Appears in the Proceedings of the 15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming

GAMBIT: Effective Unit Testing for Concurrency Libraries

Katherine E. Coons* Sebastian Burckhardt Madanlal Musuvathi
University of Texas, Austin* Microsoft Research
coonske@cs.utexas.edu sburckha@microsoft.com madanm@microsoft.com

Abstract

As concurrent programming becomes prevalent, software providers are investing in concurrency libraries to improve programmer productivity. Concurrency libraries improve productivity by hiding error-prone, low-level synchronization from programmers and providing higher-level concurrent abstractions. Testing such libraries is difficult, however, because concurrency failures often manifest only under particular scheduling circumstances. Current best testing practices are often inadequate: heuristic-guided fuzzing is not systematic, systematic schedule enumeration does not find bugs quickly, and stress testing is neither systematic nor fast.

To address these shortcomings, we propose a prioritized search technique called GAMBIT that combines the speed benefits of heuristic-guided fuzzing with the soundness, progress, and reproducibility guarantees of stateless model checking. GAMBIT combines known techniques such as partial-order reduction and preemption-bounding with a generalized best-first search framework that prioritizes schedules likely to expose bugs. We evaluate GAMBIT's effectiveness on newly released concurrency libraries for Microsoft's .NET framework. Our experiments show that GAMBIT finds bugs more quickly than prior stateless model checking techniques without compromising coverage guarantees or reproducibility.

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

General Terms Algorithms, Reliability, Verification

Keywords Concurrency, model checking, partial-order reduction, preemption bound, multithreading, software testing

1. Introduction

Multiprocessor architectures have become uniquitous and, as a consequence, parallel and concurrent programming will become increasingly common. Engineering principles suggest that abstraction layers will play a key role in enabling programmers to write correct parallel and concurrent programs. Examples of this trend include the recent emergence of concurrency libraries such as the java.util.thread package [16], the Intel Threading Building Blocks [25], and the Microsoft Task Parallel Library [30, 18]. These libraries improve programmer productivity by providing control, synchronization, and data abstractions that simplify the design and implementation of concurrent and/or parallel programs.

While these libraries help simplify higher-level programs, implementing the libraries themselves is challenging. Users of concurrency libraries expect consistent high performance. In response, library developers often optimize their code with sophisticated synchronization techniques that are fast, but error-prone. At the same time, correctness is crucial for these libraries. Bugs in libraries often introduce insidious errors in seemingly unrelated modules, making concurrent applications, which are already difficult to debug, even *more* challenging. For example, prior work [20] describes a single bug in a low-level library that caused unreproducible errors in more than 30 independent tests.

This paper is motivated by experience with developers and testers using a model-checker to find concurrency errors in industry-scale concurrency libraries. We use unit tests – many small test cases that exercise different scenarios, and focus specifically on *concurrent unit tests*. These tests use both external concurrency, where multiple threads call into the library, and internal concurrency, where multiple library-internal threads operate concurrently. Each test is short, but the set of all possible thread interleavings for a given test is still very large, and the probability of choosing a schedule that triggers a bug by pure chance may be very low.

While working with the test team we learned that testers initially want to find bugs very quickly, in an almost interactive way. After finding as many bugs as possible, however, they are willing to run long background jobs for *proof* of coverage guarantees. Such guarantees allow the test team to identify when to stop testing and reallocate test resources appropriately. The following three requirements summarize our experience:

Fast response: Most bugs should be found very quickly. Bugs may manifest as livelocks, deadlocks, or assertion failures.

Reproducibility: Once a bug has been found, it should be easy to reproduce. In particular, it must be possible to attach a debugger and step through the schedule that causes the bug.

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

Strategies to detect concurrency bugs fall into three classes, which meet these requirements to varying degrees:

1. *Stress testing* repeatedly runs the test under heavy load with random noise to increase the probability of executing a rare schedule. Stress testing provides fast response during the initial stages of software development, but fails to perform as the software matures, even if the software still contains errors. Stress testing does not provide reproducibility or coverage.

2. *Heuristic-based fuzzing* uses heuristics such as patterns that indicate potential deadlocks or atomicity violations to direct an execution towards an interleaving that manifests a bug [7, 29, 22, 15]. While these techniques often provide fast response, they do not provide coverage guarantees.
3. *Stateless model checking* systematically enumerates all schedules to provide full coverage [10], or all schedules within a preemption bound to provide *preemption-bounded coverage* [19]. Model checkers provide coverage guarantees and reproducibility, but they do not typically provide fast response.

Tools in each class provide either fast response or precise coverage guarantees, but they seldom provide both. Unfortunately, fast response and precise coverage often involve conflicting goals. Fast response favors a targeted search with flexible heuristics. Coverage guarantees require systematic search, proofs, and sound reduction techniques[1]. To balance these goals, we introduce an algorithm and a tool called GAMBIT that prioritizes a systematic search without compromising coverage.

GAMBIT provides a generalized best-first search in a stateless model checker. To provide fast response, GAMBIT supports weighted combinations of heuristics that can be customized by the user. Regardless of the heuristics chosen, however, the search (1) does not waste time repeating the same schedule, and (2) eventually explores all schedules, thus guaranteeing the desired level of coverage (full or preemption-bounded). Moreover, because GAMBIT controls all nondeterministic scheduling choices, GAMBIT vastly improves the chances of reproducing a bug.

A key problem with any best-first strategy is state-management. To alleviate this problem, we propose a compressed representation called an *execution tree* that stores discovered, yet unexecuted schedules. The execution tree 1) stores only deltas from a default behavior, 2) stores only executions that still need to be performed, rather than executions that have already been performed and (3) uses space proportional to the number of high-priority schedules, rather than the total number of schedules. Our experiments show that this compressed representation reduces the space overhead of the best-first search by 10x on average, compared to a standard representation used by previous stateless model checkers. This compression is effective enough that we do not need to use a disk, as is commonly done for best-first search in other contexts. Our experiments run out of time before they run out of space.

GAMBIT is built on top of the CHESS tool [20], but the techniques it uses are applicable to other stateless model checkers. We evaluate a suite of heuristics on 23 concurrency bugs in unit tests for libraries under development at Microsoft. We find that an hour-long search using the default CHESS settings is unable to find 9 of the bugs. In contrast, GAMBIT finds all 23 bugs in less than two minutes, with an appropriately chosen heuristic. Similarly, for a full-coverage search using partial-order reduction (with no preemption bound), CHESS is unable to find 10 of the bugs within an hour, while GAMBIT finds all of them in less than two minutes. We also find that different heuristics work best for different tests. These results suggest that the best way to use GAMBIT is to run multiple versions in parallel with each test using a different heuristic.

2. Background

This section describes two techniques to alleviate the state-space explosion problem in model checkers for concurrent software: partial-order reduction, and bounded search. GAMBIT is complementary to both techniques, and we use them to augment the search.

[1] Testing tools often neglect proofs, probably because they are thought to be infeasible in practice. Our experience, however, shows that for concurrency unit testing, coverage guarantees such as preemption-bounded coverage are worth striving for and greatly improve the user experience.

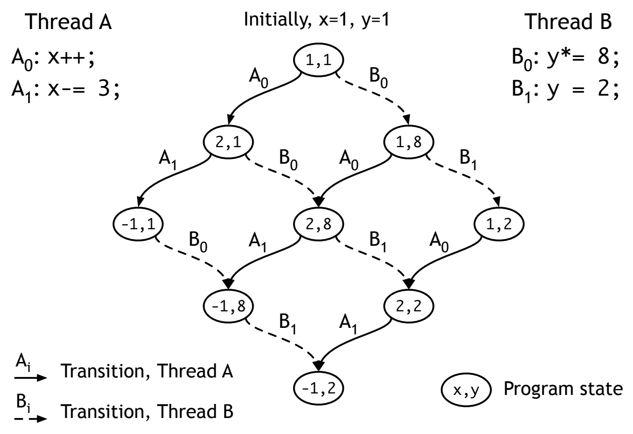


Figure 1. State space for two threads operating on volatile variables x and y.

2.1 Partial-order reduction

Partial-order reduction identifies equivalent thread interleavings that lead to the same program state. By avoiding redundant thread interleavings, partial-order methods can reduce the number of schedules explored without compromising coverage.

Partial-order methods identify *independent* transitions and exploit their commutativity; the search will reach the same state with each interleaving of independent transitions. Flanagan and Godefroid define independence and summarize it informally as follows: independent transitions can neither enable nor disable one another, and enabled independent transitions commute [8]. For example, two attempts to acquire the same lock are dependent, yet attempts to acquire two different locks are independent.

Figure 1 shows a state transition diagram for two threads, Thread A and Thread B, each executing a short program. Nodes in this diagram represent program state including global variables, the heap, the stack, and kernel state. Edges represent transitions, or synchronization variable accesses, that take the program to a new state. There are six possible paths through the diagram, thus a standard model checker would require six executions to ensure full state-space coverage. All six of these executions lead to the same final state, however. An optimal partial-order reduction algorithm would require only one execution to uncover all possible states.

Partial-order reduction algorithms can be categorized into *persistent set* techniques and *sleep set* techniques [9]. Persistent set techniques conservatively reason about the possible future actions that a thread may perform. Using this information, a persistent set algorithm selects only a subset of the enabled threads in each state and guarantees that none of the unselected transitions will interfere with their execution [21, 12, 31]. While most persistent set techniques use static analysis to reason about the future actions a thread may perform, Flanagan and Godefroid introduce a dynamic partial-order reduction (DPOR) algorithm that improves over static algorithms by exploiting dynamic information about dependences between transitions [8]. We implement the DPOR algorithm to represent the state-of-the-art in partial-order reduction techniques.

Sleep sets use information about the enabled transitions and the past of the search to avoid different interleavings of independent transitions [9]. Sleep sets are complementary to persistent sets and combining them further reduces the size of the search space. We

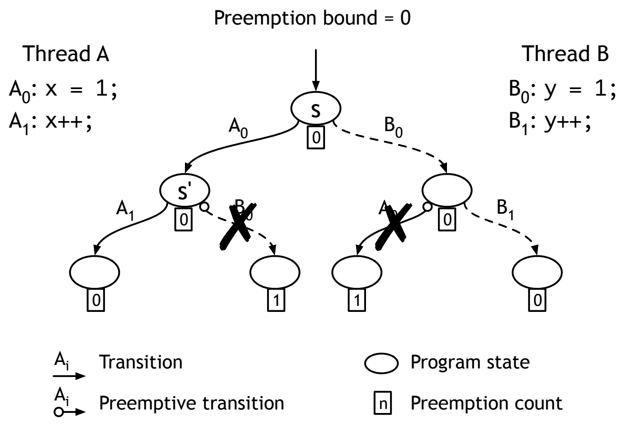


Figure 2. Threads A and B perform otherwise independent transitions A_1 and B_1. Executing either of these transitions disables the other, however, so A_1 and B_1 are not independent.

implement sleep sets as well and use them both as a heuristic and a reduction technique.

2.2 Bounded search

During a bounded search, the model checker does not explore executions that exceed an arbitrary bound on some property of the execution. A bounded search is not sound; it does not maintain full coverage of the underlying state-space. A bounded search may, however, guarantee that safety properties hold among all executions that do not exceed the bound. Many model checkers bound the search based on depth; the model checker does not explore any executions that contain more than n steps. An alternate way to bound the search is to limit the number of preemptions that occur during an execution [19]. A preemption bound may be more useful than a depth bound because it limits the size of the state-space significantly without arbitrarily limiting the depth in the execution. In addition, Musuvathi and Qadeer empirically found that a small number of preemptions is sufficient to expose many bugs [19].

2.3 Combining bounded search and partial-order reduction

Combining partial-order reduction with bounded search raises several problems. Partial-order reduction is possible because different interleavings of *independent* transitions cannot lead to new states. Two transitions are independent if they commute, and they can neither enable nor disable one another [8]. In an unbounded search, this usually means that transitions by different threads that operate on different synchronization variables are independent. Once the execution is bounded, however, these transitions may no longer be independent.

After reaching the bound in a bounded search, any future transition that increases the bounded value becomes disabled. Using a preemption bound, executing a preemption may disable any future transition that requires a preemption. Instructions that do not operate on the same synchronization variables or communicate in any way may disable one another, making independence difficult to prove even in a dynamic setting.

To illustrate this point, Figure 2 shows an example in which threads A and B perform otherwise independent transitions A_0 and B_0. The following condition must hold if A_0 and B_0 are independent [8]: if A_0 is enabled in a state, s, and s' is the state reached after taking transition A_0 from s, then B_0 is enabled in s iff B_0 is enabled in s'. A_0 and B_0 do not satisfy this condition because B_0 is enabled in s, the initial state, but is not enabled in s', the state reached after executing A_0, as it would violate the preemption bound if executed from s'. Thus, A_0 and B_0 are not independent even though they do not access the same synchronization variables or communicate in any way. This example shows why naïvely combining partial-order reduction with bounded search is not sound. Instead, we use the intuitions from partial-order reduction to *prioritize* rather than prune the search. Likewise, we use the number of preemptions to prioritize searches using partial-order reduction.

3. Best-first search

GAMBIT uses greedy best-first search to prioritize the large search space of possible schedules. Traditional best-first search [17] is an algorithm that generates a graph by maintaining sets of open and closed nodes. Open nodes are nodes whose successors have not yet been generated. Closed nodes are nodes whose successors have already been generated. In every iteration, the algorithm removes the open node with the highest priority from the list of open nodes, generates all of its successors, and inserts the node into the list of closed nodes. Then, it inserts each newly generated successor into the list of open nodes, provided that it is not already present in either list. The algorithm terminates once all nodes are closed.

In our case, nodes in the graph represent an execution of the program; a unique sequence of transitions. This graph is always a tree. Thus, whenever we generate a node during best-first search, it is guaranteed to be a new node and we can insert it into the list of open nodes without performing a check. This also implies that we need not maintain a list of closed nodes. We call the list of open nodes the *fringe*.

Unlike a depth-first search, where storage scales with the length of the longest simple path through the graph, the storage overhead of best-first search scales with the size of the graph. Thus, a concise representation that limits space overhead is desirable. We refer to the graph of states and transitions in a search as the *schedule space*, and describe two different representations of this schedule space in the next section: a straightforward one, and a more efficient one.

3.1 Uncompressed schedule space

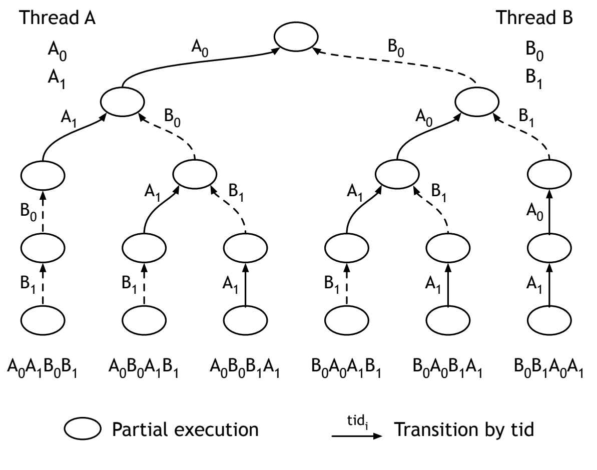
We can represent the schedule space as a tree of partial schedules as shown in Figure 3(a), where arrows point from child nodes to their parent node. Each node at depth k represents a partial schedule of length k, and its successors are the partial schedules that extend their parent's schedule by one more step. Each node has as many successors as there are enabled threads after executing the partial schedule that node represents.

Each node stores only the last transition, and a pointer to its parent. The partial schedule corresponding to a given node can be reconstructed by following the pointers to the root. Each node in the tree in Figure 3(a) represents a partial path through an execution, so the storage overhead scales with the number of partial states. We propose a state-space representation in which the storage overhead instead scales with the number of complete executions, or the number of leaf nodes in Figure 3(a).

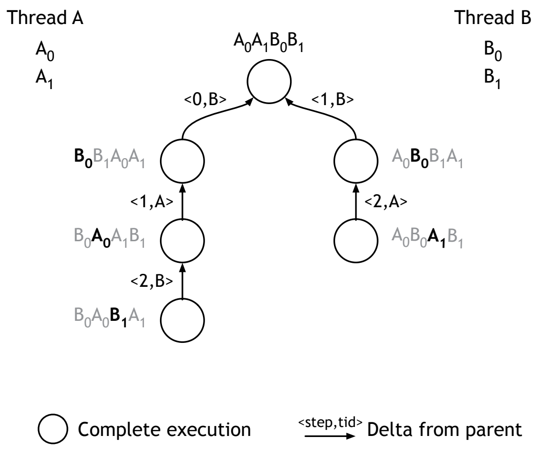
3.2 Execution trees

An *execution tree* is a compressed schedule space representation that allows storage overhead to scale with the number of unique program executions rather than the total number of partial states. Figure 3 compares the full schedule space for a simple program to its execution tree. In both cases, the only information actually stored is the information along each edge.

The execution tree in Figure 3(b) leverages the systematic state-space search by storing only deltas from other executions. Each node represents a complete *execution* of the program, and each edge represents a delta from its parent's execution. The correspond-



(a) Uncompressed representation.



(b) Compressed representation.

Figure 3. Compressed and uncompressed schedule space representation for two threads each performing two operations.

ing execution appears next to each node, and bold transitions represent the point at which the delta occurs.

We assume that the model checker has some *default behavior* that can be systematically varied. Our model checker's default behavior is to never preempt the running thread, and to always schedule the enabled thread with the minimum thread ID when the running thread blocks. Any default behavior is acceptable provided that it can be systematically varied. The root node in Figure 3(b) represents the initial execution, which always performs the default behavior. The two edges pointing from its successors represent steps in that initial execution at which a different thread could have been executed: at step 0, thread B could have been executed, $\langle 0, B\rangle$; at step 1, thread B could have been executed, $\langle 1, B\rangle$. The root node's two successor nodes represent the executions obtained by always performing the default behavior except at these steps. The remaining nodes follow a similar pattern.

The model checker can replay any execution by performing the default behavior except at the locations indicated along the edges leading to that node. For example, to replay the execution associated with the node labeled $A_0B_0B_1A_1$, the model checker would perform the default behavior until step 1. At step 1, the default behavior would be to schedule thread A again because scheduling thread B requires a preemption, but the edge labeled $\langle 1, \mathrm{B}\rangle$ indicates that thread B should be scheduled at step 1 instead. The rest of the execution performs the default behavior, so B is not preempted at step 2, and A is allowed to execute again only once B has terminated, producing the execution $A_0B_0B_1A_1$. Each node in Figure 3(b) is equivalent to a leaf node in Figure 3(a). The execution tree eliminates storing information for steps at which the model checker performs its default behavior. We perform a best-first search on this execution tree.

3.3 Best-first search algorithm

Figure 4 shows how the execution tree can be incorporated into a best-first search in a stateless model checker. A node in the execution tree in Figure 3(b) is called an ExecTreeNode. Each node stores a link to its parent, a bit indicating whether it requires a preemption, the $step$ at which it diverges from the behavior of its parent, and the thread it should perform at that step, tid. The $step$ and tid represent a backtracking point from its parent execution.

The `BestFirstSearch` routine in lines 1-16 first creates the root node, initialized at line 2, which represents the initial, default execution. The loop in lines 4-15 iterates through all nodes beginning with the root node and continuing in priority order. Although the root node is initially the only node in the fringe, each iteration potentially adds new nodes to the fringe. Each $node$ provides information to perform one unique execution of the program, and at line 5 the model checker executes the program based on this information. The `Execute` routine runs the program using the default behavior except at the steps associated with nodes between $node$ and the root node, as described in Section 3.2. This `Execute` routine returns a data structure, $exec$, that contains information about the execution including the transition executed at each step and the enabled threads at each step.

At line 6 the algorithm checks whether the search is bounded. If the search is bounded, then the algorithm finds backtracking points via the `FindBacktrackingPoints` routine, which adds all backtracking points that do not exceed the bound. If the search is unbounded, then the algorithm calls `FindDporBacktrackingPoints`, which uses a modified form of the DPOR algorithm [8] to update the persistent set at each step.

The `FindBacktrackingPoints` routine in lines 17-26 adds new executions (nodes) to the fringe for a preemption-bounded search. In lines 18-25, the search iterates through all steps from $node.step$ to the end of the execution. These steps represent the new portion of the execution, after it diverges from its parent's execution at $node.step$. At each of these previously unseen steps, lines 20-24 iterate through all threads that were enabled but were *not* executed. Line 21 identifies these threads and checks whether executing them will exceed the preemption bound. For each unexecuted $\langle step, tid\rangle$ pair that does not exceed the preemption bound, the routine calls `AddBacktrackingPoint` at line 22, which creates a successor for $node$ in the execution tree and adds this new node to the fringe with its corresponding priority value.

Lines 27-41 show this same process for a DPOR search. This routine iterates through new transitions in lines 28-40 and through every thread, both enabled and disabled, in lines 29-39. Line 30 finds the *lookahead* transition for each thread, which is the next transition that thread will perform. GAMBIT stores lookahead values during execution, so they are immediately available. Line 31 finds the most recent prior transition that conflicts with tid's lookahead. Unlike preemption-bounded search, where all new backtracking points occur below $node.step$, DPOR may produce a value for $bstep$ that is higher in the execution than $node.step$. As a result,

```
 1: procedure BestFirstSearch() begin
 2:    $root :=$ ExecTreeNode(null, 0, 0, 0);
 3:    $node := root$;
 4:    while $node \neq$ null do
 5:       $exec :=$ Execute($node$);
 6:       if $boundedSearch$ then
 7:          FindBacktrackingPoints($node$, $exec$);
 8:       else
 9:          FindDporBacktrackingPoints($node$, $exec$);
10:       end if
11:       if $node.refCount == 0$ then
12:          $node$.Detatch();
13:       end if
14:       $node := fringe$.RemoveLowestCostNode();
15:    end while
16: end
17: procedure FindBacktrackingPoints(ExecTreeNode $node$,
    Execution $exec$) begin
18:    for $s \in [node.step, exec.$NumSteps()$]$ do
19:       $threadExecuted := exec.$Transition$(s).tid$;
20:       for all $tid \in exec.$EnabledThreads$(s)$ do
21:          if $tid \neq threadExecuted$ && $preemptBound >=$
             NumPreempts($exec, node, s, tid$) then
22:             AddBacktrackingPoint($exec$, $node$, $s$, $tid$);
23:          end if
24:       end for
25:    end for
26: end
27: procedure FindDporBacktrackingPoints(ExecTreeNode $node$,
    Execution $exec$) begin
28:    for $s \in [node.step, exec.$NumSteps()$]$ do
29:       for all $tid \in exec.$GetThreads() do
30:          Transition $lookahead := exec.$Lookahead$(s, tid)$;
31:          $bstep :=$ StepOfMostRecentConflict($s, lookahead$);
32:          if $exec.$IsEnabledAtStep($bstep, tid$) then
33:             AddBacktrackingPoint($exec$, $node$, $bstep$, $tid$);
34:          else
35:             for all $t \in exec.$GetEnabledThreads($bstep$) do
36:                AddBacktrackingPoint($exec$, $node$, $bstep$, $t$);
37:             end for
38:          end if
39:       end for
40:    end for
41: end
42: procedure AddBacktrackingPoint(Execution $exec$, ExecTreeNode
    $node$, int $step$, Thread $tid$) begin
43:    $succ := node.$CreateSuccessor($step, tid$);
44:    $fringe$.Insert($succ$, GetCost($exec$, $node$, $step$, $tid$));
45: end
```

Figure 4. Best-first search using DPOR/preemption bounding.

each node in the DPOR search must keep track of its successors, until all of them have been deleted, to avoid redundant work.

After computing the backtracking step, Line 32 checks whether tid is enabled at $bstep$, and if so adds a backtracking point there. If tid is not enabled, then the DPOR algorithm conservatively adds a backtracking point for every enabled thread at $bstep$. This algorithm explores the same executions that the depth-first DPOR algorithm explores, but it may explore them in a different order.

After generating new nodes, Line 11 checks whether any new nodes were actually created and if not, Line 12 deletes $node$. Deleting $node$ is sound in both preemption-bounded search and DPOR, if $node$ has no successors. In preemption-bounded search, $node$ will never acquire new successors after `FindBacktrackingPoints` completes. With DPOR, $node$ may acquire additional successors, but only after executing other successors of $node$, of which there are none. Thus, if $node$ has no successors now, then it will never have any, and deleting $node$ will not sacrifice soundness.

The ExecTreeNode's `Detatch` method, called at line 12, removes $node$ from the execution tree by decreasing its parent's reference count. If, as a result, its parent has no successors, then $node$ recursively detatches its parent (which may then detatch its own parent, etc.). Thus, ExecTreeNodes delete themselves and release any memory they have allocated as soon as they become unnecessary. After all executions have been performed, the tree will therefore be completely empty aside from the root node. The tree both discovers itself dynamically and deletes itself dynamically, so the entire tree is never in memory. Every leaf node in the tree is in the fringe and must still be executed, and every interior node has already been executed and has at least one descendent leaf node that is in the fringe. Line 14 removes the leaf node with the lowest priority from the fringe so it can be executed during the next iteration.

The space required by the execution tree scales with the number of executions that have been discovered, but have not yet been executed. In the tests we performed, GAMBIT can run for days using the best-first search without running out of memory. If running for very long periods is desirable, however, or the state-space for a program is very large, then writing the lowest priority nodes to disk is a simple addition to GAMBIT. The nodes in the frige are all leaf nodes and will not be touched until they are ready to be executed, so writing them to disk should have a low performance impact. Section 4 describes priority functions that provide the tester with leverage in prioritizing this search.

4. Priority functions

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

1. Find new happens-before executions at a faster rate.
2. Provide a randomized search with progress guarantees.
3. Allow the tester to guide the search.
4. Target known bug patterns.

We implemented priority functions using the first three approaches. Heuristics from prior work [22, 15] could target the search at known bug patterns, but we chose more general heuristics so that we could detect many different types of bugs quickly.

4.1 Prioritizing new happens-before executions

A *happens-before execution* is an execution with a unique partial-order on its synchronization variable accesses, or happens-before graph. A priority function can guide the search towards new happens-before executions, and thus towards previously unexplored program behavior. The following priority functions use unsound reductions as priority functions:

BF(*pb*) prioritizes executions with fewer preemptions. This priority function allows algorithms using persistent sets or sleep sets to also favor fewer preemptions without sacrificing coverage, though it cannot provide guarantees about coverage within the preemption bound for these searches.

BF(*dpor*) uses the DPOR algorithm [8] to prioritize backtracking points; those that would be performed by DPOR have higher priority than those that would not. A preemption-bounded search cannot prune these schedules without sacrificing coverage, but it can defer them.

BF(*ss*) gives preference to backtracking points that appear useful according to the sleep sets algorithm. Like persistent sets, sleep sets are not sound in a preemption-bounded search.

BF(*mdpor*) defers backtracking points that are conservatively added by DPOR, including those added by Lines 35-37 of Figure 4, and those that occur prior to a release operation that cannot soundly be pruned.

These examples show how GAMBIT uses insights from unsound reduction techniques to prioritize the search without sacrificing coverage. We use the unsound combination of DPOR and sleep sets with preemption-bounding as an example.

4.2 Random search

Random search is often surprisingly effective in a large search space [4]. A pure random search provides no progress guarantees, however, and may fail to find existing bugs. GAMBIT provides randomization while guaranteeing progress and coverage by randomly walking a program's execution tree:

BF(*rand*) assigns a random priority to each execution.

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

4.3 Tester input

Unit tests are often designed to test a specific functionality. The unit tests for concurrent data structures that we used often test the interaction of specific methods, for example a simultaneous `Enqueue` and `Dequeue` from a concurrent queue. The tester can target the search at specific methods by specifying them at the command line and using the following priority function:

BF(*method*) returns one of the following three priorities:

> **High:** if the method being preempted is specified, *and* the method preempting it is specified as well.
>
> **Medium:** if the method being preempted is specified, *or* the method preempting it is specified, but not both.
>
> **Low:** if neither the method being preempted nor the method preempting it is specified.

The *method* priority function allows the tester to target specific methods. This priorify function may be useful for regression testing, where only a subset of methods have been modified, or reproducing errors from a stack trace.

BF(*var*) allows the tester to target specific variables. This priority function is similar to the *method* priority function, but it reasons about variables rather than methods. This priority function may be useful to target data races, or for regression testing.

We provide these priority functions as examples, but GAMBIT is totally general and adding new priority functions is very simple. Each priority function implements a single method,

$$\text{int } \texttt{GetPriority}\ (exec, bstep, tid)$$

where $exec$ is the completed execution being backtracked, $bstep$ is a step in $exec$ at which a different thread could have been executed, and tid is an alternative thread to execute at $bstep$. We make the individual priority functions described above more powerful by combining them hierarchically.

4.4 Hierarchical priority functions

We include a hierarchical priority function that combines individual priorities into a single value. The priority function listed first receives the highest priority. When the first priority value is equivalent, ties are broken by the second priority function, etc. We indicate hierarchical priority functions by separating them with commas. For example, the priority function BF(*mdpor*,*pb*) would first prioritize backtracking points according to the BF(*mdpor*) priority function. Among executions whose values for BF(*mdpor*) were equal, those that contain fewer preemptions (smaller values for BF(*pb*)) would be executed first. The tester can combine any number of priority functions. Other operators are possible, as well, but we do not evaluate other operators here.

When priority values for two executions are completely equal, the fringe always returns the execution that was most recently added. This behavior is equivalent to a depth-first search, and helps keep the space requirement of the best-first search reasonable. If an entire portion of the search space has the same priority value, then it will be searched in a depth-first manner and have little impact on the overall space requirement.

4.5 Selecting priority functions

We tried a variety of priority functions and present results for the most successful ones in Section 5. Prioritizing executions with fewer preemptions was very effective for full-coverage searches using DPOR. As the number of preemptions grows, the DPOR algorithm becomes increasingly likely to swap conflicting accesses and return to an already visited state. Sleep sets help combat this problem, but we cannot prune a portion of the state-space due to sleep sets without performing at least one redundant execution. Prioritizing by preemption count helps defer these redundant executions. In addition, testers find preemption points useful for identifying the root cause of a bug. Traces that contain fewer preemptions are thus preferable, as they make it easier to identify the root cause.

The modified DPOR priority function also improved the speed with which the DPOR search finds bugs. Thus, we combined these heuristics to find bugs even faster. The BF(*mdpor*,*pb*) priority function first prioritizes by the modified DPOR priority function, and among executions with equivalent priority it chooses executions that contain fewer preemptions. We found this priority ordering was preferable because the fewer preemptions priority function was more likely to defer useful backtracking points, whereas the modified DPOR priority function typically defers redundant executions.

The preemption-bounded search uses priority functions to gain some of the benefits of DPOR and sleep sets without sacrificing preemption-bounded coverage. The BF(*ss*,*dpor*) priority function, which prioritizes first by sleep sets and then by DPOR, was particularly useful with preemption-bounded search. We prioritized by sleep sets first because CHESS's sleep sets implementation includes a special case for non-preemptive context switches, whereas the DPOR implementation does not. Non-preemptive context switches can be critical in a preemption-bounded search because they expose a whole new segment of the search space that can be accessed within the preemption bound. DPOR is thus more likely than sleep sets to defer an execution that is really necessary. To further improve the preemption-bounded search, we replace the DPOR priority function with the modified version, BF(*ss*,*mdpor*). We evaluate these priority functions on a set of concurrency bugs in the next section.

5. Results

We evaluate the speed with which GAMBIT finds known bugs in unit tests for concurrent libraries under development at Microsoft. We compare different priority functions and also evaluate the memory usage of the execution tree.

5.1 Methodology

We evaluate GAMBIT on both preemption-bounded search and unbounded search with dynamic partial-order reduction (DPOR).

Program	Unit test (max steps)	Description
CCR	Iterator (165), Causality (2615), ScatterGather (12156), TaskCoverage (203), GatherPost1 (142), GatherPost2 (164)	Provides a highly concurrent programming model based on message-passing with powerful orchestration primitives enabling coordination of data and work.
RegionOwnership	RegOwn (277)	Supports ownership-based separation of the heap for parallel programs.
SYN	Barrier1 (124), Barrier2 (102), ManualResetEventSlim(MRSE) (93), SemaphoreSlim (120)	Low-level synchronization primitives.
CDS	ConcBag1 (944), ConcBag2 (336), BlockingColl (936)	Basic parallel data structures for .NET 4.0.
TPL	NQueens (1079)	Provides support for imperative task-parallelism in .NET 4.0
PLINQ	NQueens (972), ParallelDo (2721)	Provides support for declarative data-parallelism in .NET 4.0.

Table 1. Programs and corresponding unit tests. Each unit test includes the maximum number of synchronization variable accesses observed in a single run in parenthesis.

We compare directly to the CHESS tool's default configuration, which uses a depth-first search with a preemption bound of two. The only modification we made to CHESS's default configuration was to disable sleep sets, which are used by default in CHESS even though they are not sound with the default configuration.

Table 1 describes the unit tests that we used to evaluate GAMBIT. These unit tests were developed by testers for concurrent libraries and data structures at Microsoft such as the Concurrency Coordination Runtime (CCR) [1] and the .NET 4.0 concurrency libraries. Column 2 provides, in parentheses, the maximum number of transitions performed in a given execution for each unit test. This number is generally correlated with the size of the state-space.

CCR ships as part of the Microsoft Robotics Studio Runtime and provides primitives for building asynchronous programs, similar to those required for a robot controller. RegionOwnership tests a library that supports ownership-based separation of the heap for parallel programs. The remaining unit tests are for parts of the .NET 4.0 concurrency library. At the lowest level are unit tests for synchronization primitives including barriers and lightweight versions of a semaphore and a manual reset event. On top of those synchronization primitives are several concurrent data structures (CDS) including a concurrent bag and a blocking collection. TPL is a task-parallel library that creates lightweight tasks for imperative task-parallelism. Finally, PLINQ is an implementation of the declarative data-parallel extensions to the .NET framework [3]. All of these unit tests were written by the testers of the respective products.

5.2 Time to find bugs

Table 2 shows how many seconds were required to find each bug with different priority functions. Column 2 contains the type of bug. Multiple entries appear for unit tests that contain multiple bugs. Columns labeled DPOR use both DPOR and sleep sets and columns labeled PB(2) use a preemption bound of two. The "DF" columns contain baseline depth-first results for each search. The PB(2), DF results are the CHESS default. Cells marked "–" indicate that the search did not find the bug within one hour. The row labeled "Total hours" shows the total number of hours required to run all 23 unit tests given a one-hour time limit per test, assuming each test terminates after finding a bug. The row labeled "Bugs found" shows the number of bugs that would be found during that time period with each heuristic.

Using BF(*pb*) to favor fewer preemptions with DPOR reduces the time required to find the bug in every case. Adding the modified DPOR priority function, BF(*mdpor*,*pb*), strictly improves the time required to find the bug over BF(*pb*). The depth-first search required 11.5 hours to complete all 23 unit tests, and it only found 13 bugs during that time. The BF(*dpor*,*pb*) test required 0.4 hours to find all 23 bugs, and the BF(*mdpor*,*pb*) test required slightly less time, 0.3 hours, to find all 23 bugs.

The *ss* and *dpor* priority functions improve the preemption-bounded search significantly in all but one case. In ScatterGather, the *ss* and *dpor* priority functions actually *defer* the execution that causes the bug. The preemption-bounded search for this unit test requires a non-preemptive context switch, which would be pruned by DPOR in an unbounded search, to manifest the bug within the preemption bound. Rather than choosing this non-preemptive context switch, the DPOR algorithm selects a preemptive context switch that has the same effect on the happens-before graph, yet exceeds the preemption bound. This shortcoming illustrates why DPOR cannot be trivially combined with preemption-bounded search. Modifying the algorithm to give special treatment to non-preemptive accesses could alleviate this problem.

Replacing the DPOR priority function with the modified DPOR priority function always improves the time required to find the bug in preemption-bounded search. The depth-first search required 11.8 hours to complete all 23 tests, and it found 14 bugs during that time. The BF(*ss*,*dpor*) search required 2.4 hours to complete all 23 tests, and it found 21 bugs. The BF(*ss*,*mdpor*) search required 1.2 hours and found 22 bugs.

We performed each test 10 times using the *rand* priority function with the time as the random seed. The average, minimum, and maximum number of seconds required to find the bug appear in Table 2. The minimum time to find the bug with *rand* was at least as small as with any other priority function in all but three cases. The maximum time to find the bug was sometimes significantly higher, however. The time required to find the bug in Barrier1 varied from one second to longer than an hour depending on the random seed. We suggest performing multiple tests in parallel with different ran-

Test	Bug	Time to find bug (seconds)											
		DPOR						PB(2)					
		DF	BF (*pb*)	BF (*mdpor*,*pb*)	BF(*rand*) avg	min	max	DF	BF (*ss*,*dpor*)	BF (*ss*,*mdpor*)	BF(*rand*) avg	min	max
Iterator	Livelock	82	4	3	6	3	11	78	9	5	4	1	4
Causality	Livelock	–	748	735	90	4	265	–	367	70	18	5	61
ScatterGather	Livelock	784	94	33	41	16	88	1484	–	–	4	2	13
TaskCoverage	Livelock	1059	176	35	27	1	44	–	13	5	238	3	1202
GatherPost1	Assertion	1540	3	2	24	3	65	314	6	3	5	1	19
GatherPost2	Assertion	–	31	15	34	3	139	–	592	236	10	1	14
RegOwn	Assertion	–	265	52	72	2	371	–	92	11	–	–	–
Barrier1	Assertion	–	1	1	N/A	1	–	2654	18	18	1	1	3
Barrier2	Assertion	844	1	1	114	1	728	143	15	11	4	1	12
MRSE	Deadlock	61	21	15	34	3	54	108	–	54	102	28	218
SemaphoreSlim	Assertion	2	1	1	2	1	7	146	1	1	1	1	1
ConcBag1	Assertion	35	2	1	21	1	11	458	4	2	6	1	21
ConcBag2	Assertion	–	5	3	93	4	619	–	15	7	6	1	16
BlockingColl	Deadlock	–	84	72	190	20	1474	–	153	71	632	89	2362
NQueens	Livelock	2	2	1	2	1	5	140	2	2	4	1	7
NQueens	Assertion	–	60	36	8	2	26	1659	20	11	28	5	91
NQueens	Assertion	–	63	37	7	1	15	1595	18	10	20	2	73
NQueens	Assertion	–	3	2	8	2	25	–	17	10	73	2	323
NQueens	Assertion	227	3	2	6	1	12	–	7	3	20	3	73
NQueens	Assertion	–	3	2	9	1	39	–	17	10	52	8	157
ParallelDo	Assertion	1	1	1	3	1	5	63	1	1	18	1	59
ParallelDo	Assertion	462	24	15	10	1	28	597	23	19	18	5	37
ParallelDo	Assertion	317	24	15	20	1	16	548	21	17	21	2	68
Total hours		**11.5**	**0.4**	**0.3**	**1.2**			**11.8**	**2.4**	**1.2**	**1.4**		
Bugs found		**13**	**23**	**23**	**22**			**14**	**21**	**22**	**22**		

Table 2. Time in seconds required to find each bug with different search strategies. "–" indicates that the search took longer than an hour. If the tester set all unit tests to run for at most one hour, then "Total hours" is the number of hours required to run the entire test suite, and "Bugs found" is the number of bugs that would be found within that time period.

dom seeds, or using random in conjunction with the other priority functions. The space overhead with random search may be higher, however, because fewer executions have equal priority, which sacrifices the space benefit of depth-first search among equal priority executions. We are currently investigating probabilistic approaches that harness the power of random search.

5.3 Prioritizing based on user input

We used a case study to test the BF(*method*) priority function. The GatherPost2 unit test ran for an entire long weekend without finding the bug using depth-first search with DPOR. The number of transitions in GatherPost2 is relatively small (164), but the proportion of those synchronization variable accesses that conflict is very high. We reproduced the bug, identified the methods involved, and realized that they were fairly obvious from the unit test. One thread in GatherPost2 places a set of items on a port, and another thread dequeues those items. The methods preempted to cause the bug were the port's `Enqueue` and `TryDequeue` methods.

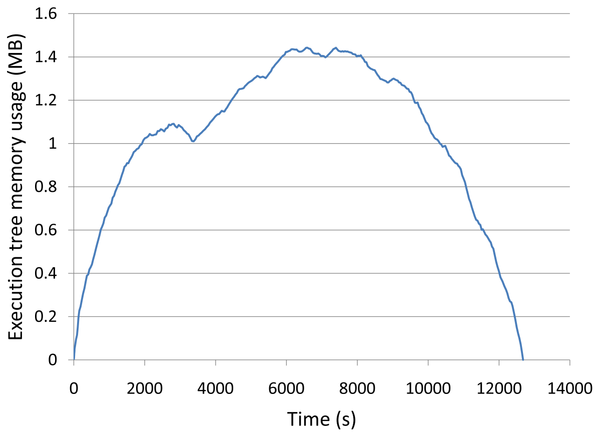
This example helped motivate the BF(*method*) priority function. By specifying both `Enqueue` and `TryDequeue` at the command line, the BF(*method*) priority function found a bug that had not manifested all weekend in 7 seconds. Specifying only the `Enqueue` method manifested the bug in 68 seconds, and specifying only the `TryDequeue` method manifested the bug in 13 seconds. We suspect that this and the BF(*var*) priority function, which prioritizes variables, will help testers target data races, perform regression testing, and reproduce stress test failures given stack traces.

5.4 Space requirements

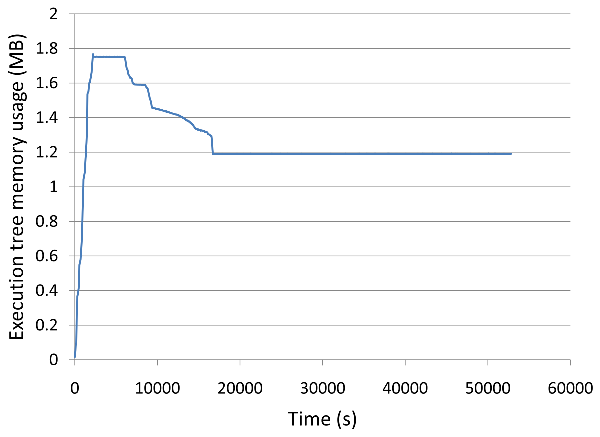
The primary downside to best-first search is its memory requirement. We recorded the number of nodes created and the number of nodes destroyed by the execution tree during a DPOR and a PB(2) search. For the DPOR search we used the best-performing non-random heuristic, BF(*mdpor*,*pb*). For the PB(2) search we used BF(*mdpor*). We did not use sleep sets with either test because our sleep sets implementation currently has a very high space overhead. This space overhead is totally orthogonal to the best-first search of the execution tree, however, and adding sleep sets would only improve the space requirements of the execution tree as it would either reduce or better target the search. We provide space results for a CDS BlockingCollection unit test that did not fail, so the test could be long-running. This test contained approximately 204 transitions.

To compute the memory required, we assumed each node in the execution would require 16 bytes; 4 bytes each for a parent pointer, the number of successor nodes, the backtracking step, and the thread to execute at that step. We recorded the number of nodes created and the number of nodes destroyed during the entire test process every 1,000 executions. The memory consumption we report is equal to $(nodesCreated - nodesDestroyed) * 16/1000000$ MB. We plot this value over time in Figure 5.

For both the DPOR and the PB(2) search we use a depth-first strategy among nodes with equal priority. The *fringe* that contains nodes waiting to be executed is organized as a set of bins of equal priority, where each bin is a stack. Thus, GAMBIT performs depth-first search among executions with equal priority, conserving memory and deleting nodes as quickly as possible. This organization helps prevent excessive growth in space overhead.



(a) DPOR with BF(*mdpor*,*pb*).



(b) PB(2) with BF(*mdpor*).

Figure 5. MB of storage consumed by the execution tree over time on a CDS BlockingCollection unit test (max 204 steps) that did not contain a bug.

In Figure 5(a) the underlying depth-first search is evident in the fluctuations in the memory usage. Each time a backtracking point high in the execution is chosen, GAMBIT learns of many new executions that it must store. The DPOR execution terminated at 326,743 executions. Because each node represents an execution, the total number of nodes created was exactly 326,743, as well. Nodes recursively delete themselves and their childless parents, however, as soon as their successors have all been executed, as described in Section 3.2. As a result, the maximum number of nodes alive in the system was 90,159. The additional space required to maintain the frontier is not included here, but it could at most double the memory usage.

Figure 5(b) shows a graph for a PB(2) search, where the effect of the priority function and the underlying depth-first search is particularly obvious. The DPOR priority function guides the preemption-bounded search towards new executions very quickly at first. At about 16,000 seconds (490,000 executions), however, the search stops discovering new behavior and the memory use becomes constant, as the search reverts to a depth-first search among low-priority executions. The memory requirement does eventually return to zero, but only after searching the entire preemption-bounded space (not shown).

These results suggest that for this test case, the DPOR algorithm would be preferable because it searches the *entire* state-space in less time than the PB(2) search searches the preemption-bounded state-space. This is not always the case, however. As the ratio of conflicting to total accesses grows, the preemption bound becomes more desirable for reducing search completion time. The amount of coverage it loses also increases, however. Fully evaluating the relative benefits of preemption-bounding and DPOR is beyond the scope of this paper, but these results suggest that there is an interesting tradeoff between the two.

6. Related Work

This work is closely related to and builds upon the CHESS tool [20]. CHESS uses model checking techniques [2, 24, 14, 32] to enumerate thread interleavings with a bound on the number of preemptions in each interleaving [19]. CHESS employs a simple partial-order reduction algorithm that relies on the fact that programs are mostly data-race free. CHESS uses a depth-first search within the preemption-bounded space. GAMBIT, in contrast, employs a best-first search and uses priority functions that provide the benefits of the more sophisticated dynamic partial-order reduction algorithm [8]. Given the complications involved in combining preemption-bounding and dynamic partial-order reduction, it is impossible to achieve the same degree of partial-order reduction in CHESS without resorting to the heuristics and state-management techniques proposed in this paper.

The *rand* priority function exploits the same insights that motivate Parallel Randomized State-Space Search (PRSS) [4] in Java PathFinder [32]. We reach a similar conclusion, that performing a few parallel searches using the *rand* priority function with different seeds would be beneficial. GAMBIT's random search differs from the PRSS random search, however; PRSS performs a depth-first, stateful search from random initial points, whereas GAMBIT performs a stateless random walk of the execution tree from a single initial point. These approaches could be complementary to one another: GAMBIT may benefit from using random initial points, and PRSS may benefit from a randomized rather than a depth-first traversal with each test.

There is a long history of using heuristics in artificial intelligence and planning [27, 23]. Our work is closely related and, in part, motivated by the success of heuristics in model checking [33, 6, 5, 26]. These *directed* model checking algorithms use sophisticated analysis of the input program model to generate heuristics for guided search. These techniques have been studied in an explicit-state or symbolic-state setting. GAMBIT scales to larger concurrent programs for which capturing the program state is impossible. It is not straightforward to translate heuristics from stateful search into a stateless setting. For instance, Yang and Dill [33] use Hamming distance to prioritize exploring states that are closer to error states. Such heuristics are not applicable in the context of GAMBIT. Preliminary work has been done in exploring heuristics in stateless search [11], but the genetic algorithms used in this work do not fare well when combined with partial-order reduction techniques nor do they provide the same soundness and progress guarantees. Finally, our use of user-provided heuristics is similar to *GuidePosts* in *SpotLight* [33].

Groce and Visser investigate guided model-checking for stateful search in PathFinder, and use several heuristics that are more applicable in a stateless search [13]. For example, they include a thread interleaving heuristic that actually favors *more* preemptions to increase the variation in thread schedules, which allowed their search to scale to more threads. We favor fewer preemptions because we find these executions are most useful to developers, as the preemptions often indicate the root cause of the bug. The state space is much smaller with fewer preemptions, as well, which makes it possible to find errors more quickly.

GAMBIT is also related to fuzzing-based techniques [7, 29, 28, 22, 15]. In contrast to fuzzing techniques, however, GAMBIT provides eventual soundness and progress guarantees.

7. Conclusions

Like most fuzzing techniques, GAMBIT allows testers to find concurrency bugs quickly with limited resources. At the same time, however, GAMBIT also provides testers with the soundness and progress guarantees associated with model checking. Although the best-first search used in GAMBIT incurs a significant space overhead when compared to the depth-first search used in typical model checkers, we introduce a compressed representation called an execution tree that reduces this space overhead significantly, cleans itself up as the search progresses, and lends itself well to a parallel search. Because GAMBIT stores *unvisited* rather than visited states, the space requirement grows with the number of prioritized executions, and reverts to a space-efficient depth-first search when priorities are equal. In cases where the space requirement is still too large, saving part of the space to disk is a simple addition.

We highlight several priority functions that find new happens-before executions more quickly, enable randomized search with progress guarantees, and allow the programmer to guide the search by specifying methods or variables likely to be involved in bugs. These heuristics help GAMBIT find more bugs in significantly less time than depth-first search with both dynamic partial-order reduction and preemption-bounding. GAMBIT provides a flexible framework for testing concurrent libraries so that developers can identify bugs quickly while preserving coverage and progress guarantees.

Acknowledgments

This work was supported by an NSF Graduate Research Fellowship and a Microsoft Research Ph.D. fellowship. We would like to thank Chris Dern for his valuable support using CHESS with the .NET concurrency libraries.

References

[1] Concurrency and Coordination Runtime — http://msdn.microsoft.com/en-us/library/bb648752.aspx.

[2] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, London, UK, 1981. Springer-Verlag.

[3] J. Duffy. A query language for data parallel programming: invited talk. In *DAMP*, page 50, 2007.

[4] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *Proceedings of the 29th international conference on Software Engineering (ICSE '07)*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

[5] S. Edelkamp and S. Jabbar. Large-scale directed model checking ltl. In *SPIN Workshop on Model Checking of Software*, pages 1–18. Springer, 2006.

[6] S. Edelkamp, A. L. Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *SPIN Workshop on Model Checking of Software*, pages 57–79. Springer-Verlag, 2001.

[7] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[8] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, pages 110–121, 2005.

[9] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. Foreword By-Pierre Wolper.

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

[11] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–280. Springer, 2002.

[12] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Computer Aided Verification (CAV '91)*, pages 332–342, 1992.

[13] A. Groce and W. Visser. Heuristic model checking for java programs. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, pages 242–245, London, UK, 2002. Springer-Verlag.

[14] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[15] P. Joshi, M. Naik, C.-S. Park, and K. Sen. Calfuzzer: An extensible active testing framework for concurrent programs. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09)*, pages 675–681, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] A first look at jsr 166: Concurrency utilities — http://today.java.net/pub/a/today/2004/03/01/jsr166.html.

[17] R. E. Korf, W. Zhang, I. Thayer, and H. Hohwald. Frontier search. *Journal of the ACM*, 52(5):715–748, 2005.

[18] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *OOPSLA*, page to appear, 2009.

[19] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Programming Language Design and Implementation (PLDI '07)*, pages 446–455, 2007.

[20] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *8th USENIX Symposium on Operating Systems Design and Implementation*, 2009.

[21] W. T. Overman. *Verification of concurrent systems: function and timing*. PhD thesis, 1981.

[22] S. Park, S. Lu, and Y. Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, pages 25–36, 2009.

[23] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[24] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *The Fifth International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[25] J. Reinders. *Intel Threading Building Blocks : Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007.

[26] N. Rungta and E. G. Mercer. Guided model checking for programs with polymorphism. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation (PEPM '09)*, pages 21–30, New York, NY, USA, 2009. ACM.

[27] S. J. Russell and P. Norvig. *Artificial intelligence : a modern approach*. Prentice Hall, 2nd edition, 2003.

[28] K. Sen. Effective random testing of concurrent programs. In *ASE*, pages 323–332, 2007.

[29] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, pages 11–21, 2008.

[30] Optimize managed code for multi-core machines — http://msdn.microsoft.com/en-us/library/bb648752.aspx.

[31] A. Valmari. Stubborn sets for reduced state space generation. In *The 10th International Conference on Applications and Theory of Petri Nets*, pages 491–515, London, UK, 1991. Springer-Verlag.

[32] W. Visser, K. Havelund, G. Brat, and S. Park. Java PathFinder - second generation of a Java model checker. In *Proceedings of Post-CAV Workshop on Advances in Verification*, July 2000.

[33] C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *DAC '98*, pages 599–604, 1998.