Soundness Proofs for Iterative Deepening

Ben Blum

CMU-PDL-16-103
September 6, 2016

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

The Iterative Deepening algorithm allows stateless model checkers to adjust preemption points on-the-fly. It uses dynamic data-race detection to avoid necessarily preempting on every shared memory access, and ignores false-positive data race candidates arising from certain heap allocation patterns. An Iterative Deepening test that reaches completion soundly verifies all possible thread interleavings of that test.

Acknowledgements: Michael J. "Sully" Sullivan, Carlo Angiuli, and Nathaniel Filardo double-checked the proof and provided valuable feedback on notation style and the flow of explanation. Thanks as well to the anonymous OOPSLA reviewer #4, whose suggestions helped improve the formality of our definitions. We 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) and by the U.S. Army Research Office under grant number W911NF0910273.

Keywords: concurrency, stateless model checking, data race analysis, verification

1. Introduction

In our OOPSLA paper [1] we introduced Iterative Deepening, an algorithm which combines stateless model checking [5, 6] and data-race analysis [11, 13]. It dynamically adds new state spaces to test each time a data-race candidate is found. We present here two proofs concerning the technique.

First, we prove that given enough time, Iterative Deepening will converge to the same degree of schedule coverage that would be provided by a naïve model checker (MC) preempting on every single instruction. In other words, when Iterative Deepening finishes generating new data-race PPs, and completes all associated state spaces, it serves as a full verification of *all possible* schedules of the given test case.

Second, we prove the soundness of the *malloc-recycle* false-positive-elimination tactic discussed in the paper. Despite powerful reduction techniques, each state space is still exponentially sized, so we wish to avoid exploring any we could prove to be redundant. We prove that it is safe to eliminate malloc-recycle data race candidates immediately, without bothering to explore their associated state spaces.

This tech report is supplemental material to our main paper; we assume the reader is already familiar with our motivation and terminology.

2. Definitions

2.1 System Model

To avoid relying on any particular programming language features, we leave the program syntax and execution semantics opaque, reasoning instead about execution traces. We require that a program's evaluation produce a trace of instructions of the following form:

$$\begin{array}{lcl} \mathcal{A} & ::= & v \leftarrow \mathsf{read}(a) \quad | \quad \mathsf{write}(a,v) \quad | \quad \mathsf{xchg}(a,v) \\ & | & a \leftarrow \mathsf{malloc}(n) \quad | \quad \mathsf{free}(a) \\ & | & \mathcal{A}_{local} \quad | \quad \mathcal{A}_{sync} \end{array}$$

The execution steps take the following forms:

Memory. read, write, and xchg access global or heap memory shared by all threads, indicated by some address a. (Other atomic swap operations are omitted for brevity.) malloc and free provide access to fresh memory accessible by all threads.

Local state. $\mathcal{A}_{local}$ represents any thread-local instruction, such as modifying local variables, flow control, function calls, and assertions. We omit a detailed list for brevity, as we do not need to reason about them in these proofs.

Threads. $\mathcal{A}_{sync}$ denotes the subclass of evaluation steps which implement inter-thread synchronization; i.e., the *synchronization API*:

$$\begin{array}{lcl} \mathcal{A}_{sync} & ::= & \mathsf{mutex\_lock}(m) \quad | \quad \mathsf{mutex\_unlock}(m) \\ & | & \mathsf{deschedule} \quad | \quad \mathsf{make\_runnable}(t) \\ & | & t \leftarrow \mathsf{thread\_fork} \quad | \quad \mathsf{thread\_exit} \quad | \quad \mathsf{yield} \end{array}$$

mutex_lock and mutex_unlock provide mutual exclusion: a thread which evaluates mutex_lock on some lock m becomes blocked until no other thread holds m. deschedule alows a thread to block itself until another thread wakes it with make_runnable, and thread_fork and thread_exit allow creation and destruction of new threads. thread_fork is defined in the Pebbles manner [3]; we omit higher-level abstractions such as cond_wait, create, or join, which can be implemented using these primitives [4]. yield allows the execution semantics to switch threads while respecting the preemption-point-switching invariant discussed below.

Definition 1 (Interleaving)**.** *An interleaving (or execution trace) is a list of these instructions, annotated to indicate the currently-running thread as well as the runnability of all existing threads:*

$$\mathcal{I} \quad ::= \quad [\mathcal{A}, t, (t \rightarrow \mathsf{bool})]$$

We say a *thread switch* occurs when adjacent elements in $\mathcal{I}$ have different thread IDs t. We say a thread is *blocked* when its value in the runnability map is false.

Interleaving invariants. We require the evaluation semantics to produce interleavings which fulfil several invariants, apart from the obvious ones ensuring correct synchronization and malloc discipline.

Model checkers often include heuristics to identify blocking via open-coded yield loops, but we assume here that such patterns are implemented more tastefully with a condition-variable-like primitive built upon deschedule.

Most importantly, we assume that threads switch only at instructions identified by the set of *preemption-point predicates*, defined below.

2.2 Stateless model checking terms

Definition 2 (Preemption point (PP) predicate)**.** *A predicate on the execution state which identifies a class of instruction pairs between which we may force threads to switch.*

We use *synchronization API PPs* to denote the set of predicates which occur immediately before or after any of $\mathcal{A}_{sync}$. Because no other instruction affects a thread's runnability, it is always possible to execute a program by switching the currently-executing thread only at synchronization API PPs.

All data-race PP predicates will occur immediately before a read, write, or atomic swap. Other predicates are possible, though we will show they are irrelevant.

When generating execution traces, the evaluation semantics is parameterized by a set of active PP predicates. As long as the set contains the synchronization API PPs, the thread-switch invariant discussed above will hold.

Definition 3 (Preemption point (PP))**.** *A PP is any site in an interleaving at which a PP predicate evaluated to true.*

As discussed above, all thread switches occur at PPs. However, an interleaving may also contain PPs at which the same thread continued running.

In the main paper, "PP" referred to what we now call "PP predicates". Hence, the "same" PP could occur multiple times during an execution. In these proofs, we separate such cases into multiple unique PPs: each PP is simply a label denoting the boundary between two transitions.

Definition 4 (Transition)**.** *A sequence of execution steps from a program's evaluation between two PPs.*

The thread-switch invariant guarantees that each transition's instructions are associated with exactly one thread.

Definition 5 (State space)**.** *A state space $\mathcal{S}$ is a set of interleavings representing all execution sequences legal under a given set of PP predicates.*

Definition 6 (Must-happen-before (MHB))**.** *Let t_1 and t_2 be two transitions of an interleaving, and $T1$ and $T2$ be the corresponding thread IDs, and let t_1 occur before t_2. Then t_1 MHB t_2 if*

1. *$T2$ is blocked immediately preceding t_1 and not blocked immediately afterward, and there does not occur another t_2' by $T2$ between t_1 and t_2; or*
2. *there occurs some t_3 by thread $T3$ such that t_1 MHB t_3, t_3 MHB t_2, and $T3 \neq T2$; or*
3. *$T1 = T2$.*

Intuitively, MHB expresses when two transitions cannot be reordered (the "enables" relation in DPOR terminology [5]). Two transitions A and B of different threads MHB if some synchronization event in A causes B to become runnable while it was previously blocked. Such synchronization events include thread_fork, make_runnable, and sometimes but not always mutex_unlock.

Note how our *must*-happen-before relation differs from the conventional definition of happens-before ("observed to happen before") [8]. Our use of MHB matches the "limited happens-before" used in [11] and [14]; the advantage of this over pure-happens-before detectors in producing fewer false negatives is well-argued in those prior works[1]. We illustrate the difference in Figure 1.

Note also that although transitions of the same thread are related by MHB, MHB is transitive only when the latter two transitions are not by the same thread (condition 2). While lock-protected critical sections can be reordered around each other (i.e., line 1 not MHB lines 8-9), one cannot be reordered to be in the middle of the other (i.e, lines 3-4 MHB line 6). In the latter case, the MHB relation is established by the mutex's blocking mechanism used during contention.

Our main paper refers to this relation (in conjunction with a lock-set analysis) as Limited HB.

[1] Because pure-HB data race detectors avoid false positives altogether, they would have no trouble avoiding our malloc-recycle false positives. However, as prior work has shown, they miss many other bugs involving unprotected variables accessed alternately before and after mutex-protected critical sections.

```
   Thread 1                Thread 2
1  my_x->foo = ...;
2  mutex_lock(...);
3  global_x = my_x;
4  mutex_unlock(...);
5                          mutex_lock(...);
6                          my_x = global_x;
7                          mutex_unlock(...);
8                          if (my_x != NULL)
9                              my_x->foo = ...;
```

Figure 1. Example program to illustrate the difference between *pure happens-before* and *must-happen-before*. Under pure happens-before (which does not identify false positives), lines 1 and 9 are not a data race candidate. Under MHB, they are; although after trying to reorder them, it will be classified as a false positive.

Definition 7 (Shared memory conflict)**.** *A pair of memory accesses between two threads to the same address where at least one of them is a write.*

Definition 8 (Independent transitions)**.** *Two transitions between different threads are independent if the intersection of their shared memory accesses contains no conflicts.*

Definition 9 (Equivalent interleaving)**.** *Two interleavings are equivalent if one can be transformed into the other by permuting only independent transitions.*

Intuitively, the behaviour of a program could change by reordering two transitions only if they contain a memory conflict. All possible interleavings of a program can be partitioned into equivalence classes, so only one interleaving from each equivalence class need be tested to ensure total schedule coverage [9]. Equivalence is, of course, transitive.

Definition 10 (Dynamic Partial Order Reduction (DPOR))**.** *A state-space search algorithm for stateless model checkers; given a state space $\mathcal{S}$, it will test at least one interleaving from each equivalence class in $\mathcal{S}$.*

Considering an interleaving $\mathcal{I}$ in $\mathcal{S}$, if two transitions t_1 and t_2 by different threads are not independent and not related by MHB, let $\mathcal{J}$ be the interleaving which reorders t_1 with t_2. DPOR is then guaranteed to test some interleaving in $\mathcal{S}$ equivalent to $\mathcal{J}$ [5].

Because equivalent interleavings produce identical execution states, DPOR guarantees to expose all reachable execution states by testing its subset of interleavings. We refer to this property as *the soundness of DPOR*.

2.3 Data race and other memory terms

Definition 11 (Data race)**.** *A shared memory conflict where furthermore:*

- *The intersection of both threads' locksets is empty (i.e., the same lock does not protect both accesses), and*
- *The containing transitions are not related by MHB.*

The same as in the paper, we distinguish between data-race *candidates* (or *potential* data races) and data-race *bugs*. For brevity, we now use "data race" to refer both to true races and to potential data-race candidates identified by MHB.

Definition 12 (False positive data race)**.** *An apparent data race that cannot be executed in the opposite order from what was observed.*

False positives are caused when some data dependency based on some other shared state changes some variable values when the threads are reordered, such that the memory addresses no longer collide.

Definition 13 (Malloc-recycle data race)**.** *A data race where the address is contained in some heap-allocated memory, and between the two accesses, that memory was passed to free() and returned again by a subsequent malloc().*

Figures 2 and 3 show an example. In the case of malloc-recycle false positives, the allocation heap is the "other shared state" mentioned in the previous definition, and malloc's return value is the variable value that changed.

Recent work [12] proposed hardware techniques for detecting many classes of stale heap pointer accesses, including the one shown in Figure 3. Future work could combine this approach with MC to identify such bugs immediately, rather than requiring Iterative Deepening to explore new state spaces corresponding to the data race. However, if the `malloc` call were in thread 1 instead of thread 2, the bug would still be nondeterministic, requiring MC to expose.

Definition 14 (Use after free)**.** *Any read or write to heap memory which was once allocated, but no longer is.*

These can immediately be identified as failures by a MC which tracks allocation state.

3. Intuition

This section provides (hopefully) intuitive summaries of our proof goals.

Intuition for Iterative Deepening convergence. We will prove that when Iterative Deepening saturates the set of data-race PPs, that set represents every instruction where a preemption could possibly affect the program's behaviour Hence, completing the associated state spaces is as strong a verification as testing all possible thread interleavings by preempting anywhere. A data-race may be hidden in control-flow paths reachable only after preempting on a different data-race, but the technique's iterative nature will eventually find it. On the other hand, relying on the soundness of DPOR, preempting on an instruction which is neither a data-race or sync API boundary cannot affect the program's behaviour, so any such PPs are irrelevant.

Note that while we defined synchronization API PPs to occur both before and after any $\mathcal{A}_{sync}$ instruction, we will only add data-race PPs *before* their associated read or write. Our proof requires preempting only before each racing instruction, not after, in order to fully saturate the PP set with all possible data races. Then, once saturated, preempting after each racing instruction would be extraneous, because all subsequent instructions before the next PP must be local operations or non-communicating reads/writes[2].

Intuition for malloc-recycle soundness. We prove that if a malloc-recycle-pattern data race is not a false positive, then DPOR will eventually interleave threads in such a way that the malloc-recycle pattern will disappear, while the access pair remains for the data-race detector to find, as shown in Figure 4. Hence, in the same state space where the malloc-

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

Thread 1            Thread 2
x1->foo = ...;
free(x1);
                    // x's memory recycled
                    y = malloc(sizeof *y);
                    // ...initialize...
                    publish(y);
                    y->bar = ...;
```

Figure 2. False-positive malloc-recycle pattern. This is the common case for which we avoid creating new state spaces.

```
Thread 1            Thread 2
publish(x1);
                    x2 = get_published_x();
x1->foo = ...;
free(x1);
                    // x's memory recycled
                    y = malloc(sizeof *y);
                    x2->foo = ...;
```

Figure 3. Adversarial program which fits the malloc-recycle pattern, but nevertheless contains a true race.

```
Thread 1            Thread 2
publish(x1);
                    x2 = get_published_x();
                    // x not free, so malloc's
                    // return value changes!
                    y = malloc(sizeof *y);
                    x2->foo = ...;
x1->foo = ...;
free(x1);
```

Figure 4. Goal interleaving, reordering the adversarial threads away from the pattern, while the data race remains.

[2] Our implementation also optimizes the synchronization API PPs, generally preempting only *after* each synchronization instruction. mutex_lock is the exception, as it can cause other threads to become blocked. All the others can only make blocked threads runnable, establishing MHB, which also provides MHB for the preceding transition of the invoking thread. Hence, coalescing those transitions does not exclude any possible interleavings.

recycle data race was found, if it's a true race, the same race will also appear without the recycle pattern. So if that race can lead to a failure, Iterative Deepening will still be led to the necessary preemption point to find it.

4. Assumptions

This section documents our assumptions about the concurrency model, language model, and test environment, and discusses some limitations that may arise therefrom.

4.1 Assumptions for both proofs

Maximal state space. We assume the model checker has all synchronization API PPs enabled during this state space exploration, and that we are not limited by a pressing CPU budget. We assume that all synchronization primitives not listed in our system model are built upon them, or otherwise annotated for the MC to add additional PPs for them.

Using only a subset of PPs or aborting early due to timeout could each ruin our ability to reach the goal interleaving or goal state space. However, Iterative Deepening aims to test the most important interleavings with the time available, so in the case of not enough time, continuing the current state space fits that goal best.

Shared memory thread communication. We assume that the only way for two threads' transitions to affect each other's behaviour, should they be reordered, is through either shared memory or a correctly-instrumented sync API. Both DPOR and data-race detection rely on this assumption, as any other way for threads to affect each other's behaviour could invisibly reduce independence and break soundness. For brevity in these proofs, we refer to all thread communication as shared memory, and assume that other mechanisms, such as system calls that access the filesystem, could be instrumented to fit the same model.

Schedule nondeterminism only. We discount the possibility of other types of nondeterminism, such as program input nondeterminism (including randomness/timestamps) or store-buffer nondeterminism on weak-memory architectures. We refer the reader to [2, 7] for related work on the former, and to [18] for the latter.

4.2 Assumptions for Iterative Deepening convergence

Locks are correct. Because hybrid data-race detection uses lockset analysis to exclude many candidate access pairs, we assume that no preemption during a lock-protected critical section could cause a contending thread to make progress. This extends to use of disabling/enabling interrupts in kernel code, which we model as a single global lock, although we do not model RCU [10].

Should the user wish to verify these properties of locks, they could either run a locking test separately (as we suggest in our paper), which would cheaply test the locks to an extent limited by the separate test case; or they could remove the data-race analysis's lockset tracking, which would expensively test all the main program's required locking properties in tandem with the program itself.

We also assume that any unusual locking discipline, such as recursive mutexes or lock hand-off, is correctly annotated to the data-race analysis. Our implementation also models the behaviour of r/w locks and 1-initialized semaphores (the latter heuristically), although this is tangential to the proof.

Halting. From a certain perspective, convergence is the same as completeness, which should be impossible for any runtime analysis of a Turing-complete language [16]. However, being already limited by practical real-world CPU budgets, we are already accepting that many tests will time out. We are concerned with the limited case of when QUICKSAND *does* terminate with all state spaces completed.

Inversely, when we prove that Iterative Deepening will eventually find an arbitrary buggy interleaving, we assume that no intermediate state spaces contain nontermination bugs. If they do, we are satisfied with finding that bug instead. In this proof, we assume that the MC's heuristic infinite loop checker has no false negatives; i.e., it will never get stuck forever in an infinite loop without identifying a bug (necessarily accepting some false positives).

4.3 Assumptions for malloc-recycle soundness

Malloc is a magic black box. We assume the malloc implementation is correct (e.g., it won't double-allocate), although we don't assume any implementation details such as a tendency to reuse blocks or allocate adjacent ones. In fact, our implementation ignores all potential PPs arising from malloc's internal mutex; in essence treating it as a "magic primitive", because we are not interested in verifying its implementation. Furthermore, we configured DPOR to ignore any shared memory conflicts arising from internal heap metadata to achieve greater state space reduction. (If the only consequence of reordering two transitions is malloc returning different addresses, we consider them independent). This is not without consequences; see section 6.4.

Sharing heap addresses. Finally, we assume that the only way the program can obtain heap addresses is through the return value of `malloc`. Because we are testing C programs, any bizarre violations of this assumption are technically possible, but should you wish to check for bugs like this, symbolic execution [2] would be more appropriate.

In Section 6, we further assume a `malloc`ed block's address cannot be obtained through arithmetic on the address of a different block; in Section 6.4 we account for this case by relaxing the previous "black box" assumption.

5. Proof of Iterative Deepening convergence

We seek to prove that, given enough time, Iterative Deepening offers the same coverage of thread interleavings that could be achieved by preempting at every single instruction. We'll call the latter the *naïve state space*, and call the condition of testing all its interleavings *convergence*[3]. Hence, our convergence statement is as follows:

Theorem 1 (Total verification)**.** *If Iterative Deepening fully saturates its data-race PP predicates and completes all associated state spaces, it serves as a verification of all possible thread schedules of the given test program.*

The contrapositive statement offers more structure for an easier proof[4]:

Theorem 2 (Convergence of Iterative Deepening)**.** *If a bug is exposed by an interleaving in the naïve state space, Iterative Deepening will eventually test an equivalent interleaving which exposes the same bug.*

Let

$$\mathcal{I} = [(t_{\alpha 1}, p_{\alpha 1}), (t_{\beta 1}, p_{\beta 1}), ...(t_{\alpha n}, p_{\alpha n}), ...]$$

be an interleaving, where the element $(t_{\alpha i}, p_{\alpha i})$ represents the ith transition t of thread α, and the associated preemption point p at its end. We will use the following notation:

- $\mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)$ indicates the first instruction executed during $t_{\alpha(i+1)}$ (the *next* transition), i.e., the instruction of α that $p_{\alpha i}$ preempted just before.
- $\mathsf{others}_{\mathcal{I}}(\alpha, i)$ indicates the set of all transitions by other threads between $t_{\alpha i}$ and $t_{\alpha(i+1)}$.
- $\mathsf{conflicts}(t_\alpha, t_\beta)$ indicates the set of shared memory conflict pairs between t_α and t_β.

5.1 Equivalence of irrelevant PPs

Iterative Deepening will not, of course, preempt on exactly the same instructions that an arbitrary naïve interleaving would, as it is only capable of preempting on data races and sync API boundaries. Thus, our first task is to show that all naïve interleavings have an equivalent interleaving which includes only data-race and sync API PPs.

Definition 15 (Relevant preemption point)**.** *We say a PP $p_{\alpha i}$ is* relevant *if either $p_{\alpha i}$ is a synchronization API PP, or if*

$$\mathsf{conflicts}(\mathsf{others}_{\mathcal{I}}(\alpha, i), \mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)) \neq \emptyset$$

i.e., the instruction by thread α immediately after $p_{\alpha i}$ has a memory conflict with some other thread interleaved between $t_{\alpha i}$ and that instruction.

Definition 16 (Fully-relevant interleaving)**.** *An interleaving comprised only of relevant PPs.*

Lemma 1. *For any interleaving in the naïve state space, there exists an equivalent interleaving which uses only relevant PPs.*

Proof. Let $p_{\alpha i}$ be the first irrelevant PP of a naïve interleaving $\mathcal{I}$. We ask, did some thread among $\mathsf{others}_{\mathcal{I}}(\alpha, i)$ conflict with any instruction from $t_{\alpha(i+1)}$, even though there was no conflict with $\mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)$?

- If $\mathsf{conflicts}(\mathsf{others}_{\mathcal{I}}(\alpha, i), t_{\alpha(i+1)}) \neq \emptyset$, or if $t_{\alpha i}$ contains a sync API instruction that's not already a PP, then let χ be either the first instruction by $t_{\alpha(i+1)}$ among the conflicts or the first sync API instruction, whichever comes first. Let $\mathsf{pfx}(t_{\alpha(i+1)}, \chi)$ denote the instruction sequence between $\mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)$ and χ, including the former but not the latter. Now, we output the new interleaving:

$$\begin{aligned} \mathcal{I}' = [..., \quad & (t_{\alpha i} \cup \mathsf{pfx}(t_{\alpha(i+1)}, \chi), p'_{\alpha i}), \\ & \mathsf{others}_{\mathcal{I}}(\alpha, i), \\ & (t_{\alpha(i+1)} \setminus \mathsf{pfx}(t_{\alpha(i+1)}, \chi), p_{\alpha(i+1)}), \quad ...] \end{aligned}$$

  In other words, we reorder $\mathsf{others}_{\mathcal{I}}(\alpha, i)$ to between $\mathsf{pfx}(t_{\alpha(i+1)}, \chi)$ and χ, removing the irrelevant $p_{\alpha i}$ and adding a new PP $p'_{\alpha i}$, which is relevant by the construction of χ. It must be possible to reorder $\mathsf{others}_{\mathcal{I}}(\alpha, i)$ to after α's execution because no synchronization occurs during $\mathsf{pfx}(t_{\alpha(i+1)}, \chi)$, hence the other threads' runnability cannot be affected. Likewise $\mathsf{pfx}(t_{\alpha(i+1)}, \chi)$ is independent with $\mathsf{others}_{\mathcal{I}}(\alpha, i)$, so by the soundness of DPOR, $\mathcal{I}' \equiv \mathcal{I}$.
- Otherwise, $\mathsf{conflicts}(\mathsf{others}_{\mathcal{I}}(\alpha, i), t_{\alpha(i+1)}) = \emptyset$, and no instructions among $t_{\alpha(i+1)}$ are synchronization. Then we output the new interleaving:

$$\mathcal{I}' = [..., (t_{\alpha i} \cup t_{\alpha(i+1)}, p_{\alpha(i+1)}), \mathsf{others}_{\mathcal{I}}(\alpha, i), ...]$$

  In other words, we reorder $\mathsf{others}_{\mathcal{I}}(\alpha, i)$ with the entire next transition by thread α. This must be possible for the same reasons as above, and again, the transitions are independent, so $\mathcal{I}' \equiv \mathcal{I}$.

This constitutes an algorithm for inductively converting (or removing) all irrelevant PPs to relevant ones. □

5.2 Saturation of data-race PPs

Now we must show that, starting from a sync-PP-only state space, Iterative Deepening will eventually encounter all data races which are used as PPs in any buggy fully-relevant interleaving. The challenge is that some data-race candidates are nondeterministic to find; i.e., they may be hidden in control flow which requires a prior preemption on a different data race to expose, as shown in Figure 5. Hence, the maximal state space of statically-available sync API PPs will not necessarily uncover all possible data-race PPs; we may need to iterate through some data-race state spaces before finding certain nondeterministic races.

Definition 17 (Reachable data race)**.** *A data race candidate which will be identified by a MC configured to preempt only on locking API boundaries, or transitively also configured to use data-race PPs of other reachable data races.*

[3] It could also be called soundness, taking the perspective that Iterative Deepening is a search ordering heuristic that doesn't miss any interleavings.

[4] For the reader who likes to avoid non-constructive proofs [17], note that we use the constructive contrapositive direction: Theorem 2 is $A \rightarrow B$ and Theorem 1 is $\neg B \rightarrow \neg A$.

```
int x = 0, y = 0;
bool t1_x = false, t1_y = false;
bool t2_x = false, t2_y = false;
```

Thread 1 (Thread 2 similar, with `t1` and `t2` vars swapped)

```
1  x = x + 1;
2  t1_x = true;
3  // "if x raced"
4  if (x == 1 && t2_x) {
5      y = y + 1;
6      t1_y = true;
7      // "if y raced"
8      if (y == 1 && t2_y) {
9          panic();
10     }
11 }
```

Figure 5. Not all data races will immediately be uncovered by sync API PPs alone. Here, preempting during line 1's race is necessary to force the program to execute the racy line 5.

Definition 18 (Reachable preemption point)**.** *A PP $p_{\alpha i}$ such that either* $\mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)$ *is part of a reachable data race, or a synchronization instruction.*

Lemma 2. *All PPs of any fully-relevant interleaving are reachable.*

Proof. Let $\mathcal{I} = [(t_{\alpha 1}, p_{\alpha 1}), ...]$ be the fully-relevant interleaving and PP sequence in the premise which exposed a bug. We proceed by induction on the PPs according to their order in $\mathcal{I}$[5]. For both the base case and inductive step, we know (vacuously, for the former) that for each other PP $p_{\beta h}$ with $t_{\beta h} \prec t_{\alpha i} \in \mathcal{I}$, $p_{\beta h}$ is reachable. We must show that a data race involving $\mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)$ is reachable, and we are not allowed to use $p_{\alpha i}$ until we find the data race between $\mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)$ and $\mathsf{others}_{\mathcal{I}}(\alpha, i)$.

Coalescing not-yet-reachable data race PPs. Consider the alternate interleaving prefix:

$$\mathcal{J} = [..., (t_{\alpha i} \cup t'_{\alpha(i+1)}, p_{\alpha j}), \mathsf{others}_{\mathcal{I}}(\alpha, i)']$$

where we have reordered $t_{\alpha(i+1)}$ to before the execution of $\mathsf{others}_{\mathcal{I}}(\alpha, i)$. Here, $p_{\alpha j}$ is the first sync API PP in α after $p_{\alpha i}$ (as $p_{\alpha(i+1)}$ may be a not-yet-reachable data race PP), and $t'_{\alpha(i+1)}$ may include transitions of α beyond $t_{\alpha(i+1)}$ itself. Then, $\mathsf{others}_{\mathcal{I}}(\alpha, i)'$ are the other threads' transitions from our target interleaving. except they may be altered by the presence of some *other* data race in $t'_{\alpha(i+1)}$ occurring after $\mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)$. Likewise, $t'_{\alpha(i+1)}$ may be altered by some other data race in $\mathsf{others}_{\mathcal{I}}(\alpha, i)'$. The only certainty so far is that $t'_{\alpha(i+1)}$ begins with $\mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)$.

It would be straightforward to show that any other data races in $t'_{\alpha(i+1)}$, which would be discovered in this interleaving, could be reordered to after $\mathsf{others}_{\mathcal{I}}(\alpha, i)'$, eventually transforming $t'_{\alpha(i+1)}$ to contain no conflicting memory accesses but $\mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)$ itself. Unfortunately, $\mathcal{J}$ is not necessarily reachable, as $\mathsf{others}_{\mathcal{I}}(\alpha, i)'$ still includes any data-race PPs from $\mathsf{others}_{\mathcal{I}}(\alpha, i)$, which were ordered after $p_{\alpha i}$ in $\mathcal{I}$ and hence not covered by the inductive assumption. Hence, we must perform the same "coalescing" for each thread in $\mathsf{others}_{\mathcal{I}}(\alpha, i)'$ that we did for α, which we will abbreviate:

$$\mathcal{K} = [..., (t_{\alpha i} \cup t'_{\alpha(i+1)}, p_{\alpha j}), \mathsf{coalesce}(\mathsf{others}_{\mathcal{I}}(\alpha, i)')]$$

For each other thread β, $\mathsf{coalesce}(\mathsf{others}_{\mathcal{I}}(\alpha, i)')$ will contain a single transition, starting with the first instruction of β's corresponding transition in $\mathsf{others}_{\mathcal{I}}(\alpha, i)$, and ending with the next sync API PP in β, which we'll call $p_{\beta k}$. Again, the only certainly-executed instruction by α after $t_{\alpha i}$ is $\mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)$. It is even possible that neither $\mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)$ nor any other instruction by thread α will conflict with any other thread, until a different data race PP between two other threads is discovered.

It follows from the inductive hypothesis (which covers the prefix indicated by "..."), and from sync API PPs being reachable by definition, that $\mathcal{K}$ is reachable.

Finding the next reachable data-race PP. Now, we show by induction that a data race on $\mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)$ is reachable. We assume that a state space $\mathcal{S}$ containing $\mathcal{K}$, plus n unique data race PPs among $t'_{\alpha(i+1)}$ and $\mathsf{others}_{\mathcal{I}}(\alpha, i)$, will be reachable. (In the base case, $n = 0$, and $\mathcal{K}$'s reachability is justified above.) We must show that in this state space, if the $\mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)$ data race is not reachable, a new unique data-race PP will be reachable instead.

By the definition of relevance, $\mathcal{I}$ guarantees that some other thread ω can execute a data-racing instruction with $\mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)$. By the soundness of DPOR, if a program behaviour is possible by interleaving at the boundaries of the given transitions, that interleaving will be tested. By contrapositive, because ω's data-racing instruction was not tested in $\mathcal{S}$, one or more necessary interleaving sites must be in the middle of some transition, rather than all at boundaries.

However, we do not get a *single* such transition for free. We have arrived at the crux of the proof: to show that there cannot be multiple data-race PPs which must both be enabled before either data race can be identified. Such a circular dependency seems intuitively impossible, but to actually find the "first reachable" PP is not straightforward. We require that in $\mathcal{S}$, there exists a single transition $t_{\beta k}$ that can alone be split into $[t_{\beta k1}, t_{\beta k2}]$, and some other communicating transition $t_{\gamma l}$ which conflicts with $t_{\beta k2}$.

We show this by contradiction. Assume that for all $t_{\beta k} \in \mathcal{S}$, and all possible points $p'_{\beta k}$ at which to split it into $[t_{\beta k1}, t_{\beta k2}]$, and all other non-MHB transitions $t_{\gamma l}$, $t_{\gamma l}$ has no shared memory conflicts with $t_{\beta k2}$. Let $\mathcal{S}' = \mathcal{S} \cup p'_{\beta k}$, i.e., the state space obtained by adding any such $p'_{\beta k}$ to $\mathcal{S}$'s PP-set. By the soundness of DPOR, because any such $t_{\gamma l}$ is

[5] Note that this is not necessarily the same as the order of the racing instructions $\mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)$, which occur in $t_{\alpha(i+1)}$, not in $t_{\alpha i}$.

independent with $t_{\beta k2}$,

$$[..., t_{\beta k1}, t_{\beta k2}, t_{\gamma l}] \equiv [..., t_{\beta k1}, t'_{\gamma l}, t'_{\beta k2}]$$

Hence, $\mathcal{S} \equiv \mathcal{S}'$. Then, the above assumption also applies to $\mathcal{S}'$, showing that for any *pair* of transitions such as $t_{\beta k}$, adding two new PPs cannot expose new program behaviour. Inductively, no set of new PPs of any size would expose new behaviour not already exposed in $\mathcal{S}$. However, the instruction by ω which conflicts with $\mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)$ was not observed in $\mathcal{S}$, so we have our contradiction.

Hence, if $\mathcal{S}$ does not expose the $\mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)$ data race directly, there must exist some transitions $t_{\beta k} = [t_{\beta k1}, t_{\beta k2}]$ and $t_{\gamma l}$ such that $t_{\gamma l}$ shared-memory-conflicts with $t_{\beta k2}$ and could be interleaved immediately before it. By the maximal state space assumption, all sync API PPs are already enabled, so the locksets of β and γ cannot overlap and there is no MHB relation. Hence the memory conflict between $t_{\gamma l}$ and $t_{\beta k2}$ will be identified as a data-race. Finally, because $t_{\beta k}$ was not previously split by a PP, the data-race was not already discovered.

Reaching $\mathbf{p_{\alpha i}}$. Hence either a data-race will be identified including $\mathsf{nextinstr}_{\mathcal{I}}(\alpha, i)$, or an infinite/nonterminating sequence of other unique data-race PPs will be identified, but by the halting assumption, this would constitute an infinite loop bug, which is sufficient. (Alternatively, for any program with a finitely-sized instruction listing, the number of unique instruction pairs is finite.) Hence by the "next reachable data-race" induction, $p_{\alpha i}$ is reachable. Hence by the "$p_{\alpha i}$ is reachable" induction, all PPs in $\mathcal{I}$ are reachable. □

5.3 Conclusion

Theorem 2 (Convergence of Iterative Deepening)**.** *If a bug is exposed by an interleaving in the naïve state space, Iterative Deepening will eventually test an equivalent interleaving which exposes the same bug.*

Proof. By Lemma 1, there must be some equivalent fully-relevant interleaving. By Lemma 2, Iterative Deepening will eventually discover and enable all necessary data-race PPs, with sync API PPs being enabled by the maximal state space assumption. Then DPOR will find the buggy interleaving within this state space. □

6. Proof of malloc-recycle soundness

We seek to prove that ignoring malloc-recycle data race candidates cannot cause DPOR + Iterative Deepening to miss a bug that could be found by using the race as a PP predicate. Our soundness statement is as follows:

Theorem 3 (Soundness of eliminating malloc-recycle races)**.** *If a malloc-recycle data race is not a false positive, DPOR will reorder threads such that either the same accesses will still race without fitting the malloc-recycle pattern, or a use-after-free bug will be reported immediately.*

Though Figures 3 and 4 show example programs, they do not capture all possible cases of how a true data race can fit the malloc-recycle pattern. We proceed by establishing what must be true of any such program, then casing on the ambiguous possibilities, and showing that PPs will exist where we need them to reorder the threads.

For certain, there must be an access in one thread, followed by a free and malloc (we'll call them "middle free" and "middle malloc"), each possibly from either thread, followed by an access from the other. If the data race is not a false positive, then the second access must not change locations based on the middle malloc's return value. WLOG, we say that thread 1 (T1) does the first access, called a_1, and thread 2 (T2) does the second, a_2.

Lemma 3. *If DPOR will reorder a_2 to before a_1, and the location of access a_2 doesn't change, then a non-malloc-recycle data race or a use-after-free bug will be identified.*

Proof. By case on which threads the middle free and middle malloc came from.

- T1 free, T2 malloc (as shown in Figure 3). The malloc will go with a_2 to before the free, and because the allocation of concern has not been freed yet, will return a different value. Hence a_1 and a_2 will be in the same allocation; hence the race is not malloc-recycle anymore.
- T1 free, T1 malloc. Same as above, but the malloc does not move. The middle malloc will still recycle the memory, but a_2 now occurs before then, being in the same, older, allocation.
- T2 free, T2 malloc. Both the free and re-malloc will occur before either a_1 or a_2. The memory will be recycled and both accesses will appear to be in the later allocation.
- T2 free, T1 malloc. The free gets reordered earlier, the malloc stays put, and the accesses go in between. This will be a use-after-free bug.

If either the middle free or middle malloc came from a third thread, the case is the same as if it belonged to T1. □

The keen reader might ask here, what if T1 contains some extra spurious malloc calls (not related to a_1) that affect what T2's malloc returns after being reordered? These could at best either cause `x`'s memory to be recycled differently (not affecting the proof), or not at all (which simply causes immediate use-after-free). In general, extra spurious mallocs that could affect a_1 or a_2 could only convert the program back into a false-positive scenario; and extra spurious synchronization events could only make it more easy to find PPs we need to trigger the reordering. So we can safely assume the only relevant events are the ones we mention explicitly.

By our last assumption, there must also be an "original malloc" which allocated the block to begin with. We must ask, which thread did the malloc which returned a_1's address in the first place? Our last assumption provides that the other thread must obtain that address through some communication mechanism (which we'll reason about later).

6.1 T2 originally malloced x

Lemma 4 (Greedo)**.** *If T2 originally malloced the block containing* x*, DPOR will reorder the threads to uncover a non-malloc-recycle race or a use-after-free bug.*

Proof. Because T1 had the first access, there was a thread switch between the original malloc and a_1, as well as between a_1 and a_2. By Definition 4, each switch will be a PP. By Definition 10, DPOR will reorder a_2 to before a_1, and because T1 is not involved in the logic determining a_2, the access's location stays the same. Lemma 3 finishes. □

This lemma also applies if a third thread was responsible for this malloc, as there would still be a thread switch in the same spot.

6.2 T1 originally malloced x

Lemma 5 (Han)**.** *If T1 originally malloced the block containing* x*, DPOR will reorder the threads to uncover a non-malloc-recycle race or a use-after-free bug.*

Proof. We must guarantee there will be a PP during T1 before its a_1 access, but after whatever action it took to communicate the heap address to T2[6].

If there was a synchronization event between the publish action and a_1, then the maximal state space assumption provides the necessary PP, and we are done. Otherwise, T1's lockset will be the same during publish and during a_1 (and the MHB-ness cannot change). For T1's publish to reach T2, they must access the same memory (*outside* of the block containing a_1; T2 doesn't have that yet), which we'll call p. Hence, p must be a data race of its own.

Because p may also be a malloc-recycle data race, as shown in Figure 6, we do not necessarily get the PP for free. In this case we need to prove that DPOR will likewise reorder any intermediate malloc-recycle pattern to generate the PP we need[7]. We handle this with induction on T2's pointer chain leading to x.

- For the base case, the publish location p_0 is either in global memory, or shared directly using synchronization. Non-heap memory data races are not subject to the malloc-recycle pattern, so will always get a data-race PP, and use of synchronization always gets a PP in the maximal state space.
- For the inductive step, a pointer p_n is published in some heap memory p_{n-1}->ptr, and we assume that however p_{n-1} is shared to T2, there will be a PP there sufficient to make the p_{n-1}->ptr access not malloc-recycle after DPOR. Hence a data race PP will be generated on the p_{n-1}->ptr access, and by Definition 10 and Lemma 3, DPOR will reorder T1's and T2's subsequent accesses to p_n sufficiently to place a PP on them.

[6] Note why we assert the publish action must come before a_1: otherwise, T2 couldn't be reordered to race a_2 with a_1 before T1 communicated the address, and it would be a false positive after all.

[7] In QUICKSAND, data race PPs are not used immediately, but rather generate new state spaces to explore in the future. Anyway, under the maximal state space assumption, we will get to it eventually.

```
Thread 1             Thread 2
p1->ptr = x1;
publish(p1);
free(p1);
x1->foo = ...;
free(x1);
                     p2 = get_published_p();
                     // p's memory recycled
                     q = malloc(sizeof *q);
                     x2 = p2->ptr;
                     // x's memory recycled
                     y = malloc(sizeof *y);
                     x2->foo = ...;
```

Figure 6. If the accesses used to publish x's address are a data race, their PPs may also be eliminated under the malloc-recycle pattern. Induction on the pointer structure leading to x handles this case.

Hence, even if the accesses by which T1 shares x with T2 appear in a different malloc-recycle pattern, a PP will be identified on the publish location p, and DPOR will reorder T2's execution to just after the publish action. As long as T2's execution occurs after the publish, it will receive the same value for its a_2, so the location of the data race does not change. Lemma 3 concludes. □

6.3 Conclusion

Theorem 3 (Soundness of eliminating malloc-recycle races)**.** *If a malloc-recycle data race is not a false positive, DPOR will reorder threads such that either the same accesses will still race without fitting the malloc-recycle pattern, or a use-after-free bug will be reported immediately.*

Proof. Between Lemmas 4 and 5, all cases of possible program structure are covered. □

6.4 Heap overruns

If we relax the "sharing heap addresses" assumption, there is another way to share the allocation's address without T1 and T2 communicating outside of a_1/a_2. One thread can overrun a *different* heap block adjacent to the one containing a_1/a_2 (call them the "neighbour block" and "real block" respectively). Figure 7 shows an example. Heap overrun bugs are quite serious [15], so we do not wish to exclude them from our proof.

In our evaluation, we used the full "sharing heap addresses" assumption to heuristically reduce state space size, skipping reorderings which could only change the addresses

```
   Thread 1                  Thread 2
1                            z = malloc(42);
2                            // TODO bounds check??
3                            x2 = &z[50];
4  x1 = malloc(...);
5  x1->foo = ...;
6  free(x1);
7                            // x's memory recycled
8                            y = malloc(sizeof *y);
9                            x2->foo = ...;
```

Figure 7. Final possibility for how T2 can share T1's allocation address, and probably a security vulnerability to boot!

allocated by malloc. This restricted our tests' scope to exclude such heap-overflow bugs. We consider this justified because recent techniques [12] can find such bugs quickly without the need for data-race PPs. However, for users wishing to test for this class of bug and concurrency bugs simultaneously, we show now how to strengthen the configuration of DPOR to cover the weakened assumption.

Note also that even if malloc-recycle candidates are *not* suppressed, a DPOR which ignores malloc's internal metadata accesses would still be unsound with respect to these bugs. We illustrate this in Figure 8: even with PPs in arbitrary places, the two threads' transitions conflict only on malloc's internal metadata, and hence DPOR would not attempt to reorder them. Hence, our proofs so far show that suppressing malloc-recycle candidates is sound with respect to the classes of bugs which DPOR is already sound to.

To soundly suppress heap-overrun malloc-recycle candidates, we must strengthen our configuration of DPOR as follows: accesses from malloc's internal implementation are not ignored when computing shared memory conflicts, and the synchronization API PP predicates are extended to include the start and end of each malloc and free call.

Lemma 6. *If T1 and T2 each malloced neighbouring blocks, and collided based on pointer arithmetic involving no shared memory accesses, DPOR will reorder the threads to uncover a non-malloc-recycle data race.*

Proof. WLOG, let T1's access in the original malloc-recycle race occur first. We require that our strengthened DPOR will reorder T2's racing access to before T1's, such that both still occur on the same address. There will be a PP in T1's execution between its malloc call and the subsequent racing access. If there are no other memory conflicts between T1 and T2, then by the soundness of DPOR, this PP suffices to reorder without changing the address. Otherwise, let p be the latest conflicting access by T1 before its access a_1. By the same inductive reasoning as we used in Lemma 5, Iterative Deepening will add a data-race PP on p. As T1 has no further conflicts between p and a_1, T2's a_2 will be reordered between them without changing the address. Lemma 3 finishes. □

```
   Thread 1                  Thread 2
1                            z = malloc(42);
2                            // TODO bounds check??
3                            x2 = &z[50];
4                            y = malloc(sizeof *y);
5                            x2->foo = ...;
6  x1 = malloc(...);
7  x1->foo = ...;
8  free(x1);
```

Figure 8. Without a PP between lines 4 and 5 of Figure 7, this is the only alternate interleaving DPOR would explore. The mallocs have been reordered and may no longer collide, which wrongly appears to be a false positive.

```
   Thread 1                  Thread 2
1                            z = malloc(42);
2                            // TODO bounds check??
3                            x2 = &z[50];
4  x1 = malloc(...);
5                            y = malloc(sizeof *y);
6                            x2->foo = ...;
7  x1->foo = ...;
8  free(x1);
```

Figure 9. Goal interleaving of Figure 7. To ensure collision, the sequence of malloc calls producing a_1 and a_2 must not be disrupted compared to the original interleaving.

Theorem 3 is modified to simply include this lemma in addition to Lemmas 4 and 5.

References

[1] B. Blum and G. Gibson. Stateless model checking with data-race preemption points. In *Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, 2016. URL http://www.pdl.cmu.edu/PDL-FTP/associated/oopsla.pdf.

[2] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating Systems Design and Implementation*, OSDI'08, pages 209–224. USENIX Association, 2008.

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

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

[5] 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.

[6] 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.

[7] 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.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[9] 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.

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

[11] 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.

[12] R. Prakash. The holy grail - real time memory access checking. https://blogs.oracle.com/raj/entry/the_holy_grail_real_time, 2015.

[13] 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.

[14] 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.

[15] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *Security and Privacy*, SP '13, pages 48–62. IEEE Computer Society, 2013.

[16] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. URL http://plms.oxfordjournals.org/content/s2-42/1/230.short.

[17] V. V. Vargomax. Generalized Super Mario Bros. is NP-complete. In *Proceedings of the 1st ACH SIGBOVIK Conference in Celebration of Harry Q. Bovik's* 2^6*th Birthday*, SIGBOVIK '07, pages 87–88, Pittsburgh, PA, USA, 2007. ACH.

[18] 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.