_i) = d$ and $V(\alpha|_j) \neq d$ for all $j \in (i,|\alpha|]$.

A pair (α,t) is a *race* if α is race-free, $NV(\alpha,t) \in \mathit{DataVar}$, and there exists $i \in [1,|\alpha|]$ such that $i = \mathit{last}(\alpha, NV(\alpha,t))$, $\alpha(i) \neq t$, and $(i, \mathit{last}(\alpha,t)) \notin HB(\alpha)$.

An execution α is k-nice if the following conditions hold:

1. For all $i \in [1,|\alpha|)$, either $\alpha(i) = \alpha(i+1)$ or $V(\alpha|_{i+1}) \in \mathit{SyncVar}$.
2. There are exactly k threads in the set

$$\{t \mid t \in \mathit{enabled}(\alpha) \wedge t \neq L(\alpha) \wedge NV(\alpha,t) \in \mathit{DataVar}\}.$$

An observable execution is 0-nice. An execution is *nice* if it is k-nice for some $k \geq 0$. A race (α,t) is k-nice if α is k-nice and $\alpha(|\alpha|) = t$. A race is *nice* if it is k-nice for some $k \geq 0$.

A.2 Proofs

LEMMA 1. *If α is a nice execution, $\alpha \cdot t$ is a race-free execution, and $\alpha \cdot t \cdot \delta$ is an execution, then there is a nice and race-free execution β equivalent to $\alpha \cdot t$ such that $NP(\beta \cdot \delta) \leq NP(\alpha \cdot t \cdot \delta)$.*

PROOF. If $V(\alpha \cdot t) \in SyncVar$ or $\alpha(|\alpha|) = t$, then $\beta = \alpha \cdot t$ is nice and we are done. Otherwise $V(\alpha \cdot t) \in DataVar$ and $\alpha(|\alpha|) \neq t$. Then, there exists $i \in [1, |\alpha|)$ such that $\alpha(i) = t$. Let $l \in [1, |\alpha|)$ be such that $\alpha(l) = t$ and $\alpha(j) \neq t$ for all $j \in (l, |\alpha|]$. Let γ be the nonempty sequence such that $\alpha = \alpha|_l \cdot \gamma$. Since $\alpha \cdot t$ is race-free and $V(\alpha \cdot t) \in DataVar$, we know that $\alpha|_l \cdot t \cdot \gamma$ is an execution equivalent to $\alpha \cdot t$. We let $\beta = \alpha|_l \cdot t \cdot \gamma$. We know that both $\gamma(1) \neq t$ and $\gamma(|\gamma|) \neq t$. If $t \in enabled(\alpha|_l \cdot t)$, we have $NP(\beta) = NP(\alpha|_l \cdot \gamma)$. If $t \notin enabled(\alpha|_l \cdot t)$, we have $NP(\beta) < NP(\alpha|_l \cdot \gamma)$. In either case, we have $NP(\beta) \leq NP(\alpha|_l \cdot \gamma) = NP(\alpha)$. In addition, since $\alpha(|\alpha|) \neq t$ and $t \in enabled(\alpha)$, we have $NP(\alpha) < NP(\alpha \cdot t)$. Therefore $NP(\beta) < NP(\alpha \cdot t)$. If δ is empty, we are done. Otherwise, let $\delta = u \cdot \delta'$ for some δ'. We have $NP(\beta) \leq NP(\beta \cdot u) \leq NP(\beta) + 1$ and $NP(\alpha \cdot t) \leq NP(\alpha \cdot t \cdot u) \leq NP(\alpha \cdot t) + 1$. Therefore $NP(\beta \cdot u) \leq NP(\alpha \cdot t \cdot u)$. Since $\beta \cdot u$ is equivalent to $\alpha \cdot t \cdot u$, we conclude that $NP(\beta \cdot u \cdot \delta') \leq NP(\alpha \cdot t \cdot u \cdot \delta')$. □

LEMMA 2. *A race-free execution α is equivalent to a nice race-free execution β such that $NP(\beta) \leq NP(\alpha)$.*

PROOF. Let $|\alpha| = n$. We construct a sequence of executions $\beta_1, \ldots, \beta_n$ by repeated applications of Lemma 1. We let $\beta_1 = \alpha(1)$. For each $i \in [1, n)$, we obtain β_{i+1} is obtained by invoking Lemma 1 with $\alpha = \beta_i$, $t = \alpha(i+1)$, and $\delta = \alpha(i+2) \cdots \alpha(n)$. We let $\beta = \beta_n$. □

THEOREM 2. **(Restatement)** *A terminating race-free execution α is equivalent to an observable terminating race-free execution β such that $NP(\beta) \leq NP(\alpha)$.*

PROOF. By Lemma 2, we know that α is equivalent to a nice race-free execution β such that $NP(\beta) \leq NP(\alpha)$. Since α is terminating, so is β. If β is k-nice for $k > 0$, then there exists $i \in [1, |\beta|)$ such that $\beta(i) \neq \beta(i+1)$ and $NV(\beta|_i, \beta(i)) \in DataVar$. Since β is nice, we know that thread $\beta(i)$ is never scheduled after step i. Therefore $NV(\beta, \beta(i)) \in DataVar$ and $\beta(i) \in enabled(\beta)$ which is a contradiction. Therefore β is 0-nice. □

LEMMA 3. *If there is a race (α, t) such that α is nice, then there is a nice race (β, u) such that $NP(\beta) \leq NP(\alpha)$.*

PROOF. Let $NV(\alpha, t) = d$. Since $d \in DataVar$ and the first action of any thread accesses a synchronization variable, there exists $i \in [1, |\alpha|]$ such that $\alpha(i) = t$. Let $l \in [1, |\alpha|]$ be such that $\alpha(l) = t$ and for all $j \in (l, |\alpha|]$, we have $\alpha(j) \neq t$. Since (α, t) is a race, we know that $last(\alpha, d) \in [1, \alpha]$, $\alpha(last(\alpha, d)) \neq t$ and therefore $l \neq last(\alpha, d)$. There are two cases:

($l < last(\alpha, d)$): Since α is nice and $\alpha(l) = t \neq \alpha(l+1)$, we have $V(\alpha|_{l+1}) \in SyncVar$. Since $V(\alpha|_{last(\alpha,d)}) \in DataVar$, the interval $(l, last(\alpha, d))$ is nonempty. Suppose $n \in (l, last(\alpha, d))$ is such that $V(\alpha|_{n+1}) = d$ and for all $i \in (l, n]$, we have $V(\alpha|_i) \neq d$. Let $\beta = \alpha|_l \cdot t \cdot \alpha(l+1) \cdots \alpha(n)$ and $u = \alpha(n+1)$. Then β is nice and $NP(\beta) \leq NP(\alpha|_n) \leq NP(\alpha)$. We have $NV(\alpha|_l, t) = NV(\alpha, t) = d$. Therefore $last(\beta, d) = l + 1$. Since $NV(\alpha|_n, u) = d$, we have $NV(\beta, u) = d$. Since $V(\beta|_{l+1}) \in DataVar$, $\beta(l+1) = t$, and $\alpha(i) \neq t$ for all $i \in [l+1, n]$, we have $(l+1, last(\beta, u)) \notin HB(\beta)$. Since α is nice and $V(\alpha|_{n+1}) \in DataVar$, we have $\alpha(n) = \alpha(n+1) = u$ and consequently $\beta(|\beta|) = u$. Therefore (β, u) is a nice race.

($l > last(\alpha, d)$): Let $\beta = \alpha|_l$. Since β is a prefix of α, we know that β is nice, $\beta(|\beta|) = t$, and $NP(\beta) \leq NP(\alpha)$. Since (α, t) is a race, we know that $(last(\alpha, d), last(\alpha, t)) \notin HB(\alpha)$. Since $HB(\beta) \subseteq HB(\alpha)$, we have $(last(\alpha, d), last(\alpha, t)) \notin HB(\beta)$. We have $last(\alpha, d) = last(\beta, d)$ and $last(\alpha, t) = last(\beta, t)$. Therefore $(last(\beta, d), last(\beta, t)) \notin HB(\beta)$. Finally $NV(\beta, t) = NV(\alpha, t) = d$, and we get that (β, t) is a nice race. □

We define a partial order $\prec$ on nice executions as follows. Let α be a nice execution and γ be the unique longest 0-nice prefix of α. Let $\alpha = \gamma \cdot \delta$. Similarly, Let α' be a nice execution, γ' be the unique longest 0-nice prefix of α', and $\alpha' = \gamma' \cdot \delta'$. Now, $\alpha \prec \alpha'$ iff $|\gamma| > |\gamma'| \ \vee \ |\gamma| = |\gamma'| \wedge |\delta| < |\delta'|$. For terminating programs, the relation $\prec$ is *well-founded*, that is, for every execution α there is a finite sequence $\alpha_n \prec \cdots \prec \alpha_1$ such that $\alpha_1 = \alpha$ and α_n is 0-nice.

LEMMA 4. *If there is a k-nice race (α, t) for some $k > 0$, then there is a nice race (β, u) such that $\beta \prec \alpha$ and $NP(\beta) \leq NP(\alpha)$.*

PROOF. Let γ be the unique longest 0-nice prefix of α. Since $k > 0$, we know that γ is a strict prefix of α. Let $l = |\gamma|$ and $x = \alpha(l)$. If (γ, x) is race, then we let $\beta = \gamma$ and $u = x$. Since $x = \alpha(l) = \gamma(l)$, (γ, x) is a nice race. Since β is 0-nice, we have $\beta \prec \alpha$. Since β is a prefix of α, we have $NP(\beta) \leq NP(\alpha)$. Otherwise, we have that (γ, x) is not a race. Let $d = NV(\gamma, x)$. Then $d \in DataVar$, $x \neq \alpha(i)$ for all $i \in (l, |\alpha|]$, and $x \neq t$. There are two cases:

($\exists j \in (l, |\alpha|]. V(\alpha|_j) = d$): Let $n \in [l, |\alpha|)$ be such that $V(\alpha|_{n+1}) = d$ and $V(\alpha|_j) \neq d$ for all $j \in (l, n]$. Let $\beta = \gamma \cdot x \cdot \alpha(l+1) \cdots \alpha(n)$ and $u = \alpha(n+1)$. Then β is nice and $NP(\beta) \leq NP(\alpha|_n) \leq NP(\alpha)$. Moreover, $\gamma \cdot x$ is a prefix of the unique longest 0-nice prefix of β. Therefore $\beta \prec \alpha$. We have $NV(\alpha|_l, x) = NV(\alpha, x) = d$. Therefore $last(\beta, d) = l + 1$. Since $NV(\alpha|_n, u) = d$, we have $NV(\beta, u) = d$. Since $V(\beta|_{l+1}) \in DataVar$, $\beta(l+1) = x$, and $\alpha(i) \neq x$ for all $i \in [l+1, n]$, we have $(l+1, last(\beta, u)) \notin HB(\beta)$. Since α is nice and $V(\alpha|_{n+1}) \in DataVar$, we have $\alpha(n) = \alpha(n+1) = u$ and consequently $\beta(|\beta|) = u$. Therefore (β, u) is a nice race.

($\forall j \in (l, |\alpha|]. V(\alpha|_j) \neq d$): Let $\beta = \gamma \cdot x \cdot \alpha(l+1) \cdots \alpha(|\alpha|)$ and $u = t$. Then β is nice and $NP(\beta) \leq NP(\alpha)$. Moreover, $\gamma \cdot x$ is a prefix of the unique longest 0-nice prefix of β. Therefore $\beta \prec \alpha$. Let $e = NV(\alpha, t) = NV(\beta, t)$. There are two cases: $d = e$ or $d \neq e$. Suppose $d = e$. Since $V(\beta|_{l+1}) \in DataVar$, $\beta(l+1) = x$, $\alpha(i) \neq x$ for all $i \in [l+1, |\alpha|]$, and $x \neq u$, we have $(l+1, last(\beta, u)) \notin HB(\beta)$. Suppose $d \neq e$. Since (α, t) is a race, we know that $(last(\alpha, e), last(\alpha, t)) \notin HB(\alpha)$. Since β contains a single additional event over α and this event is an access of variable $d \neq e$ by thread $x \neq u$, we get that $(last(\beta, e), last(\beta, t)) \notin HB(\beta)$. Therefore (β, u) is a race. Since γ is a strict prefix of α, we know that $l < |\alpha|$. Therefore $\beta(|\beta|) = u$ and we get that (β, u) is a nice race. □

THEOREM 3. **(Restatement)** *If there is a race (α, t), then there is an observable race (β, u) such that $NP(\beta) \leq NP(\alpha)$.*

PROOF. Suppose (α, t) is a race. Since α is race-free, by Lemma 2 we get a nice race-free execution α' such that $NP(\alpha') \leq NP(\alpha)$ and (α', t) is a race. From the well-foundedness of $\prec$ and repeated applications of Lemma 4, we obtain a finite sequence of nice races $(\beta_1, t_1), \ldots, (\beta_n, t_n)$ such that $(\beta_1, t_1) = (\alpha', t)$, $\beta_n \prec \cdots \prec \beta_1$, $NP(\beta_n) \leq \cdots \leq NP(\beta_1)$, and β_n is 0-nice. We let $\beta = \beta_n$ and $u = t_n$. □