Stateless Model Checking with Data-Race Preemption Points

Ben Blum
Carnegie Mellon University, USA
bblum@cs.cmu.edu

Garth Gibson
Carnegie Mellon University, USA
garth@cs.cmu.edu

Abstract

Stateless model checking is a powerful technique for testing concurrent programs, but suffers from exponential state space explosion when the test input parameters are too large. Several reduction techniques can mitigate this explosion, but even after pruning equivalent interleavings, the state space size is often intractable. Most prior tools are limited to preempting only on synchronization APIs, which reduces the space further, but can miss unsynchronized thread communication bugs. Data race detection, another concurrency testing approach, focuses on suspicious memory access pairs during a single test execution. It avoids concerns of state space size, but may report races that do not lead to observable failures, which jeopardizes a user's willingness to use the analysis.

We present QUICKSAND, a new stateless model checking framework which manages the exploration of many state spaces using different preemption points. It uses state space estimation to prioritize jobs most likely to complete in a fixed CPU budget, and it incorporates data-race analysis to add new preemption points on the fly. Preempting threads during a data race's instructions can automatically classify the race as buggy or benign, and uncovers new bugs not reachable by prior model checkers. It also enables full verification of all possible schedules when every data race is verified as benign within the CPU budget. In our evaluation, QUICKSAND found 1.25x as many bugs and verified 4.3x as many tests compared to prior model checking approaches.

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

Keywords model checking, data races, verification

1. Introduction

Concurrency bugs are notoriously hard to find and reproduce because they appear only in specific thread interleavings, which arise at random during normal program execution. *Stateless model checking* [25] offers a method for finding such bugs, or verifying their absence, by forcing a program to execute each distinct interleaving, capturing this nondeterminism in a finite state space. Unfortunately, these state spaces explode exponentially in the size of the input program. Techniques such as Dynamic Partial Order Reduction [23] and Maximal Causality Reduction [29] expand the limits of feasible test completion, and search ordering strategies such as Iterative Context Bounding [40] encourage bugs to be found sooner in a given space should they exist.

However, all stateless model checkers to date are bound by a fixed set of *preemption points*: code locations that define the granularity at which threads interleave. For example, CHESS [41] by default preempts only on synchronization operations and library calls, which can miss lock-free shared memory races. On the other hand, SPIN [27] preempts threads before any shared memory access. Such fine granularity would automatically check each data race for the possibility of failure, but risks timing out before the state space can be completed. Some tools, such as CHESS and Inspect [58], can strike a middle ground by using compiler instrumentation to statically add preemption points on memory accesses. Nevertheless, choosing preemption points is a tradeoff between schedule coverage and feasibility of completion: even with state-of-the-art reduction techniques, fixing the degree of coverage in advance necessarily leaves some tests unaffordably large [15, 29].

We present QUICKSAND, a model checking framework for deciding at runtime which preemption points to test, according to which resulting state spaces are most likely to fit a prescribed CPU budget. It uses data-race analysis [47] to dynamically find new preemption points which expose bugs not reachable by preempting on API calls alone. When prior approaches would time out on large tests by trying several preemption points simultaneously, QUICKSAND identifies this pitfall in advance using state space size estimation [53], and instead tests smaller state spaces based on subsets of those preemption points. Often, testing these smaller state spaces can even find the same bugs sooner.

On the other hand, when the CPU budget is large enough to fully test all data-race preemption points, we prove that this constitutes a total verification of all possible thread

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
ACM. 978-1-4503-4444-9/16/11...$15.00
http://dx.doi.org/10.1145/2983990.2984036

schedules. To achieve the same level of verification, prior model checkers must decide in advance to preempt on every single memory access [27], which is computationally prohibitive for even moderately-sized tests. Our approach provides the best of both worlds: by estimating the size of the test on-the-fly, QUICKSAND can find bugs quickly in large tests *and* provide fast total verification for small ones.

We evaluate QUICKSAND by testing 157 student thread libraries and kernels from the undergraduate operating systems classes at Carnegie Mellon University, University of California at Berkeley, and University of Chicago. We find that data-race preemption points quickly expose many new bugs that prior model checkers could not find at all, and that they enable full verification of many more tests than before.

Our contributions are as follows:

1. *Iterative Deepening*, a new algorithm for combining data-race analysis with stateless model checking, and QUICKSAND, an open-source implementation;
2. A proof of convergence, showing that should it be possible in the given CPU budget, fully testing every discovered data-race preemption point is equivalent to testing all possible thread schedules (assuming a sequentially-consistent memory model);
3. A new tactic for eliminating one class of false-positive data race candidates, which cannot soundly be used in a single-pass analysis, but which we prove correct when used with Iterative Deepening;
4. A large evaluation in which QUICKSAND compares favourably to stand-alone data-race detection and stateless model checking approaches, finding new bugs that would be missed by either alone.

The rest of the paper is organized as follows. §2 reviews the background material, §3 and §4 discuss our design and implementation, §5 provides our proofs of soundness, §6 presents our evaluation, §7 discusses limitations and future work, §8 surveys the related work, and §9 concludes.

2. Background

We review the background in stateless model checking and data-race analysis using the example program in Figure 1.

2.1 Stateless Model Checking

Stateless model checking [25] is a testing technique for systematically exploring the possible thread interleavings of a concurrent program. A stateless model checker executes the program repeatedly, each time according to a new thread interleaving, until the state space (or the CPU budget) is exhausted. During each execution, it forces threads to execute serially, thereby confining the program's nondeterminism to controlled thread switches.

Rather than identifying suspicious conditions which may include false alarms, the approach of many static analyses [5, 20], stateless model checkers focus on concrete observed

```
                 int x = 0; mutex_t mx;
     Thread 1                       Thread 2
1    mutex_lock(&mx);
2    int tmp = x;
3                                   atomic_xadd(&x, 1);
4                                   yield();
5                                   atomic_xadd(&x, 1);
6    x = tmp + 1;
7    mutex_unlock(&mx);
8                                   assert(x >= 2);
```

Figure 1. Example program with a data-race bug. In this interleaving, the assertion on line 8 will fail. Two data-race preemptions are required to expose the bug: one just before thread 1's line 6, and one just before thread 2's line 8.

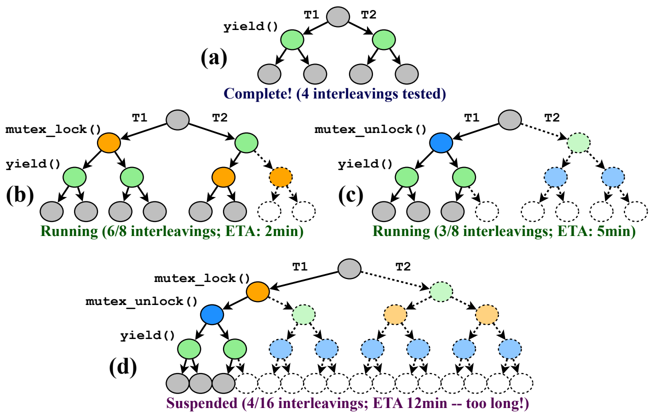

yield()
T1
T2
(a)
Complete! (4 interleavings tested)
mutex_lock()
yield()
(b)
Running (6/8 interleavings; ETA: 2min)
mutex_unlock()
yield()
(c)
Running (3/8 interleavings; ETA: 5min)
mutex_lock()
mutex_unlock()
yield()
(d)
Suspended (4/16 interleavings; ETA 12min -- too long!)

Figure 2. Iterative Deepening example. The minimal state space (a) includes only voluntary thread switches, such as `yield()`. Multiple further tests can be run: preempting on calls to `mutex_lock` alone (b), `mutex_unlock` alone (c), or both together (d). Each option increases the state space size unpredictably, so multiple state spaces should be tested in parallel. Estimation techniques [53] inform which state spaces to prioritize.

failures such as assertions, deadlocks, segfaults, and use-after-free. For example, data races do not always lead to failures, but represent suspicious violations of common locking discipline. Although C++ defines all data races as undefined behaviour [2], this work focuses on reporting races to the user only when accompanied by direct evidence of a failure.

The checker defines the granularity of thread interleavings by the *preemption points* it uses to switch threads. Most model checkers [41, 51] choose synchronization and thread library API boundaries for these points; in our example program, these would be lines 1, 4, and 7. Figure 2 shows several possible resulting state spaces. The approach of prior work is to enable all preemption points simultaneously, i.e., to test only the state space denoted (d).

To mitigate the exponential explosion, Dynamic Partial Order Reduction [23] identifies equivalent execution sequences according to Mazurkiewicz trace theory [38], and tests at least one execution from each equivalence class. Intuitively, if two thread transitions between preemption points do not conflict on any shared resource access, reordering them produces an equivalent interleaving, i.e., the same program behaviour. Iterative Context Bounding [40], another popular technique, heuristically reorders the search to prioritize interleavings with fewer preemptions first, according to the insight that most bugs require few preemptions to uncover. Nevertheless, state spaces are still exponentially-sized in the number of conflicting transitions.

This motivates *Iterative Deepening*, our new technique for heuristically adjusting the preemption points at runtime. Rather than committing to one state space with every available preemption point enabled, we will search among different *subsets* of the points. Hence, we will test all the state spaces shown in Figure 2 in parallel, and decide on-the-fly whether to pursue each test, or to defer it in favour of others.

Recent advances. Like many prior checkers [31, 41, 51, 58], ours implements DPOR for sequentially-consistent hardware. In the worst case, these tools may all suffer false negatives as they miss weak-memory-only bugs. Recently, Zhang et al. [59] introduced a new formalism with which DPOR can control weak memory nondeterminism, such as reordering store buffers. Iterative Deepening could use this new technique, provided a model checker which implements such reorderings (not yet available to us).

Maximal Causality Reduction (MCR) [29], a reduction algorithm which may replace DPOR, has also recently shown promising performance improvements over prior model checkers. We expect Iterative Deepening to be compatible with MCR, and look forward to evaluating the combination when an MCR implementation becomes widely available.

2.2 Data Race Detection

Data race analysis [47] identifies pairs of unsynchronized memory accesses between threads. Two instructions are said to race if they both access the same memory address, at least one is a write, the threads do not hold the same lock, and no synchronization enforces an order on the thread transitions (the *Happens-Before* relation). In Figure 1, lines 3 and 5 each race with 2 and 6, and line 6 races with 8.

A data race analysis may be either *static* (inspecting source code) [20] or *dynamic* (tracking individual accesses arising at run-time) [49]. This paper focuses exclusively on dynamic analysis, so although our example refers to numbered source lines for ease of explanation, in practice we are actually classifying the individual memory access events corresponding to those lines during execution.

Though state-of-the-art model checkers preempt only on synchronization events, many serious concurrency bugs are caused by data races leading to corrupted shared state. Figure 1's buggy interleaving is possible only with *data-race preemption points*: preempting just before an instruction identified as part of a data race. None of the state spaces in

```
int x = 0; bool y = false; mutex_t mx;
Thread 1                  Thread 2
x++; // A1
mutex_lock(&mx);
mutex_unlock(&mx);
                          mutex_lock(&mx);
                          mutex_unlock(&mx);
                          x++; // A2
```

(a) True potential data race.

```
x++; // B1
mutex_lock(&mx);
y = true;
mutex_unlock(&mx);
                          mutex_lock(&mx);
                          bool tmp = y;
                          mutex_unlock(&mx);
                          if (tmp) x++; // B2
```

(b) No data race in any interleaving.

Figure 3. Data-race analyses may be prone to either *false negatives* or *false positives*. Applying HB to program (a) will miss the potential race possible between A1/A2 in an alternate interleaving, while using Limited HB on (b) will produce a false alarm on B1/B2.

Figure 2 contain this interleaving, as none of the mutex/yield preemptions split lines 2 and 6 across different transitions.

Variants of Happens-Before. Most prior work focuses on *Happens-Before* (HB) [22, 34, 46] as the order relation between accesses. [55] and [44] identify a problem with this approach: it cannot identify access pairs separated by an unrelated lock operation which could race in an alternate interleaving. Figure 3(a) shows a contrived example program in which HB masks the potential race. We call such unreported access pairs *false negatives*. However, consider the similar program in Figure 3(b), in which the access pair ceases to exist in the alternate interleaving. O'Callahan et al. [44] introduced the *Limited HB* relation, which will report such potential races by considering only blocking operations like `cond_wait` to enforce the order. Limited HB will report all potential races, avoiding many false negatives [49], but at the cost of necessarily reporting some such *false positives*.

Finally, the *Causally-Precedes* relation [55] extends HB to additionally report a subset of potential races while soundly avoiding false positives. It tracks conflicting accesses in intervening critical sections to determine whether lock events are unrelated to a potential race. Causally-Precedes will identify the potential race in Figure 3(a), as the two critical sections do not conflict, although it can still miss true potential races in other cases.

Being dynamic analyses, both HB and Limited HB may suffer false negatives when a racy access pair is not executed at all in a specific interleaving. Limited HB offers the advantage of identifying a potential race as long as the access pair is observed under any interleaving, rather than requiring the accesses to be adjacent in time, as HB would. While standalone data-race analyses must avoid inundating the user with false alarms [20], we incorporate data-race analysis in an internal feedback loop, using model checking to automatically test each potential race and report only directly observed failures to the user. Hence, we can accept some overhead from Limited HB's false positives for the sake of finding data-race candidates more quickly. In §6 we will evaluate how HB and Limited HB each influence QUICKSAND's bug-finding and verification speed.

Philosophy of bugs. While there is a vast body of work on how to detect data races to begin with, judging data races once found is a matter of philosophical debate unto itself. Some recent tools classify races depending on how they impact program behaviour [31, 42], overlooking *benign* races in search of more dangerous ones. [20] acknowledges that a program may have too many races for a user to worry about, so bug reports must be prioritized by severity of effects. However, other prior work argues that data races are always bugs [9, 10], largely due to the possibility of compiler or hardware reordering of racy accesses. We take the former camp: we consider a data race a bug only when it results in a visible failure state (e.g., crash or deadlock)[1]. We bypass concerns of compiler reordering by checking programs at the executable level; for a study of hardware reordering in the context of DPOR, we refer the reader to [59].

2.3 Terminology

For the rest of the paper, we will abbreviate *preemption point* (PP), *happens-before* (HB), *model checking* (MC), *single-state-space model checking* (SSS-MC), *Dynamic Partial Order Reduction* (DPOR), *Iterative Context Bounding* (ICB), and *state space estimate* (ETA).

SSS-MC indicates the approach of prior tools: the set of PPs is fixed in advance, and the tool commits to testing every interleaving available with those PPs. Many techniques can skip equivalent interleavings or order the search to uncover bugs faster [13, 23, 26, 40, 56], but new PPs cannot be added, nor ineffective ones removed, by any dynamic analysis.

We distinguish between data-race *candidates* and data-race *bugs*. We refer to racing (or potentially-racing) memory access pairs as *data race candidates*. Should preempting during such accesses lead to an observable failure, then we report a *data-race bug*. Otherwise, if the access pair can be reordered, but does not produce a failure under any interleaving, it is a *benign data race* (with respect to the test input). If they cannot be reordered at all, due to some other communication such as in Figure 3(b), it is a *false positive*.

We also identify the *minimal* and *maximal state space* for each test. The *minimal state space* includes only thread switches arising from no preemptions (Figure 2(a)). The *maximal state space* is the one tested by SSS-MC: all statically-available PPs are enabled (Figure 2(d)).

[1] C++ declares any race between two non-atomic locations to be undefined behaviour [11]. From a C++ perspective, we assume all concurrent accesses are implemented by `std::atomic` loads and stores.

Algorithm 1: Naïve Iterative Deepening method

```
Input: $j$, the currently-running job
Input: $\mathcal{A}$, the set of all known preemption points
1 if $\exists p \in \mathcal{A}.p \notin PPSet(j)$ then
2     return NewJob($\mathcal{A}$) // New maximal state space
3 else
4     return $j$ // $j$ is still maximal
5 end
```

3. Design

Named after the analogous technique in chess artificial intelligence [33], Iterative Deepening makes progressively deeper searches of the state space until the CPU budget is exhausted. In this context, the depth is the number of PPs used. Hence, QUICKSAND schedules multiple MC instances in parallel to test many different subsets of the available PPs, We refer to each unique set of PPs as a *job*.

Note that Iterative Deepening is a *wrapper* algorithm around stateless MC. A MC tool is still used to test each state space, and other reduction techniques are still applicable. Moreover, because Iterative Deepening treats the set of preemption points as mutable, it can add new preemption points reactively based on any runtime analysis. We focus on run-time data-race detection [19, 44, 49] as the mechanism for finding new preemption candidates.

3.1 Changing State Spaces

To introduce the Iterative Deepening algorithm, we first show a simple approach for handling new PPs in the absence of any CPU budget restriction.

Naïve approach. Given unlimited CPU time for testing, we would always switch to the new maximal state space whenever adding a new PP. The maximal state space is guaranteed to subsume all execution sequences reachable in any subset state space, so considering any incomplete subset of the known PPs would be duplicate work. Algorithm 1 demonstrates this naïve approach. It is seeded with the set of all statically-known synchronization API PPs, and invoked whenever a new data-race candidate is found. Our proofs in §5, being concerned with the verification guarantee provided when QUICKSAND completes within the CPU budget, are based on this simple version of Iterative Deepening.

Prioritizing smaller jobs. However, in tests where full verification is not feasible, focusing on the maximal state space alone is likely to be fruitless. Hence, we prioritize all subset jobs based on number of PPs, ETA, and whether they include data-race PPs, We rely on state-space estimation [53] to predict which jobs are likely to complete within a reasonable time, before actually testing a large fraction of interleavings for each. The overall goal is to decide automatically when to defer testing a state space, so an inexpert user can provide only their total CPU budget as a test parameter, and to enable completing appropriately-sized jobs within that budget. We seek to maximize completed state spaces, as each one serves as a guarantee that all interleavings possible with its PPs were tested. The next three subsections will show how we schedule these smaller jobs based on their PP sets and ETAs.

3.2 Initial PP Configuration

Iterative Deepening must be seeded with a set of initial state spaces, which can be any number of subsets of the statically-available PPs SSS-MC would use. For completeness (§5.1), the maximal state space must be included among these.

For testing user-space code, we begin with the four PP sets from Figure 2: $\{yield\}$, $\{yield, lock\}$, $\{yield, unlock\}$, and $\{yield, lock, unlock\}$, By extension, these also introduce PPs on any other primitives which use internal locks, such as condvars or semaphores. Preempting on voluntary switches such as `yield` is always necessary to maintain the invariant that only one thread runs between consecutive PPs.

For kernel-level testing, we consider interrupt-disabling analogous to locking, so we also preempt just before a disable-interrupt opcode (`cli`) and just after interrupts are re-enabled (`sti`)[2]. QUICKSAND is configured to begin with $\{yield\}$, $\{yield, lock\}$, $\{yield, unlock\}$, $\{yield, cli\}$, $\{yield, sti\}$, and $\{yield, lock, unlock, cli, sti\}$. As a heuristic, we don't test every intermediate subset such as $\{lock, sti\}$, which could potentially be improved in future work (§7).

3.3 Data-Race Preemption Points

During stateless MC, runtime data-race detection may find data-race candidates that we wish to investigate further. Because data races indicate access pairs that can interleave at instruction granularity, it is logical to re-execute the test and issue preemptions just before those instructions to test alternate thread interleavings [31, 48].

With Iterative Deepening, this is a simple matter of creating a new state space with an additional PP enabled on the racing instructions by each thread, as shown in Algorithm 2. We call these *data-race PPs*. Note that even though a data race may involve two different instructions, α and β, we add new state spaces with only one new PP at a time. Rather than adding a single large state space, i.e., AB = PPSet(j_0) $\cup\, \alpha \cup \beta$, we prefer to add multiple smaller jobs which have a higher chance of completing in time, i.e., A = PPSet(j_0) $\cup\, \alpha$ and B = PPSet(j_0) $\cup\, \beta$. If A and B are bug-free, they will in turn add AB later. The condition on line 1 ensures that we avoid duplicating any state spaces with multiple data-race PPs; for example, AB is reachable by multiple paths through its different subsets, but should be added only once.

[2] During data-race detection, `cli/sti` are treated as a single global lock. Some kernels disable preemption without disabling interrupts, which can be communicated to the MC using manual annotations. This also assumes uniprocessor scheduling; for SMP kernels, replace `cli/sti` with spinlocks.

Algorithm 2: Adding new jobs with data-race PPs.

```
Input: $j_0$, the currently-running job
Input: $\mathcal{J}$, the set of all existing (or completed) jobs
Input: $\alpha$, an instruction reported by the MC as part of a
       racing access pair
1 if $\forall j \in \mathcal{J}$, $PPSet(j_0) \cup \alpha \not\subseteq PPSet(j)$ then
2     AddNewJob(PPSet($j_0$) $\cup\, \alpha$, HeuristicPriority($\alpha$))
3 end
4 if $\forall j \in \mathcal{J}$, $PPSet(j) \neq \{yield, \alpha\}$ then
5     AddNewJob($\{yield, \alpha\}$, HeuristicPriority($\alpha$))
6 end
```

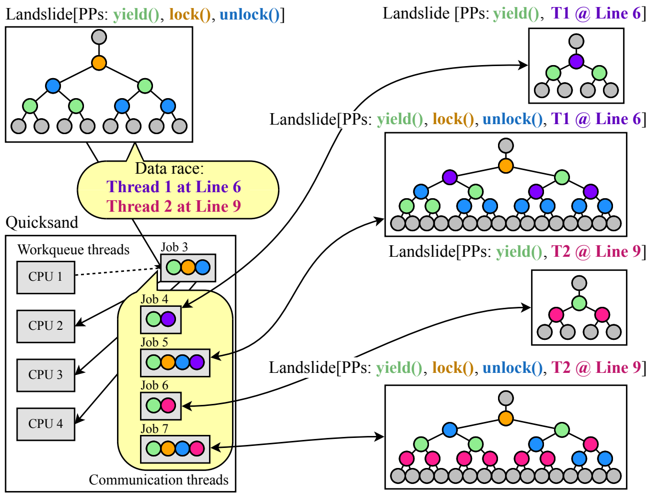

Landslide[PPs: yield(), lock(), unlock()]
Landslide [PPs: yield(), T1 @ Line 6]
Landslide[PPs: yield(), lock(), unlock(), T1 @ Line 6]
Data race:
Thread 1 at Line 6
Thread 2 at Line 9
Quicksand
Workqueue threads
CPU 1
CPU 2
CPU 3
CPU 4
Job 3
Job 4
Job 5
Job 6
Job 7
Communication threads
Landslide[PPs: yield(), T2 @ Line 9]
Landslide[PPs: yield(), lock(), unlock(), T2 @ Line 9]

Figure 4. QUICKSAND manages the exploration of multiple state spaces, communicating with each MC instance to receive ETAs, data race candidates, and bug reports. When an access pair is reported as a data race candidate, we generate a new PP for each access, and add new jobs corresponding to different combinations of those with the existing PPs.

Furthermore, we do not always strictly increase the number of PPs when we find a new data race. For each instruction involved in a data race, QUICKSAND adds two new jobs: a "small" job to preempt on that instruction only (line 5), and a "big" job to preempt on that instruction as well as each PP used by the reporting job (line 2). Hence, each *pair* of racing accesses will spawn four new jobs, as shown in Figure 4. The rationale of spawning multiple jobs is that we don't know in advance which will be most fruitful: while the big job risks not completing in time, the small job risks missing the data race entirely if the original PPs were required to expose it. In practice, we observed some bugs found quickly by these small jobs, and other bugs missed by the small jobs found eventually by the big jobs. This phenomenon motivates Iterative Deepening to prioritize jobs at run-time.

The new state spaces may expose a failure, in which case we report a data-race bug, or complete successfully, indicating a benign or false-positive data race. They may also uncover a new data-race candidate entirely, in which case we may iteratively advance to a superset state space containing PPs for both racing access pairs. Being constrained by a CPU budget, we may time out before completing a data race's associated state space, in which case we report a potential false positive that the user must handle (§7).

3.4 Choosing the Best Job

With a limited CPU budget, we must avoid running tests that are likely to be fruitless. Hence, we separate the available PP sets into a set of *suspended* jobs (partially-explored state spaces with high ETAs), and a set of *pending* jobs (untested ones with unknown ETAs). When the MC reports an ETA too high for some job, we compare with other pending and suspended jobs to find another one more likely to complete in time. Our method, listed in Algorithm 3, is the heart of Iterative Deepening. Its main feature is understanding that if PPSet(j_1) $\subset$ PPSet(j_2), and j_1 is suspended, then j_2's state space is guaranteed to be strictly larger, so j_2 will take at least as long. Hence we should avoid testing j_2 unless j_1 is later resumed and its ETA improves after further execution. Similarly, whenever a job finds a bug, we cancel all pending superset jobs, as they might find only the same bug.

We also account for the inherent inaccuracy of ETA estimates. Line 1 heuristically scales up the time remaining to avoid suspending jobs too aggressively in case their ETAs are actually overestimated. Lines 12-15 account for the possibility that among two suspended jobs, PPSet(j_1) $\subset$ PPSet(j_2) but ETA(j_1) $>$ ETA(j_2). This can arise because estimates tend to get more accurate over time, and j_1 perhaps ran much longer before suspending. We heuristically assume the smaller job's ETA is more accurate to avoid repeatedly resuming larger jobs briefly while their ETAs only become worse (it lets us avoid thrashing in QUICKSAND).

3.5 Heuristics

Algorithm 3 allows heuristically scaling a job's ETA when comparing to the time budget, to express how pessimistic we are about the estimate's accuracy. We use a scaling factor defaulting to 2 based on the results in [53]. We also include a heuristic to never suspend jobs before they pass a certain threshold of interleavings tested, with a default of 32, so that their ETAs have some time to stabilize.

We classify data-race candidates as *single-order* or *both-order* [31] based on whether the MC observed the racing instructions ordered one or both ways in the original state space. Single-order candidates are more likely to be false positives (§2.2), although preempting during the access itself is necessary to say for sure. Hence, we add PPs for both types of candiates, and heuristically prioritize jobs with both-order data-race PPs (indicated by the HeuristicPriority(α) call in Algorithm 2). For single-order races, we do not initially add a PP for the later access at all: if preempting on the first access can reorder the race, it will be upgraded to both-order in the new state space, and we will add the second PP then.

Algorithm 3: Suspending exploration of a state space in favour of a potentially smaller one.

```
Input: $j_0$, the currently-running job
Input: $\mathcal{P}$, the list of pending jobs, sorted by decreasing
       heuristic priority
Input: $\mathcal{S}$, the list of already-suspended jobs, sorted by
       increasing ETA
Input: $T$, the remaining time in the CPU budget
if $ETA(j_0) < HeuristicETAFactor \times T$ then
    return $j_0$ // Common case: job is expected to finish.
end
foreach job $j_P \in \mathcal{P}$ do
    // Don't run a pending job if a subset of it is already
    suspended; its ETA would be at least as bad.
    if $\forall j_S \in \mathcal{S}, PPSet(j_S) \not\subset PPSet(j_P)$ then
        return $j_P$
    end
end
foreach job $j_S \in \mathcal{S}$ do
    if $PPSet(j_0) \not\subset PPSet(j_S) \wedge ETA(j_0) > ETA(j_S)$
    then
        // If a subset of $j_S$ is also suspended, don't run
        the larger one first.
        if $\forall j_{S2} \in \mathcal{S}, PPSet(j_{S2}) \not\subset PPSet(j_S)$ then
            return $j_S$
        end
    end
end
return $j_0$ // ETA($j_0$) was bad, but no other $j$ was better.
```

4. Implementation

4.1 Landslide

We chose LANDSLIDE [7] as our MC tool due to its ability to trace execution at the granularity of individual instructions and memory accesses, which dynamic data-race detection requires. LANDSLIDE implements DPOR [23], state space estimation [53], the DJIT+ HB data-race analysis [22, 46], and a hybrid lockset/Limited HB data-race analysis [44]. It can optionally replace DPOR with ICB [40], for which it uses Bounded Partial Order Reduction [14] for a similar reduction, though QUICKSAND does not employ this feature (§7). It avoids state space cycles (e.g. spin loops) with a heuristic similar to Fair-Bounded Search [14]. Its bug-detection metrics are assertion failure, deadlock, segfault, use-after-free [43], and (heuristically) infinite loops.

LANDSLIDE can test both userspace and kernel code (although it is limited to timer nondeterminism), and runs programs in a full-system hardware simulator [36]. The simulator allows LANDSLIDE to track memory accesses and check for heap errors on uninstrumented binaries, although for performance, a similar MC under QUICKSAND could use compiler instrumentation instead. It also provides a convenient backtracking mechanism to avoid the need to re-simulate common execution prefixes among many interleavings.

Restricting PPs with stack trace predicates. When testing a particular module in a large codebase, the user is likely to be uninterested in PPs arising from other modules. Rather than preempting indiscriminately on any synchronization call, regardless of the call-site, prior work introduced Preemption Sealing [4] for identifying which call-sites matter. LANDSLIDE provides this feature through a configuration command, `within_function`. Before inserting a PP, LANDSLIDE requires at least one argument to `within_function` to appear in the current thread's stack trace. The `without_function` directive is the dual of `within_function`, indicating a blacklist. We use these to restrict the scope of some tests in our evaluation (§6.1).

Data races in lock implementations. Data race tools in prior work [31, 49] recognize the implementations of synchronization primitives to avoid spuriously flagging memory accesses that implement them. Assuming that the lock implementation is already correct enables more productive data-race analysis on the rest of the codebase, while the locks themselves can be verified separately [50]. We included a mutex test in our evaluation to showcase QUICKSAND's ability to verify synchronization primitives with data-race PPs (§6.1). To support this test, we extended LANDSLIDE to optionally make its data race analysis consider accesses from `mutex_lock()` and `mutex_unlock()`.

4.2 Quicksand

QUICKSAND is an independent program that wraps the execution of several LANDSLIDE MC instances. The implementation is roughly 3000 lines of C. The interface with the MC has two parts. First, when starting each job, QUICKSAND creates a configuration file declaring which PPs to use, plus other MC-specific options such as our modifications to LANDSLIDE for testing mutexes. Then, a dedicated QUICKSAND thread communicates with the MC process via message-passing. The MC messages after testing each interleaving to report updated progress and ETA and whenever a new data-race candidate or bug is found. QUICKSAND in turn replies whether to resume/suspend (due to too high ETA) or quit (due to timeout). We suspend jobs simply by making the MC wait on a message-passing reply. Should QUICKSAND later re-schedule a suspended job, it sends a message to continue, resuming the job where it left off.

5. Soundness

In this section we present two theorems concerning Iterative Deepening's correctness. Our full proofs, available at [8], discuss our assumptions explicitly and include more formal definitions and structure.

These proofs are built on a DPOR algorithm definition which assumes sequentially-consistent memory hardware, as discussed in §2.1. We also assume the Limited HB definition for the data-race analysis, as discussed in §2.2.

5.1 Convergence to Total Verification

Although Iterative Deepening's main purpose is to heuristically choose the most effective PP subsets to test when the maximal state space is too large, some tests may be small enough that even their maximal state spaces could be completed in time. For such tests, preempting on every shared memory access [27, 58] would provide a total verification of all possible thread schedules, if it could complete in time. In this section, we show that Iterative Deepening provides a verification of the same strength if it completes the state spaces associated with every discovered data-race PP. A proof sketch of the contrapositive statement follows.

Theorem 1 (Convergence). *If a bug can be exposed by any thread interleaving possible by preempting on all instructions during a specific test, Iterative Deepening will eventually test an equivalent interleaving which exposes the same bug.*

Proof Sketch. The proof has two parts: first, we show that preempting on data-racing instructions and synchronization API boundaries suffices to test all possible program behaviour; second, we show that Iterative Deepening will eventually detect all such data races. Given a PP p, let $\mathsf{next}(p)$ denote the next transition after p executed by the thread which ran immediately before p, let $\mathsf{instr}(p)$ denote the first instruction of $\mathsf{next}(p)$, and let $\mathsf{others}(p)$ denote the transitions by other threads between p and $\mathsf{next}(p)$.

Lemma 1 (Equivalence of non-data-race PPs)**.** *For any thread interleaving possible by preempting on any instruction, there exists an equivalent interleaving which uses only data-race PPs and synchronization API PPs.*

Let p be the first PP in the given interleaving such that $\mathsf{instr}(p)$ is not a data race with $\mathsf{others}(p)$ nor is a synchronization API boundary. Because $\mathsf{instr}(p)$ is not a synchronization boundary, no lock can be held during $\mathsf{others}(p)$ that was also held by the first thread across p. Hence, because $\mathsf{instr}(p)$ is not a data race, it cannot be a shared memory conflict with $\mathsf{others}(p)$ at all. Let i be the first instruction among $\mathsf{next}(p)$ which is such a conflict, or a synchronization boundary. If i is a shared memory conflict, it must be a data race, for the same reasoning as above. We modify the input interleaving by reordering $\mathsf{instr}(p)$ until i, not including i, to before $\mathsf{others}(p)$. By the soundness of DPOR [23], this is equivalent to the input interleaving. In other words, we have transformed p into p' such that $\mathsf{next}(p') = i$, which is a data race or synchronization boundary. All PPs in the input trace can be inductively converted in the same manner.

Definition 1 (Reachability)**.** *A data race candidate, and its associated PPs, are reachable if it will be identified by a MC configured to preempt only on already-reachable PPs.*

Initially, the statically-available synchronization API PPs are reachable. Reachability of data-race PPs is transitive.

Lemma 2 (Saturation of data-race PPs)**.** *Given any interleaving comprising only data-race PPs and synchronization API PPs, all involved PPs are reachable.*

We induct on the PPs according to the order of their preemptions. Given that the interleaving prefix preceding some PP p is reachable, we require that either p is reachable, or a new data race among others(p) will be newly reachable. The latter condition suffices because in a finitely-sized codebase, there must be finitely many unique racing instruction pairs.

First, we must "coalesce" away p, as well as any other not-yet-reachable PPs in others(p). Consider the alternate interleaving in which the first thread executes past p until the first already-reachable PP, then the other threads among others(p) execute the same way. This interleaving's PPs are all reachable, so a state space $\mathcal{S}$ containing it will be tested.

If p is a not-yet-reachable data-race PP, it must be possible for some other thread to execute a data-racing instruction with instr(p). If this conflict was observed in the state space containing our coalesced interleaving, we have reached p. Otherwise, we appeal to the soundness of DPOR: If a program behaviour is possible by interleaving threads at the boundaries of the given transitions, it will be tested in the containing state space. By contrapositive, to expose this behaviour, one or more preemptions must occur in the middle of some transition, rather than at the boundaries.

We now show by contradiction there cannot be *multiple* data-race PPs which must all be enabled before either data race can be identified. Assume there does not exist a single transition $t_1 \in \mathcal{S}$ which alone can be split into $\{t_1', t_1''\}$ by a PP q, such that another thread's concurrent transition t_2 conflicts with t_1''. By the soundness of DPOR, because all t_2s are independent with t_1'', $\mathcal{S} \equiv \mathcal{S} \cup q$. Replacing $\mathcal{S}$ with $\mathcal{S} \cup q$ in the above assumption shows that no *pair* of new qs would expose new program behaviour, and inductively, no set of qs of any size, which contradicts the previous paragraph.

Hence, a single new not-yet-reachable data race is reachable in $\mathcal{S}$. Hence p will be reached.

To conclude, for any possible interleaving, Lemma 1 provides an equivalent one with only data-race and synchronization PPs, and Lemma 2 proves all involved PPs are reachable. Hence, Iterative Deepening will eventually test a state space containing the equivalent buggy interleaving. □

5.2 Suppressing "Malloc-Recycle" False Positives

We identify a particular class of false positive data-race candidates under Limited HB in which the associated memory

```
        struct x { int foo; int baz; } *x;
        struct y { int bar; } *y;

    Thread 1            Thread 2
1   x->foo = ...;
2   free(x);
3                       // x's memory recycled
4                       y = malloc(sizeof *y);
5                       // ...initialize...
6                       publish(y);
7                       y->bar = ...;
```

Figure 5. A common execution pattern with `malloc()` that produces false positive data race candidates.

```
    Thread 1            Thread 2
1   publish(x);
2   x->foo = ...;
3   free(x);
4                       x2 = get_published_x();
5                       // x's memory recycled
6                       y = malloc(sizeof *y);
7                       x2->foo = ...;
```

Figure 6. If a single-pass Limited HB analysis discarded candidates matching the malloc-recycle pattern, it would miss the bug in this adversarial program.

was recycled by re-allocation between the two accesses. Figure 5 shows a common code pattern and interleaving which can expose such behaviour. If the `malloc` on line 4 returns the same address passed to `free` on line 2, then lines 1 and 7 will be flagged as a potential data race. We call this a *malloc-recycle data race candidiate*. To the human eye, this is obviously a false positive: reordering lines 4-7 before lines 1-2 will change `malloc`'s return value, causing `x` and `y` to no longer collide. Here, Thread 2's logic usually corresponds to an initialization pattern [47], but for generality we have added an arbitrary `publish` action on line 6.

When limited to a single test execution, suppressing any data race candidate matching this pattern is unsound. Consider the more unusual program in Figure 6, in which the memory is recycled the same way, but the racing access's address is not tied to `malloc`'s return value. Here, reordering lines 6-7 before line 3 will allow `x` and `x2` to race. Such collisions could be avoided with a hacked allocator which never recycles memory, but this could unacceptably impact performance in `malloc`-heavy tests.

Fortunately, when data-race detection is combined with DPOR and Iterative Deepening, pruning all malloc-recycle candidates is sound, even considering adversarial programs such as Figure 6. This makes it unnecessary to verify such candidates by actively adding more preemptions, achieving a potentially combinatorial reduction in how many state spaces we generate. We provide a proof sketch below.

Theorem 2 (Soundness of eliminating malloc-recycle candidates)**.** *If a malloc-recycle candidate is not a false positive, DPOR will test an alternate thread interleaving in which the accesses can race without fitting the malloc-recycle pattern.*

Proof Sketch. Any such program must contain an access a_1 by one thread T1, followed by a `free` and a `malloc` possibly by either thread, followed by an access a_2 by the other thread T2. Without loss of generality, we say that T1 performs the `free` and T2 the subsequent `malloc`. We also assume the only way for the program to get pointers to heap memory is through `malloc`; hence, there must also be some "publish" action p by T1 which communicates the address to T2. Because this is a true potential data race, p must occur before a_1, as a_2 cannot be reordered before p.

We require that a PP will be identified during T1 between p and a_1. The publish action must involve some thread communication, whether through a shared data structure or message-passing API. If locking or message-passing is used, our set of hard-coded PPs suffices to provide a PP. Otherwise, p (and the corresponding read by T2) will be a potential data race, although that may itself be a malloc-recycle candidate. In this case we use induction on the pointer chain leading to the shared address containing p: in the base case, p is communicated via global data or message-passing, and in the inductive step, DPOR will reorder threads sufficiently to identify the PP on p. Hence there will be a PP between p and a_1 no matter the mode of communication.

With this PP, DPOR will reorder a_2 before a_1, while not changing a_2's location. As T2's `malloc` now occurs before T1's `free`, it will allocate different memory. Hence a_1 and a_2 can race without fitting the malloc-recycle pattern. $\square$

We implemented a simple check in LANDSLIDE to recognize the malloc-recycle pattern: each heap allocation is given a unique ID, and when evaluating whether two heap accesses can race, the IDs of their containing blocks must match. Note that this proof does not require PPs on `malloc`'s internal lock, which is an ideal candidate to ignore via `without_function` (§4.1) to reduce state space size.

This class of false positive is unique to heap-allocated memory among all ways threads could communicate. By contrast, global memory has unlimited lifetime, and message-passing primitives enforce an ordering which precludes the race. Finally, note that as long as concurrent `malloc` is implemented with an internal lock, these false positives are of concern only under a Limited HB analysis (§2.2). Nevertheless, Theorem 1's need for the Limited HB definition justifies our choice of it over full HB.

6. Evaluation

Although QUICKSAND presents Iterative Deepening and data-race PPs as interconnected techniques, they each could be employed alone as well. For example, a single-state-space tool could use data-race PPs during immediately subsequent interleavings, changing the state space on the fly. Likewise, a message-passing-only tool could use Iterative Deepening despite a concurrency model lacking data races, to promote completing subset PP sets for large tests. Hence, we also evaluated each technique individually, though many of our experiments compare QUICKSAND to the state-of-the-art as a whole. Our evaluation answers the following questions:

1. Does QUICKSAND improve upon state-of-the-art MC?
   (a) Do data-race PPs expose new bugs that couldn't be found with SSS-MC's fixed-PP-set approach?
   (b) Does QUICKSAND find bugs faster in subset state spaces, even without data-race PPs?
   (c) Does QUICKSAND provide more full verifications of bug-free programs more quickly?
   (d) How does the choice between HB and Limited HB affect bug-finding and verification performance?
2. Does MC improve the accuracy of data-race detection?
   (a) Does QUICKSAND avoid false positives compared to a single-execution Limited HB data race analysis?
   (b) Does QUICKSAND find data-race bugs that would be false-negatives during a single-execution analysis?

6.1 Test Suite

Our test suite consists of 79 "P2" student thread libraries, from Carnegie Mellon's 15-410 operating systems class, and 78 "Pintos" student kernels, from Berkeley's CS162 and University of Chicago's CMSC 23000 operating systems classes. The P2 project comprises `thr_create()`, `thr_exit()`, `thr_join()`, mutexes, condition variables, semaphores, and reader/writer locks; all implemented from scratch in userspace with a UNIX-like system call interface [17, 18]. The Pintos kernel project comprises priority scheduling, `sleep()`, and user-space process management (`wait()` and `exit()`) using provided mutex, context-switch, and virtual memory implementations [45]. Both projects are quite complex: the P2s average 1807 lines of code, and the Pintoses average 718 lines, for a total of 198,772 lines tested for this paper.

We chose P2s and Pintoses for our test suite because of the relative ease of generating hundreds of unique state spaces, varied in size and correctness, and with a diverse set of bug types[3]. We believe that merely finding a small handful of new real-world bugs is largely anecdotal, and that our test suite's size allows for a more statistically significant comparison among MC and data-race testing strategies.

We tested P2s with 6 multithreaded programs: `mx_test`, for locking algorithm correctness, `join_test`, a test of thread lifecycle, `bcast_test` and `signal_test` for condition variables, `sem_test` for semaphores, and `rwlock_test`

[3] Many of the codebases exhibited *deterministic* bugs (i.e., encountered on the first interleaving tested). We fixed these by hand, before running tests, to ensure that every bug in our study required meaningful work by the MC.

for r/w locks. For `mx_test`, `sem_test`, and `signal_test`, we used the `without_function` command to blacklist `thr_create`, `thr_exit`, and `thr_join`, and for `mx_test` we enabled LANDSLIDE's mutex-testing option (see §4.1). We tested Pintoses with 3 programs from the class's provided test suite: `sched_test`, a test of the kernel scheduling algorithm, `alarm_test`, for the timer sleep routine, and `wait_test`, for process lifecycle system calls[4]. The source code of all 9 test cases is available at `https://github.com/bblum/oopsla-dataset`. For all tests, we blacklisted PPs on `malloc`'s internal lock using LANDSLIDE's `without_function` command (§4.1). In total, our evaluation comprises 629 unique tests (i.e., pairs of a test program and a Pintos or P2), at least 181 of which will expose bugs under one or more MC trials.

6.2 Experimental Setup

To evaluate the benefits of data-race PPs and Iterative Deepening separately, we ran the test suite under QUICKSAND in three different configurations, each of which was given a 1-hour budget and 10 CPUs for each test.

- **QS-Limited-HB**: QUICKSAND using Limited HB for its data-race analysis,
- **QS-Pure-HB**: QUICKSAND using pure HB instead, and
- **QS-Sync-Only**: QUICKSAND with PPs still seeded as described in §3.2, but never adding new PPs from reported data-race candidates.

We represented the MC state-of-the-art with 3 configurations of stand-alone LANDSLIDE on the same test suite:

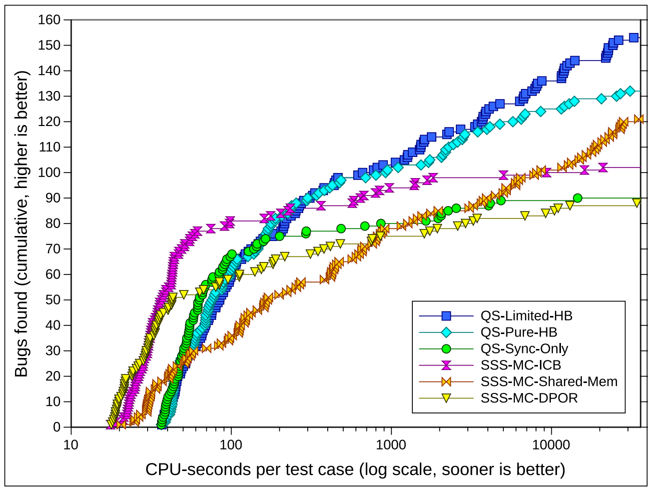
- **SSS-MC-DPOR**: Single state space using the maximal PP set from §3.2, explored with DPOR [23],
- **SSS-MC-ICB**: With PPs as above, but instead using ICB [40] with BPOR [14] to find bugs faster, and
- **SSS-MC-Shared-Mem**: Using ICB+BPOR, configured to preempt on any shared memory access [58] (decided at runtime, excluding threads' accesses to their own stacks), which in principle includes all possible data-race PPs.

Because parallelizing DPOR/ICB during SSS-MC is an open research problem [54], we optimistically gave control experiments a linear speedup of 10 hours per test with 1 CPU. QUICKSAND reports the CPU-time spent in addition to the wall-clock time for a resource-fair comparison (although, with the growing importance of multicore for performance, QUICKSAND's inherent parallelism is a convenient benefit). All tests ran on 12-core 3.2 GHz Xeon W3670 machines.

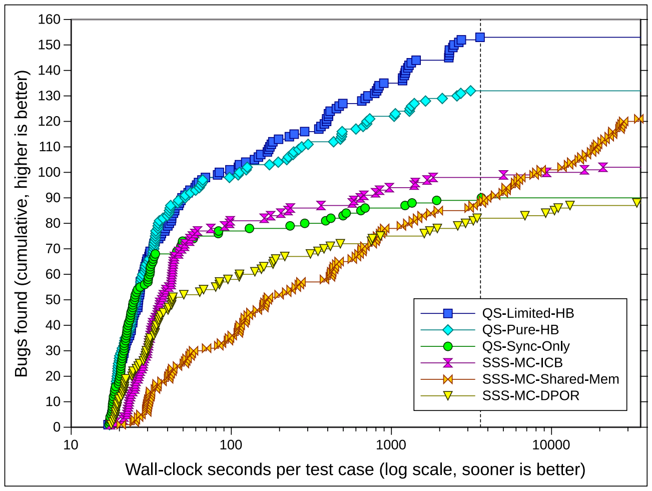
6.3 Comparison to State-of-the-Art MC

Figure 7 plots the cumulative distribution of bugs found by each experiment against the time taken to find each bug.

[4] Some of the Pintoses were partially implemented, so each test could only be run on a subset of the 78 submissions; see "Number tested" in Table 1.



(a) Bugs found as a function of elapsed CPU time. Overall, a more resource-fair comparison than (b), although QUICKSAND's start-up overhead is exaggerated, as the SSS-MC tests are not parallelized.



(b) Bugs found by elapsed wall-clock time. QUICKSAND is parallelized tenfold; the vertical line indicates its 1 hour limit.

Figure 7. Comparison of bug-finding performance by several configurations of QUICKSAND and the SSS-MC control. QUICKSAND finds 125% as many bugs at the 10-hour mark compared to the best SSS-MC approach.

Figure 7(a) is the main, resource-fair comparison by CPU-time; we additionally show a wall-clock comparison in (b) to highlight the impact of QUICKSAND's parallelism.

Finding new data-race bugs. Compared to SSS-MC-ICB (the fastest among the control experiments), QUICKSAND finds more bugs within any CPU budget greater than 200 seconds. Compared to SSS-MC-Shared-Mem (the best SSS-MC approach in the long term), QUICKSAND's Limited HB version ultimately concludes 10 CPU-hours with 125% as many bugs in total. Before the break-even point

		Total bugs					Data-race bugs					
	Num.	QUICKSAND		Control (SSS-MC)			Limited HB		Pure HB		Mutual	Avg. tested
Test	tested	LHB	PHB	ICB	DPOR	ShMem	All	Nondet.	All	Nondet.	timeouts	subset SSes
bcast	79	8	8	5	6	7	2	1	2	1	7	112.3
join	79	23	20	13	13	14	11	4	7	3	12	69.7
mx	79	10	9	1	1	12	9	1	8	1	0	-
sem	79	17	16	12	11	12	7	3	6	2	50	77.4
signal	79	10	8	5	5	11	6	1	3	2	45	59.6
rwlock	79	27	26	25	23	28	4	1	3	0	44	86.3
sched	59	7	7	1	1	8	6	4	6	6	2	13.0
alarm	44	21	12	16	5	29	17	1	7	6	17	7.8
wait	52	30	26	24	23	1	7	2	2	0	15	33.8
Total	629	153	132	102	88	122	69	15	44	21	192	65.8

Table 1. Summary of bugs found by each test program. QS-LHB and QS-PHB are QUICKSAND; ICB/DPOR/ShMem are the controls (§6.2). "Data-race bugs" counts among QUICKSAND's bugs how many required data-race PPs to expose (§6.3); among those, the "Nondeterministic" columns show how many candidates required MC integration to identify (§6.4). "Mutual timeouts" counts how often both QS-Limited-HB and SSS-MC-ICB timed out with no bug found; among those, "Average tested subset SSes" counts how many partial verifications QUICKSAND provided on average for each test (§6.3).

at 200 seconds, QUICKSAND lags behind SSS-MC-ICB due to additional start-up overhead from its tenfold parallelism. However, converting SSS-MC's early CPU-time advantage into faster wall-clock performance remains an open research problem [54]. Figure 7(b) gives QUICKSAND full credit for its inherent parallelism: with a 10 core allocation, it outperforms SSS-MC for any fixed budget of wall-clock time. After 1 hour of wall-clock time, tenfold QUICKSAND performs 158% as well as SSS-MC-ICB.

The left half of Table 1 breaks down the number and types of bugs found by each test program. In mx_test, in which we do not trust the lock implementation's correctness, SSS-MC-ICB and SSS-MC-DPOR found dramatically fewer bugs (just 1)[5]. Though it often suffices to assume correctly-implemented locks [50], we consider this strong evidence that new low-level synchronization code must be verified with data-race PPs.

Finding the same bugs faster. The QS-Sync-Only experiment tests whether Iterative Deepening is effective even for MC domains without data races. When QUICKSAND ignores all data-race candidates, its results are competitive with SSS-MC-DPOR, but SSS-MC-ICB outperforms it. This is unsurprising: the seed subsets of PPs QS-Sync-Only is limited to (§3.2) are much less flexible than ICB's preemption strategy (§2.1). This result suggests that in future work, QUICKSAND should consider using ICB in parallel with its default configuration when it finds no data-race candidates to test.

On the other hand, comparing QS-Limited-HB to SSS-MC-Shared-Mem shows that Iterative Deepening thoroughly outperforms ICB when shared-memory preemptions come into play. Statically configuring a PP for every shared memory access in advance produces orders of magnitude more PPs than waiting for an access to be identified as part of a (potential) data race at runtime. In principle, DPOR and BPOR should identify and prune any equivalences arising from extraneous PPs on non-conflicting accesses. However, in practice, the sheer number of accesses during each new execution (often thousands) added significant performance overhead to the MC when computing DPOR and backtracking. Iterative Deepening avoids this overhead by waiting until runtime to identify fewer, more relevant PPs dynamically, and is hence more suitable for MC with data-race PPs.

To ensure that our corpus of P2 and Pintos bugs gives an unbiased comparison between QUICKSAND and ICB, we also counted the preemption bounds necessary for ICB to find each of its bugs. Table 2 shows the distribution of these bounds, which is consistent with the results of [40], showing no bias towards bugs that would be harder for ICB to find.

Bound	SSS-MC-ICB	SSS-MC-Shared-Mem
0	2	1
1	82	86
2	16	32
3	2	3
4+	0	0
Total	102	122

Table 2. Distribution of preemption bounds among bugs found by ICB control experiments. (Bound 0 means the bug was found by switching threads only on `yield()`s.)

Partial verification. When a MC job times out, the user may prefer a brief summary of what parts of the test were verified, rather than writing off all the CPU time as a waste. While recent work [12] attempts to quantify the probability that a bug remains in some untested interleaving, QUICKSAND instead reports which subsets of PPs resulted in state

[5] The one bug SSS-MC found was in a fully-assembler lock implementation. `yield()`'s return value clobbered a value stored in `%eax`, which could lead to a failure after two repeated contentions. Preempting only on `yield()` (in the contention loop) was sufficient to find the bug.

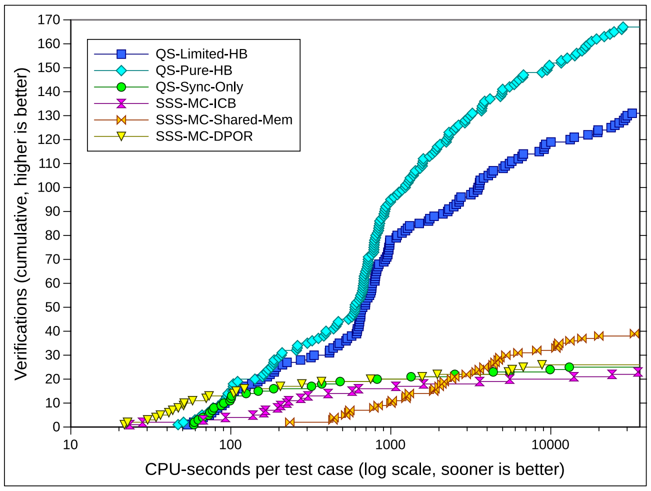


Figure 8. Cumulative distribution of tests fully verified by QS-Limited-HB, QS-Pure-HB, and SSS-MC-Shared-Mem (§5.1). 36 data-race-free tests were also soundly verified by QS-Sync-Only, SSS-MC-DPOR, and SSS-MC-ICB.

Test	Total DR PPs	Benign DRs	Untested DR PPs	Malloc DRs
`bcast`	655	97	150	52
`join`	566	68	249	338
`mx`	911	127	44	7
`sem`	783	2	414	166
`signal`	936	9	510	180
`rwlock`	543	1	310	156
`sched`	65	51	3	0
`alarm`	35	0	29	35
`wait`	71	1	28	31
Total	4565	356	1737	965

Table 3. Additional data race statistics. "Total DR PPs" counts how many unique data-racing instructions QS-Pure-HB identified among tests where it found no bugs. Among those, "Benign DRs" counts how many we refuted as non-failing (§6.3), while "Untested DR PPs" counts how many could not be checked in the time limit (§7). "Malloc DRs" counts how many false positive PPs QS-Limited-HB suppressed (§6.4).

spaces that did complete in time. On 229 tests, SSS-MC-ICB timed out after 10 hours with no bugs found. Among these tests, QUICKSAND found bugs in 37. For the other 192, we show the number of state spaces QUICKSAND was able to complete in the "Average tested subset SSes" column of Table 1. These completions guarantee that, if the test program could expose a bug, it would only be found by a new data-race PP not discovered yet, or by a superset combination of PPs not reached. Prior work [4] has argued the value of similar *compositional testing* when full verification is intractable, deferring to the user's expertise to judge the value of each subset of PPs verified.

Full verification. For 153 of our 629 tests, QS-Limited-HB was able to provide the total verification guarantee described in §5.1, and QS-Pure-HB completed a verification for 167 tests. In Figure 8 we plot the cumulative distribution of verifications provided by each approach. The next best approach for verifications was SSS-MC-Shared-Mem, which completed its search in only 39 cases.

Among QS-Limited-HB's verified tests, 36 contained no data-race candidates whatsoever, so the same verification could be provided with synchronization PPs only. We plot the verifications by QS-Sync-Only, SSS-MC-DPOR and SSS-MC-ICB as well: among these, the otherwise more antiquated SSS-MC-DPOR performs best, while the other two lag behind due to redundant work (§7). While QS-Sync-Only and SSS-MC-ICB are competitive with each other, using data-race PPs increases our verification capacity by 4.25x. Finally, assuming sequentially-consistent hardware, QS-Pure-HB classified many true data races as benign, while the SSS-MC-ICB approach could at best report such races to the user. We count these cases in Table 3.

Happens-Before comparison. Ultimately, using Limited HB for finding data-race candidates allowed QUICKSAND to find more bugs, while the "pure" Happens-Before analysis improved QUICKSAND's performance on verifications. This trade-off is attributable to the fact that QS-Limited-HB need not wait to test many alternate thread interleavings before a potential data-race candidate is confirmed; rather, it can add new jobs to start testing potential races immediately[6]. On the other hand, QS-Limited-HB can get overwhelmed by too many false positives, needing to refute such candidates by testing new state spaces, while QS-Pure-HB can refute false positives *en passant* by testing alternate interleavings in its original state spaces. This suggests that MCs which feature data-race analysis should implement both modes and offer the user to choose based on their testing philosophy.

6.4 Comparing to Single-Pass Data-Race Analysis

Beyond finding new bugs and completing full verifications with data-race PPs, we evaluated QUICKSAND's performance for classifying data-race candidates in two ways.

Suppressing "malloc-recycle" false positives. In §5.2 we showed the soundness of suppressing data race reports between two heap accesses when the surrounding memory was re-allocated in between. In Table 3, the column "Malloc-recycle DRs" shows the total number of such data-race candidates for each test program. In total, 965 data-races fit the malloc-recycle pattern across all tests, only 64 of which were observed to avoid the re-allocation in an alternate interleaving. Our proof in §5.2 guarantees the safety of pruning all 901 other state spaces.

[6] Table 1 corroborates: the difference is most dramatic in `alarm`, the test where QUICKSAND struggled most to finish even small subset jobs.

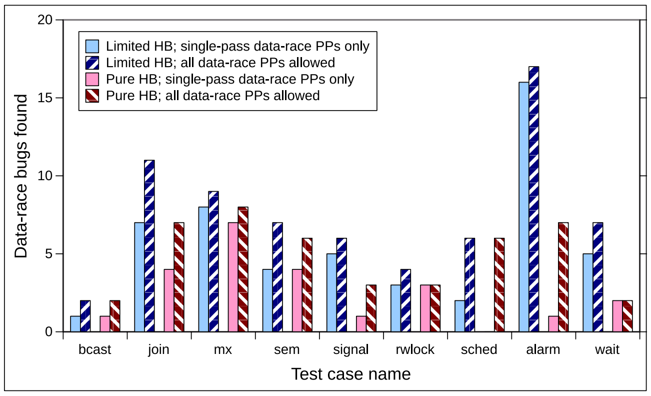


Figure 9. Some data-race candidates may not be identified during a single program execution. Using nondeterministic data races as PPs, QUICKSAND found 128% (Limited HB) to 191% (Pure HB) as many data-race bugs compared to using single-pass candidates alone.

Among those 64 true data-races, none exposed a new bug when used as a PP. This suggests that for other data-race tools, suppressing malloc-recycle candidates may be a productive heuristic, even if unsound without Iterative Deepening. However, QUICKSAND was able to correctly identify the 64 violations of that heuristic (among 26 distinct tests), and fall back to classifying them with DPOR.

Finding nondeterministic data-race candidates. Some memory accesses may be hidden in a control flow path that requires a nondeterministic preemption to be executed. In such cases, a single-pass dynamic data-race detector could fail to identify a racing access pair as a candidate at all. We counted how many such data-races, used as PPs, led to QUICKSAND finding new bugs, thereby making them *false negatives* of the single-pass approach. We classified each data-race candidate according to whether LANDSLIDE reported them during the first interleaving, before any backtracking or preempting: if so, they were *single-pass data races*; otherwise, *nondeterministic*.

To ensure a fair comparison, we disabled LANDSLIDE's *false-positive*-avoidance techniques during this experiment. For example, we reported malloc-recycle data races during the first interleaving, as a single-pass analysis must (§5.2). This prevents LANDSLIDE from suppressing an observed data race on the first interleaving, which would falsely classify it as nondeterministic.

Figure 9 compares the types of data-race candidates necessary to expose each data-race bug in our test suite. The first and third series represent the bugs found using PPs from single-pass data-race candidates, i.e., the state-of-the-art approach used by [31, 48]. The second and fourth series show all data-race bugs QUICKSAND found, which includes the former type as well as new bugs involving nondeterministic data-races. QS-LHB found 69 data-race bugs in total, 15 of which could not be found with single-pass data-race candidates alone. QS-PHB is even more dependent on nondeterministic data-race PPs, requiring nondeterministic data-race PPs in 21 cases among its 44 total data-race bugs.

Note that we are not comparing how much testing time is required before identifying the data-race candidates involved in each bug. Single-pass data races can all be found after a single program execution, while QUICKSAND may potentially take up to all 10 CPU-hours before identifying a nondeterministic data race. However, prior work data-race tools [49], being not integrated with a MC, are not intended to discover new candidates under subsequent runs. Running a single-pass data-race tool repeatedly for 10 CPU-hours could potentially uncover some nondeterministic candidates, but stress testing's comparative problem with achieving reliable coverage is already well-understood [13, 40]. Likewise, replay-based tools [31] are dependent upon the data-race detector to provide an execution trace leading to each candidate. This result suggests that such tools could benefit from a similar feedback loop as is used in Iterative Deepening.

7. Discussion

In this section, we discuss QUICKSAND's limitations and opportunities for future improvement.

Avoiding redundant work. When we extend a small state space with more PPs, the new state space is guaranteed to test a superset of interleavings compared to the old one. Any interleaving which does not preempt threads on any of the new PPs will be repeated work. This may make us slower than SSS-MC to find certain bugs, for example, if both `lock` and `unlock` PPs together expose a bug, but not either alone. Predicting whether an upcoming interleaving has already been tested is not straightforward, but we believe future implementations could incorporate cross-job memoization to prune some or all such repeated work. Prioritizing the maximal state space in particular could also improve completion times: whenever the maximal job finishes with no new data-races, future implementations could immediately prune all subset jobs and declare a total verification.

Finer-grained PP subsets. QUICKSAND was able to partially guarantee safety for some PPs in 93% of tests with too-large maximal state spaces. However, in 6 cases, no more than the minimal state space could be verified, and in 18 others, no state spaces were completed at all. While we used `within_function` (§4.1) *statically* to restrict where PPs could arise in advance of the test, future implementations could use this mechanism to *dynamically* subset PPs further, making partial verification of larger tests possible.

Integration with static data-race analysis. In §6.3, we evaluated SSS-MC's ability to find data-race-induced failures by configuring a static predicate to preempt on any non-stack memory access. This introduced hundreds of new PPs on each new test execution, with a prohibitive performance impact. While this performance could be improved by instead using a static or single-pass analysis to find data-race candidate PPs in advance [31], this strategy sacrifices the soundness of the verification guarantee, as shown in §6.4. However, QUICKSAND itself could employ static data-race analysis [20] in future work. Statically-identified data race candidates could heuristically be included in our "seed" PP sets (§3.2), enabling QUICKSAND to focus on the most suspicious races immediately, rather than waiting for them to be identified after potentially many iterations of MC.

Partial verification. While we guarantee safety when using certain combinations of PPs (§6.3), ICB guarantees safety under no more than a certain number of preemptions [40]. These guarantees could each be useful to developers in different scenarios, and future work could combine the two approaches to provide both at once. One benefit of our technique is that `within_function` would enable expert developers to restrict Iterative Deepening to only the modules of a codebase they wish to test.

Likewise, when full verification is not computationally feasible, some jobs with data-race PPs will time out. We cannot guarantee those races are benign, even though no bug was found. In the "Untested DR PPs" column of Table 3, we show how many such candidates we could not verify (38%). For a more formal treatment of these cases, we refer the reader to the *k-witness harmless* metric introduced by [31], which could be combined with QUICKSAND in future work.

8. Related Work

8.1 Stateless Model Checking

We build upon many established model-checking techniques, dating back to Verisoft, the original C model checker [25]. We compare related tools by their treatment of shared-memory thread communication.

Synchronization events only. CHESS [41] and dBug [51] instrument the thread library API, and can preempt programs only during calls to this API. Hence, they will miss any bugs that require interleaving threads at instruction granularity during a data race. [41] discusses the ability of CHESS to add PPs using a single-pass data-race analysis, but does not evaluate either the increase in bug-finding capacity or the soundness properties. Our convergence theorem (§5) is a natural extension of [40]'s Theorem 3, which alone provides soundness only for race-free programs.

Message-passing. Other stateless model checkers, such as SAMC [35], MaceMC [32], MoDist [57], ETA [52], and Concuerror [1], limit thread communication to a message-passing API to more effectively test distributed systems. This eliminates the need for data-race analysis, but restricts the class of programs that can be tested.

Preempting at instruction granularity is a prerequisite for using data-race PPs. However, the resulting state space explosion demands that any such tool either choose a small subset of instructions to consider as PPs or be limited to very small test programs. SKI [24] approaches kernel code by statically choosing a random set of instructions in advance, which is perhaps more similar to schedule fuzzing [12] than to exhaustive state space exploration. SPIN [27] specializes in verifying synchronization primitive implementations such as RCU [39], which is similar to our `mx_test` experiment, although it requires code to be written in the PROMELA language. Inspect [58] instruments source code by instrumenting all accesses to potentially-shared data. It identifies such instructions in advance with an over-approximating static alias analysis, while LANDSLIDE [7] traces the memory locations of accesses at runtime. Both SPIN and Inspect fix their set of PPs in advance, so could be extended with Iterative Deepening in future work.

Reduction techniques. Various improvements or alternatives to DPOR have been developed, such as Optimal DPOR [1], Dynamic Interface Reduction [26], and R4 for event-driven applications [30]. These are all compatible with our technique. Recent work [59] has extended DPOR for relaxed memory models [3], which we do not yet account for in our proofs (§5). SATCheck [16] and Maximal Causality Reduction [29] replace DPOR by using a SAT or SMT solver to search for schedules guaranteed to expose new behaviour. They improve reduction by considering values read and written to identify additional independences, while DPOR considers only addresses. They generate new schedules at memory access granularity, which Iterative Deepening could allow to be relaxed for large tests in future work. Parrot [15] combines MC with a partially-determinizing runtime for further reduction, but still, fewer than half the nontrivial tests in their evaluation could be completed, which motivates QUICKSAND's CPU-budget-oriented approach.

Restricting preemptions. Preemption Sealing [4] presents a mechanism similar to the `within_function` command (§4.1) for users to manually restrict preemptions. It demonstrates the need to consider subsets of PPs, as well as developers' willingness to limit a test case's scope so the resulting state space may be fully verified (§5.1). Probabilistic Concurrency Testing (PCT) [12] is a randomized algorithm that can quantify the probability of uncovering bugs. PCT targets tests with impossibly large state spaces, eschewing DPOR's depth-first search model to instead sample broad cross-sections of large state spaces. However, it proposes no alternate reduction algorithm to make up for its incompatibility with DPOR, so is unsuitable for verification of medium-sized tests. Future work could use ETAs to heuristically switch between DPOR and PCT. Finally, ICB [40] is most similar to our work, as both approaches provide a partial verification when full completion is intractable (§7). However, ICB cannot estimate remaining time to total verification, and can incorporate data-race PPs only when statically coded in advance. Our results in §6.3 outperform both such configurations of ICB.

8.2 Data Race Detection

Happens-Before. Many advances have been made on the false-positive potential data race problem since it was first introduced in [47]. [44] and [49] introduce Limited Happens-Before, which QUICKSAND uses to achieve its best bug-finding result. Other tools such as DJIT+ [46] and FastTrack [22] opt for the precise Happens-Before relation first defined by Lamport [34], which produces no false positives but is prone to more false negatives as shown in §2.2. FastTrack optimizes the representation of variable write clocks for performance, which guarantees to detect at least the first race on each variable. However, as we are interested in classifying data race candidates as *benign* or *buggy*, this optimization would be unsound for total verification if the first race on a variable were benign. Hence, we implement precise HB with write vector clocks as defined by DJIT+. Finally, [55] recently introduced the Causally-Precedes relation for a precise analysis which covers some common false negative cases such as our example in Figure 3(a).

Other domains. Several tools [6, 28, 37] have recently emerged to target Android applications, using domain-specific heuristics (orthogonal to our method) to reduce false positives. Like LANDSLIDE, DataCollider [21] finds data races in kernel code. IFRit [19] improves the performance of HB using an interference analysis, which could allow future work to avoid tracing every memory access.

Replay analysis. Closer to our work, replay analysis [42] also suppresses false positives by testing multiple thread interleavings. This work compares the program states immediately after the access pair for differences, preferring to err on the side of false positives. RaceFuzzer [48] avoids false positives by requiring an actual failure be exhibited, as we do, although it uses random schedule fuzzing rather than stateless MC. While this technique can classify malloc-recycle candidates as false positives (§5.2), they require replaying the threads in a new interleaving. Moreover, [31] argues that accurate classification may require many re-executions, which is tantamount to adding a new state space in QUICKSAND. Our proof in §5.2 allows us to eliminate this special case with no additional replay beyond what DPOR already requires.

Portend [31] is the most closely related work we have found. Based on reports from single-pass data-race analysis, it tests alternate executions to classify candidates in a taxonomy of likely severity. It uses symbolic execution to test input nondeterminism as well as schedule nondeterminism, and additionally reports non-failing races which nevertheless cause different program output. However, Portend does not test alternate interleavings *in advance* of knowing any specific data races, which is necessary to find certain bugs (§6.4) or to provide full verification (§5.1). Future work could combine the two approaches, using MC to produce new data-race traces for Portend to classify, or using Portend's analysis to inform QUICKSAND's heuristic priorities.

9. Conclusion

We have presented Iterative Deepening and QUICKSAND, a new technique and tool for automating the choice of preemption points (PPs) during stateless model checking. QUICKSAND incorporates data-race analysis to create new PPs tailored specifically to the program under test, and automatically finds state spaces that are appropriately sized to complete in a given CPU budget.

We achieve better bug-finding results than either single-pass data-race detection or single-state-space model checking alone, finding new bugs with data-race PPs that could not be exposed by preempting only on synchronization APIs. Moreover, when all data-race PPs can be fully tested within the CPU budget, we provide a verification as strong as if every single instruction had been used as a PP. Between data-race bugs and verifications, QUICKSAND is shown to provide the best of both worlds. By using 157 student operating system implementation projects as our test suite, we also show the potential benefit as a debugging platform in educational settings.

QUICKSAND is open-source and its interface can be adapted to fit any tool similar to LANDSLIDE. We have also posted our evaluation's data set. They are available at:

```
https://github.com/bblum/oopsla-quicksand
https://github.com/bblum/oopsla-dataset
```

Acknowledgments

Many thanks to Vaishaal Shankar, Haryadi Gunawi, and David A. Eckhardt for generously providing student implementations from Berkeley's, U. of Chicago's, and CMU's OS classes respectively. Thanks to Wind River for the use of their simulator SIMICS. Thanks to Jiří Šimša, Joshua Wise, Michael J. Sullivan, Carlo Angiuli, Jean Yang, Brandon Lucia, Haryadi Gunawi, John Wilkes, and the anonymous OOPSLA reviewers for their helpful comments. We also thank the member companies of the PDL Consortium (Broadcom, Citadel, EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, Intel, Microsoft Research, MongoDB, NetApp, Oracle, Samsung America, Seagate, Tintri, Two Sigma, Uber, Veritas and Western Digital). This work was supported in part by the U.S. Army Research Office under grant number W911NF0910273.

References

[1] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *Principles of Programming Languages*, POPL '14, pages 373–384. ACM, 2014.

[2] S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Commun. ACM*, 53(8):90–101, Aug. 2010.

[3] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, Dec. 1996.

[4] T. Ball, S. Burckhardt, K. E. Coons, M. Musuvathi, and S. Qadeer. Preemption sealing for efficient concurrency testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 420–434. Springer-Verlag, 2010.

[5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, Feb. 2010.

[6] P. Bielik, V. Raychev, and M. Vechev. Scalable race detection for Android applications. In *Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 332–348. ACM, 2015.

[7] B. Blum. Landslide: Systematic dynamic race detection in kernel space. Master's thesis, Carnegie Mellon University, Pittsburgh, PA, USA, May 2012. URL `http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-CS-12-118.pdf`.

[8] B. Blum. Soundness proofs for iterative deepening. Technical Report CMU-PDL-16-103, Carnegie Mellon University, September 2016. URL `http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-16-103.pdf`.

[9] H.-J. Boehm. How to miscompile programs with "benign" data races. In *Hot Topics in Parallelism*, HotPar'11, pages 3–3. USENIX Association, 2011.

[10] H.-J. Boehm. Position paper: Nondeterminism is unavoidable, but data races are pure evil. In *Relaxing Synchronization for Multicore and Manycore Scalability*, RACES '12, pages 9–14. ACM, 2012.

[11] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *Programming Language Design and Implementation*, PLDI '08, pages 68–78. ACM, 2008.

[12] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 167–178. ACM, 2010.

[13] K. E. Coons, S. Burckhardt, and M. Musuvathi. Gambit: Effective unit testing for concurrency libraries. In *Principles and Practice of Parallel Programming*, PPoPP '10, pages 15–24. ACM, 2010.

[14] K. E. Coons, M. Musuvathi, and K. S. McKinley. Bounded partial-order reduction. In *Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 833–848. ACM, 2013.

[15] H. Cui, J. Simsa, Y.-H. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Symposium on Operating Systems Principles*, SOSP '13, pages 388–405. ACM, 2013.

[16] B. Demsky and P. Lam. SATCheck: SAT-directed stateless model checking for SC and TSO. In *Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 20–36. ACM, 2015.

[17] D. Eckhardt. Pebbles kernel specification. `http://www.cs.cmu.edu/~410-s16/p2/kspec.pdf`, 2016.

[18] D. Eckhardt. Project 2: User level thread library. `http://www.cs.cmu.edu/~410-s16/p2/thr_lib.pdf`, 2016.

[19] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. Ifrit: Interference-free regions for dynamic data-race detection. In *Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 467–484. ACM, 2012.

[20] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Symposium on Operating Systems Principles*, SOSP '03, pages 237–252. ACM, 2003.

[21] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Operating Systems Design and Implementation*, OSDI'10, pages 1–16. USENIX Association, 2010.

[22] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Programming Language Design and Implementation*, PLDI '09, pages 121–133. ACM, 2009.

[23] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Principles of Programming Languages*, POPL '05, pages 110–121. ACM, 2005.

[24] P. Fonseca, R. Rodrigues, and B. B. Brandenburg. SKI: Exposing kernel concurrency bugs through systematic schedule exploration. In *Operating Systems Design and Implementation*, OSDI'14, pages 415–431. USENIX Association, 2014.

[25] P. Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *Computer Aided Verification*, CAV '97, pages 476–479. Springer-Verlag, 1997.

[26] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang. Practical software model checking via dynamic interface reduction. In *Symposium on Operating Systems Principles*, SOSP '11, pages 265–278. ACM, 2011.

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

[28] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn. Race detection for event-driven mobile applications. In *Programming Language Design and Implementation*, PLDI '14, pages 326–336. ACM, 2014.

[29] J. Huang. Stateless model checking concurrent programs with maximal causality reduction. In *Programming Language Design and Implementation*, PLDI 2015, pages 165–174. ACM, 2015.

[30] C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. Vechev. Stateless model checking of event-driven applications. In *Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 57–73. ACM, 2015.

[31] B. Kasikci, C. Zamfir, and G. Candea. Data races vs. data race bugs: Telling the difference with Portend. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 185–198. ACM, 2012.

[32] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: finding liveness bugs in systems code. In *Networked Systems Design & Implementation*, NSDI'07, pages 18–18. USENIX Association, 2007.

[33] R. E. Korf. Iterative-deepening-A: An optimal admissible tree search. In *International Joint Conference on Artificial Intelligence*, IJCAI'85, pages 1034–1036. Morgan Kaufmann Publishers Inc., 1985.

[34] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July

1978.

[35] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Operating Systems Design and Implementation*, OSDI'14, pages 399–414. USENIX Association, 2014.

[36] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, Feb. 2002.

[37] P. Maiya, A. Kanade, and R. Majumdar. Race detection for Android applications. In *Programming Language Design and Implementation*, PLDI '14, pages 316–325. ACM, 2014.

[38] A. Mazurkiewicz. Trace theory. In *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 279–324. Springer-Verlag New York, Inc., 1987.

[39] P. McKenney and J. Walpole. What is RCU, fundamentally? `https://lwn.net/Articles/262464/`, 2007.

[40] 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. ACM, 2007.

[41] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Operating Systems Design and Implementation*, OSDI'08, pages 267–280. USENIX Association, 2008.

[42] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. In *Programming Language Design and Implementation*, PLDI '07, pages 22–31. ACM, 2007.

[43] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Programming Language Design and Implementation*, PLDI '07, pages 89–100. ACM, 2007.

[44] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Principles and Practice of Parallel Programming*, PPoPP '03, pages 167–178. ACM, 2003.

[45] B. Pfaff, A. Romano, and G. Back. The Pintos instructional operating system kernel. In *Computer Science Education*, SIGCSE '09, pages 453–457. ACM, 2009.

[46] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Principles and Practice of Parallel Programming*, PPoPP '03, pages 179–190. ACM, 2003.

[47] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997.

[48] K. Sen. Race directed random testing of concurrent programs. In *Programming Language Design and Implementation*, PLDI '08, pages 11–21. ACM, 2008.

[49] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71. ACM, 2009.

[50] J. Simsa. *Systematic and Scalable Testing of Concurrent Programs*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2013. URL `http://repository.cmu.edu/dissertations/285/`.

[51] J. Simsa, R. Bryant, and G. Gibson. dBug: Systematic evaluation of distributed systems. In *Systems Software Verification*, SSV'10, pages 3–3. USENIX Association, 2010.

[52] J. Simsa, R. Bryant, G. Gibson, and J. Hickey. Efficient Exploratory Testing of Concurrent Systems. Technical Report CMU-PDL-11-113, Carnegie Mellon University, November 2011. URL `http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-11-113.pdf`.

[53] J. Simsa, R. Bryant, and G. Gibson. Runtime estimation and resource allocation for concurrency testing. Technical Report CMU-PDL-12-113, Carnegie Mellon University, December 2012. URL `http://www.pdl.cmu.edu/PDL-FTP/Storage/CMU-PDL-12-113.pdf`.

[54] J. Simsa, R. Bryant, G. Gibson, and J. Hickey. Concurrent systematic testing at scale. Technical Report CMU-PDL-12-101, Carnegie Mellon University, May 2012. URL `http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-12-101.pdf`.

[55] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *Principles of Programming Languages*, POPL '12, pages 387–400. ACM, 2012.

[56] P. Thomson, A. F. Donaldson, and A. Betts. Concurrency testing using schedule bounding: An empirical study. In *Principles and Practice of Parallel Programming*, PPoPP '14, pages 15–28. ACM, 2014.

[57] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: transparent model checking of unmodified distributed systems. In *Networked Systems Design and Implementation*, NSDI'09, pages 213–228. USENIX Association, 2009.

[58] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient stateful dynamic partial order reduction. In *Workshop on Model Checking Software*, SPIN '08, pages 288–305. Springer-Verlag, 2008.

[59] N. Zhang, M. Kusano, and C. Wang. Dynamic partial order reduction for relaxed memory models. In *Programming Language Design and Implementation*, PLDI 2015, pages 250–259. ACM, 2015.