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 explosively 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 two kinds of context switches, preempting and nonpreempting, and show that bounding the number of preempting context switches to a small number significantly alleviates the state space explosion, without limiting the depth of the execution. 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. The iterative context-bounding algorithm uncovered 7 previously unknown bugs in our benchmarks. Each of these bugs was exposed by an execution with at most two context switches. Our initial experience with the technique is very encouraging and demonstrates that iterative context-bounding is a significant improvement on existing techniques for testing multithreaded programs.

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, 22] 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* [23], 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, 17]. 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 the execution of a multithreaded program, a *context switch* occurs when a thread temporarily stops execution and a different thread starts. Some context switches occur when the currently running thread either terminates or blocks temporarily on

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.

Copyright © ACM [to be supplied]. . . $5.00.

a resource. We call these *nonpreempting* context switches. On the other hand, *preempting* context switches occur when the scheduler suspends the running thread at an arbitrary point. Intuitively, the iterative context-bounding algorithm prioritizes executions with fewer preempting context switches during state space search. For a given context-bound c, the algorithm executes only those executions in which the number of preempting context switches is at most c. Unlike depth-bounding, a thread in the program can execute an arbitrary number of steps between context switches. This allows the model checker to go deeper in the state space with a small number of context switches. Moreover, even with a context-bound of 0, a model checker can execute a terminating and deadlock-free program to completion. In the rest of the paper, unless otherwise qualified, we will refer to prempting context switches simply without the qualifier.

Limiting the number of context switches has many powerful and desirable consequences for systematic state-space exploration of multithreaded programs. First, we show (Section 2) that for a fixed number of context switches, 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.

Second, we provide (Section 4) empirical evidence that a small number of context switches are sufficient to expose intricate nontrivial concurrency bugs. Also, for a set of programs for which complete search is possible, we show that few context switches 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 context-switch bound is a productive search strategy.

Third, iterative context-bounding has the important property that it finds a trace with the smallest number of context-switches 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 context-switches, the algorithm guarantees that any error in the program requires at least $c+1$ context switches. 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 executables directly, much along the lines of Verisoft [10], but designed for shared-memory multithreaded programs. Our evaluation, based on these two implementations, is described in Section 4. Our implementation uncovered 7 previously unknown bugs in several real-world multithreaded programs. Each of these bugs was exposed by an execution with at most 2 context switches.

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 contexts is superior to bounding the depth as a strategy for systmatic exploration of multithreaded executions.
- A combinatorial arugment that for a fixed number of contexts, the number of executions is polynomial in the total number of steps excuted by the program.
- An iterative context-bounding algorithm that systematically enumerates program executions in increasing order of context switches.
- Empirical evidence that context-bounded executions expose interesting behavior of the program, even when the number of contexts 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. Since the scheduler is allowed to choose the next thread at each step, the number of possibilities explodes exponentially with the number of steps. To make this point concretely, suppose P is a *terminating* multithreaded program, i.e., there is a number i such that the length of every execution of P is bounded by i. 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*. In program executions, there are two kinds of context switches—preempting and nonprempting. A *preempting context switch* occurs when the scheduler preempts the execution of the currently running thread, say at the expiration of a time slice, and schedules another thread. On the other hand, a *nonpreempting context switch* occurs when a thread voluntarily yields execution, either when the thread terminates or blocks on an unavialable resource. In context-bounding, we bound the number of preempting context switches but leave the number of nonpreempting context switches unconstrained. We show below that the number of executions of P with at most c preempting context switches 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 and also because in our experience many 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.

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 now present a theoretical bound on the number of context-bounded executions of a multithreaded program. Let ${}^{x}\mathcal{C}_{y}$ denote the number of ways of choosing y objects out of x.

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 preempting context switches.*

PROOF. The length of each execution of P is bounded by nk. Therefore, there are are at most nk points where a preempting

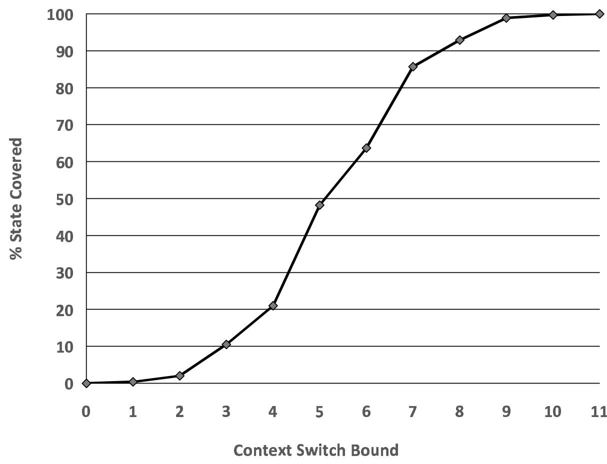


Figure 1. Coverage graph

context switch can occur and at most ${}^{nk}\mathcal{C}_c$ ways of selecting c context switches from these nk points. Once the c context switches 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 preempting context switches. □

Assuming that c is much smaller than both k and nb, the bound given in the theorem above is simplified to $(nk)^c(nb)^c(nb)! = (n^2kb)^c(nb)!$. 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 $(n^2k)^cn!$.

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 with eleven context switches although the program has executions with *at least* 35 preempting context switches (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 context switches.

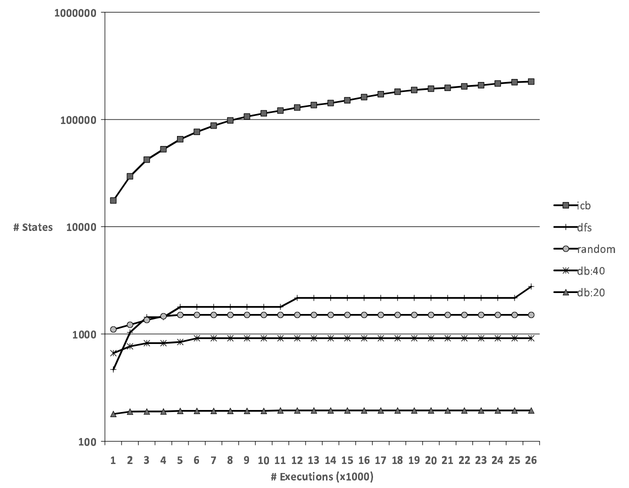


Figure 2. Coverage growth

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 achives 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 systematically searches the state space of a program by iteratively increasing the number of preempting context switches. The algorithm takes as input s_0, the initial state of the program and csb, the context-switch bound. The algorithm works in phases. In phase 0, the algorithm explores using a depth-first search all states that are reachable from s_0 via executions with at most csb preempting context switches. All states that are reachable with exactly csb preempting context switches are added to a work list called $workQueue$. In phase 1, these states are removed from $workQueue$ one by one. For each state, another depth-first search with a context-switch bound of csb is initiated. Just as before, new states generated from these states via executions with exactly csb context switches are pushed to the back of $workQueue$. The algorithm continues in this fashion until $workQueue$ is empty. Our algorithm ensures that at the end of phase i, it has explored all executions with $(i + 1) * \text{csb} + i$ preempting context switches. If $\text{csb} = 0$, the algorithm performs strict iterative context-bounding—for any $i \geq 0$, every execution with i preempting context switches is explored before any execution with $i + 1$ context switches.

We now present a more detailed description of the algorithm. The algorithm assumes that the set of thread identifiers is given by Tid. The variable $workQueue$ is a queue of work items whose type is given in line 1. Each work item w contains a state $w.state$, a thread identifier $w.tid$, and a phase identifier $w.phase$. The work represented by this work item is a depth-first search with context-switch bound csb to be performed from state $w.state$ with the

```
Input: initial state s0 ∈ State and context switch bound csb
1  struct WorkItem { State state; Tid tid; int phase; }
2  Queue⟨WorkItem⟩ workQueue;
3  WorkItem w;
4  int currPhase;
5  for t ∈ Tid do
6      w.state := s0;
7      w.tid := t;
8      w.phase := 0;
9      workQueue.Add(w);
10 end
11 currPhase := 0;
12 while ¬workQueue.Empty() do
13     w := workQueue.Front();
14     workQueue.Pop();
15     if currPhase < w.phase then
           /* explored (currPhase + 1) * csb + currPhase
              preempting context switches          */
16         currPhase := w.phase;
17     end
18     Search(w, 0);
19 end

20 Search(WorkItem w, int ncs) begin
21     if ¬w.state.Enabled(w.tid) then
22         return;
23     end
24     WorkItem x;
25     x.state := w.state.Execute(w.tid);
26     x.tid := w.tid;
27     x.phase := w.phase;
28     Search(x, ncs);
29     for t ∈ Tid \ {w.tid} do
30         x.tid := t;
31         if ¬x.state.Enabled(w.tid) then
32             x.phase := w.phase;
33             Search(x, ncs);
34         else if ncs = csb then
35             x.phase := w.phase + 1;
36             workQueue.Push(x);
37         else
38             x.phase := w.phase;
39             Search(x, ncs+1);
40         end
41     end
42 end
```

Algorithm 1: Iterative context bounding

proviso that only thread $w.tid$ is executed from $w.state$. The field $w.phase$ represents the phase of the algorithm in which the work item will be processed.

In lines 5–10, $workQueue$ is initialized with work items corresponding to the initial state. One work item is created for each thread in Tid. Line 11 initializes the current phase of the algorithm stored in the variable $currPhase$ to 0. The phase is updated in line 16 whenever the phase of the work item extracted from the work queue is greater than the current value of the phase. The loop in lines 12–19 removes a work item from the queue, updates $currPhase$ if required, and invokes the procedure Search on it. Whenever control reaches line 16, the algorithm guarantees that all executions with $(currPhase + 1) * \mathsf{csb} + currPhase$ preempting context switches have been executed.

The recursive procedure Search takes two arguments—a work item w and an integer ncs. The integer ncs represents the number of preempting context switches that have occurred in the current depth-first search. The search is pruned when $ncs = \mathsf{csb}$ and an preempting context switch is about to occur. The invocation of Search on line 18 has the value 0 for this parameter because a fresh search is being initiated.

The implementation of the procedure Search is as follows. If the thread $w.tid$ is enabled in $w.state$ we execute that thread (line 25) one step to create a new state. In lines 25–28, we create a new work item containing the new state and the same thread $w.tid$ that was executed to reach it. Then Search is called recursively with the new work item but with the same value of ncs that was passed in since no additional context switch has occurred. In this way, our algorithm always gives preferences to schedules with fewer context switches. After this search terminates, the pseudocode in lines 29–42 schedules all threads other than $w.tid$ from the new state obtained in line 25. Scheduling a different thread results in a context switch; the test on line 31 determines if the context switch is nonpreempting. If $w.tid$ is not enabled in the new state, then the context switch is nonpreempting. Therefore, Search is invoked with the same value of ncs that was passed in. Otherwise, the context switch is preempting. The test on line 34 determines whether this preempting context switch is allowed given the bound of csb. If the test succeeds, then the context switch is not allowed, the search is pruned, and a work item is added to the back of the queue. Otherwise, Search is called recursively. The algorithm is guaranteed to terminate if for all possible thread schedules, the input program reaches a state in which every thread is either blocked or terminated.

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⟨State⟩ 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.state) then
    return;
end
table.Add(w.state);
```

4. Empirical Evaluation

We have created two implementations of iterative context-bounding in the ZING and CHESS model checkers. 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 acylic 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.

For each program execution, each model checker verifies language-specific and programmer-supplied assertions. In addition, the absence of data-races along each execution is also verified using an implementation of the Goldilocks algorithm [4]. Our verification methodology partitions the set of program variables into data and synchronization variables. Synchronization variables, such as locks, events, and semaphores, are used to ensure that there are no data-races on the data variables. CHESS introduces context switches only at accesses to synchronization variables. This is sound as long as there are no data-races on the data variables [1], a specification that is verified by the model checker.

4.1 Benchmarks Used

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 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

Programs	LOC	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 preempting context switches. The table reports the maximum values of K,B, and c seen during our experiments.

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 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.

4.2 Benchmark Characteristics

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 manager is a ZING model constructed semi-autmatically 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 preempting context switches is an effective method of exploring

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	3	1	2	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 preempting context switches, for c ranging from 0 to 3. The 7 bugs in the first three programs was previously known. Iterative context-bounding algorithm found the 7 previously *unknown* bugs in Dryad and APE.

interesting behaviors of the system, while alleviating the state space explosion problem. Note, as described in Section 2, bounding the number of preempting context switches 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 preempting context switches.
2. Most states can be covered with few preempting context switches

4.3 Small context bounds expose subtle 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 concurrency 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 the APE and Dryad programs. These programs are the largest examples our model checkers is able to currently handle. We found a total of 7 previously *unkonwn* 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 4 describes an error that requires only one preempting context switch, but 6 nonpreempting context switches. The iterative context-bounding algorithm finds the error when the bound is set to one. Note, the algorithm does not bound the number of nonpreempting context switches.

The error involves a message channel, which contains a few worker threads that process messages in the channel. When the 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. Interestingly, this bug scenario is prevented if the context switch happens right before the call to `AlertApplication` or after the worker thread enters the critical section.

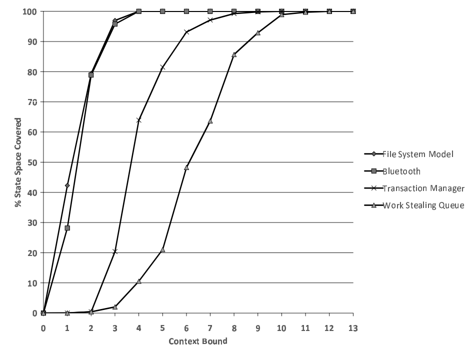


Figure 3. Figures shows the percentage of the entire state space (y axis) covered by executions with bounded number of preempting context switches (x axis). For state spaces of programs small enough for our model checkers to completely search, the graph shows that more than 90% of the state space is covered with exeuctions with at most 8 context switches.

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 preempting context switches, and is thus unable to expose the error within reasonable time limits.

4.4 Few context bounds cover most states

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

Figure 3 shows the cumulative percentage of the entire states 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 actual executables run directly by the CHESS model checker. CHESS is a stateless model checker, and it is fairly complicated to capture the state of these executables, which make extensive call to the synchrnoization primitives provided by the kernel. Thus, capturing states would require accounting for this kernel state, apart from the executable state in the global variables, heap, and the stack. [1] Instead of capturing the states, we use the Mazurkiewicz trace [16] as a representation for the state. A Mazurkiewicz trace captures the happens-before relation between accesses to the synchronization variables in an execution trace. Two executions that produce the same Mazurkiewicz trace are guaranteed to result in the same state.

Figure 3 shows that for both Bluetooth and the filesystem model, 4 preempting context switches are sufficient to completely

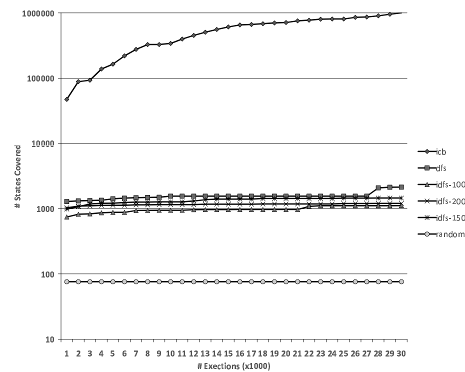
[1] This difficulty in capturing the states is the key reason for designing CHESS as a stateless model checker.

```
// Function called by worker thread
void RChannelReaderImpl::AlertApplication(
                                RChannelItem* item)
{
    RChannelInterruptHandler* interruptHandler = NULL;
    {
        // need a context switch here for the bug
        EnterCriticalSection(&m_baseCS);

        if (m_interruptHandler != NULL)
        {
            // code removed here
            // process interrupts
        }
    }
}

// Function called by the main thread
void TestChannel(WorkQueue* workQueue, ...)
{
    // RChannelSerializedReader is a subclass
    //  of RChannelReaderImpl
    RChannelReader* channel =
        new RChannelSerializedReader(..., workQueue);

    // ... do work here

    channel->Close();
    // wrong assumption that channel->Close() calls
    // workQueue->Stop(), which waits for all
    // worker threads to be done

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

Figure 4. 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.

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 preempting context switches.

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. For comparison, we also performed a random state space search, and we report the coverage only for APE. Random search on Dryad did very poorly. From the graph, it is very evident



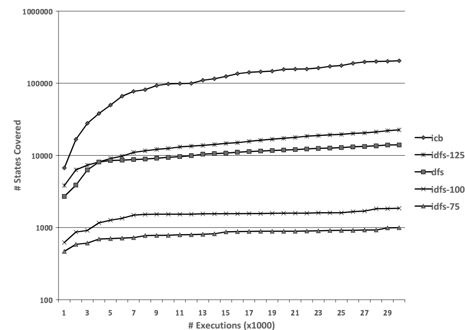


Figure 6. Coverage growth for Dryad

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 [21] 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 [20] 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, 19, 18, 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 [24] 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 [25] 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 context switches. We show that that by bounding the number of context switches, 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 7 previously unknown bugs in realistic multithreaded benchmarks. Each of these bugs required at most 2 context switches. 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 context switches. 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 find that on very large benchmarks, search does not terminate even for a context-switch bound of 2. 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.

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*, 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*. ACM, 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, 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, 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] A. Mazurkiewicz. Trace theory. LNCS 255, pages 279–324. Springer-Verlag, 1987.

[17] Madanlal Musuvathi, David Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Operating Systems Design and Implementation*, dec 2002.

[18] 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.

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

[20] 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.

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

[22] 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.

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

[24] 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.

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