Fast Bug Finding in Lock-Free Data Structures with CB-DPOR

by

Jelle van den Hooff

S.B., Massachusetts Institute of Technology, 2013

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014

Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 2014

Certified by . . . .
M. Frans Kaashoek
Professor
Thesis Supervisor

Certified by . . . .
Nickolai Zeldovich
Associate Professor
Thesis Supervisor

Accepted by . . . .
Prof. Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Fast Bug Finding in Lock-Free Data Structures with CB-DPOR

by

Jelle van den Hooff

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2014, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes CB-DPOR, an algorithm for quickly finding bugs in lock-free data structures. CB-DPOR is a combination of the CHESS and DPOR model checking algorithms. CB-DPOR performs similar to the concurrently developed preemption-bounded BPOR algorithm.

CODEX is a tool for finding bugs in lock-free data structures. CODEX implements CB-DPOR and this thesis demonstrates how to use CODEX to find bugs. This thesis describes new bugs in open-source lock-free data structures, and compares the performance of CB-DPOR with the earlier model checking algorithms CHESS, DPOR, and PCT. CB-DPOR find bugs one to two orders of magnitude faster than earlier algorithms.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor

Thesis Supervisor: Nickolai Zeldovich
Title: Associate Professor

Acknowledgments

I would like to thank my advisors, Frans Kaashoek and Nickolai Zeldovich, for their guidance, feedback, and for suggesting up the exciting topic of bug finding in concurrent programs.

I am grateful to my parents, Peter en Henriëtte, and my sister, Andrea, for their support, encouragement, and e-mails throughout the past four years.

Contents

List of Figures

Chapter 1

Introduction

The design and implementation of lock-free data structures is a tricky art. It is difficult to argue about the correctness of the implementation of parallel algorithms. Even a correctness proof of the underlying algorithm is no guarantee for a correct implementation. For example, Chapter 5 describes a bug in an implementation of the proven-correct Michael and Scott queue [10]. The bug involves four different threads and a total of three preemptions, and it is difficult to construct a sequence of events that triggers the bug even knowing what the bug is.

Such bugs are hard to find by stress-testing because the implementations are non-deterministic and the bugs appear with low probability. Model checkers that construct and enumerate thread schedules, or *traces*, have been a successful alternative to stress-testing [6, 7, 14]. This thesis contributes a new model checker called Context Bounded Dynamic Partial-Order Reduction, or CB-DPOR, designed to enumerate a subset of inequivalent traces likely to exhibit bugs.

It is infeasible to enumerate all possible traces for most non-trivial programs. Instead, past work has focused on two main approaches to enumerate fewer traces while still finding bugs. The first approach, found in the original Dynamic Partial-Order Reduction or DPOR algorithm, aims to enumerate only one representative trace for each class of *equivalent* traces that are indistinguishable by the program using partial order reduction techniques [6]. However, the number of inequivalent traces is still too large to enumerate in practice [11].

The second approach instead focuses on enumerating only a subset of traces likely to exhibit bugs, such as traces with a limited number of forced context switches. The CHESS model checker explores only traces where the number of preemptions does not exceed a pre-defined *context bound* [11, 13]. As an example, a trace that briefly runs thread 1, switches to thread 2, back to thread 1 and then finishes thread 2 has a context bound of 2 as the first two context switches are preemptions of threads that later continue running, while the third context switch is inevitable as thread 1 has finished running. When the context bound is increased over time, CHESS's exploration resembles a breadth-first search that quickly covers a diverse set of traces. In practice many bugs can be found with a small number of preemptions [14], and even though CHESS explores many equivalent traces, it successfully finds bugs.

Another algorithm for enumerating traces that are likely to contain bugs is the non-deterministic Probabilistic Concurrent Testing, or PCT, model checker [4]. PCT assigns threads priorities and runs threads strictly according to priority order. To find bugs, PCT randomly picks several priority change points where the current running thread is assigned a lower priority. Because of its sampling approach PCT finds some bugs quickly while missing other bugs completely.

CB-DPOR aims to enumerate CHESS's subset of traces with DPOR's efficiency by combining the two algorithms. The core of CB-DPOR is a combination of CHESS with DPOR. CB-DPOR respects the context bound from CHESS to rapidly explore a large number of interesting traces. At the same time, CB-DPOR inserts preemption points only at memory accesses that can be reordered after a later access by another thread. For example, CB-DPOR will never preempt a thread while it accesses a thread-local variable.

The difficult part in combining CHESS with DPOR is the addition of sleep sets, which are important to DPOR's efficiency [6]. Musavathi and Qadeer have shown sleep sets to be incompatible with CHESS [12]. Sleep sets stop the construction of traces from a given prefix if all traces starting with the prefix can be guaranteed to be equivalent to an already enumerated trace, which requires that the model checker enumerate all possible traces starting with each previous prefix. Yet this assumption does not hold for CHESS combined with sleep sets, so sleep sets will also stop the construction of traces when the equivalent trace has never been enumerated because it exceeded the context bound.

This thesis refines the incompatibility result from Musavathi and Qadeer and contributes preemption-only sleep sets, which are compatible with CB-DPOR by putting threads to sleep only when they get preempted. Using preemption-only sleep sets CB-DPOR never enumerates equivalent traces because of a preemption. The intuition behind preemption-only sleep sets comes from DPOR where all context switches can be thought of as preemptions. In CHESS and CB-DPOR not all context switches are preemptions, and precisely those context switch points violate the sleep sets assumption. Preemption-only sleep sets are compatible with CHESS, CB-DPOR and normal DPOR.

We have implemented CB-DPOR in a tool called CODEX along with implementations of CHESS and PCT. CODEX instruments all memory accesses of a C++ program and allows the algorithms to exactly control the scheduler for deterministic program execution. CODEX also includes a modified linearizability checker from Line-Up [3] to work with PCT, as the original Line-Up algorithm was designed with CHESS and in some cases finds linearizability violations only in traces that are not explored by PCT. We propose a modification to Line-Up so it considers all equivalent traces at the same time and no longer needs the equivalent traces.

We have tested CODEX on random test cases generated by the Line-Up framework for lock-free data structures from CDS [9] and Boost's [1] lock-free library, finding both previously known and unknown bugs. CB-DPOR does so an order of magnitude faster than both CHESS and PCT because it explores far fewer equivalent traces.

In summary, the contributions of this thesis are as follows:

- A trace-enumerating algorithm CB-DPOR combining the two main approaches used for practical software model checking. CB-DPOR was developed in parallel with the Bounded Partial-Order Reduction, or BPOR, model checking algorithm [5]. BPOR combines DPOR with bounds, such as the preemption bound used by CHESS. Algorithmically, preemption-bounded BPOR and CB-DPOR function almost the same.

- An extension of the Line-Up linearizability checking algorithm that checks all equivalent traces at the same time for compatibility with CB-DPOR.

- Implementations of CB-DPOR, CHESS, PCT and the extended Line-Up linearizability checker in the testing framework CODEX.

- A detailed evaluation comparing the performance of CB-DPOR, CHESS and PCT.
- Previously unknown bugs in `libcds`, an open-source concurrent data structure library.

The rest of this thesis is organized as follows. First, a description of previous work and a formal description of key terms in Chapter 2. Then Chapter 3 contains a detailed description of CB-DPOR along with pseudocode. CODEX's implementation is described in Chapter 4. Chapter 5 describes a bug reproduced using CODEX. In Chapter 6 we evaluate and compare CB-DPOR with earlier model checkers.

Chapter 2

Background

CB-DPOR builds on the algorithms and definitions from CHESS [14] and DPOR [6]. In this chapter we provide the relevant definitions, as well as brief summaries of the internals of different trace enumeration algorithms.

2.1 Partial-order reduction

Partial-order reduction techniques aim to reduce the number of different interleavings explored, skipping interleavings where a program's output will be the same. To reason about a program's result we need a formal view of program execution. The definitions used in this thesis are based on the definitions used by DPOR.

A program consists of several *threads* that perform local operations and shared-memory accesses. All local operations are assumed to be deterministic, so that a model checker can instead focus on the shared-memory operations. All memory accesses are assumed to be linearizable so that a program's execution summarizing the interleaving of all threads is a linear history of memory accesses or *transitions*. A transition can be a simple memory read or write, or a more complex atomic increment, compare-and-swap or a lock acquire. The sequence of transitions executed by a program is a *trace*.

Some model checkers, including CB-DPOR, do not fairly schedule threads. To prevent a thread from spinning forever under unfair schedules, transitions can have a programmer-supplied runnability predicate. For example, a lock acquire implemented using a compare-

and-swap should be *runnable* only if the compare-and-swap will succeed. A useful follow-up definition is to consider a thread *runnable* if and only if its next transition is runnable.

The relative orders of many transition pairs, such as writes to different memory addresses, do not influence a program's result. Two transitions are *independent* if their relative order never changes the result of any transition. All other transition pairs are called *dependent*. Two transitions by the same thread are always dependent. For any two dependent transitions in a trace, the later transition *depends* on the earlier transition.

Two traces are *equivalent* if they can be transformed into each other by reordering adjacent independent transition pairs. Because all transitions have the same result in two equivalent traces, the program's outcome must be the same in both traces, and so it suffices to enumerate only *inequivalent* traces to enumerate all possible outcomes.

To generate inequivalent traces it is useful to know if two transitions in a trace could possibly be reordered. Although two independent transitions can always be reordered if they are adjacent, that is not necessarily true if they are further apart, as they might both be dependent with an intermediate transition which imposes a relative ordering. In that case, the later transition transitively depends on, or *happens after*, the earlier transition and cannot be reordered.

2.2 Context-bounded model checking

One way of exploring traces likely to contain bugs is to limit the number of preemptions in each trace, as done by CHESS. A *context switch* in a trace happens whenever two consecutive transitions come from different threads. A *preemption* is a forced context switch where the original thread remains runnable after the first transition. Context-bounded model checking in CHESS and CB-DPOR limits the number of preemptions in each trace.

Besides practical evidence that context-bounded exploration finds many bugs, context-bounded exploration also provides a theoretical upper-bound on the number of traces that must be explored: since the total number of traces with a fixed preemption limit grows polynomially in program length, all traces up to a fixed preemption limit can be explored in polynomial time.

To find bugs as quickly as possible, CHESS and CB-DPOR use *iterative deepening* and increase the preemption limit after each successive run of the algorithm. Using iterative deepening CHESS and CB-DPOR can quickly find bugs with a low preemption count, and provide coverage guarantees after each run in case there are no bugs.

2.3 Enumerating traces

From a low-level point of view, a model checker has only limited knowledge of and control over a tested program: the model checker knows each threads' next transition, can decide the next transition to run, and no more. To systematically enumerate traces, CB-DPOR, CHESS and DPOR all have a recursive exploration function that constructs traces as if they were exploring a tree. These exploration functions start with an empty trace *prefix*, decides what thread to run next, extend the prefix by one thread, and recurse. The algorithms differ in how they decide what threads to run.

CHESS. The CHESS algorithm runs all threads at each prefix unless running a thread would violate the preemption limit. CHESS does not consider dependencies between transitions.

DPOR. The dynamic partial order reduction algorithm attempts to run only the threads necessary to explore all inequivalent traces. Initially, DPOR picks an arbitrary transition to run at each prefix. DPOR observes that there is no need to run other transitions at that prefix unless there is a later transition dependent with the original transition that could be reordered with the original transition. Only if such a later transition is found will DPOR's exploration algorithm run other threads at a prefix.

PCT. Instead of performing a structured exploration, PCT [4] randomly samples traces and stores no state during its search. For each sample, PCT picks a number of priority switch points which are positions in the trace where the current thread's priority is adjusted. Then PCT randomly assigns starting priorities, and constructs a trace solely based on its scheduler. PCT guarantees a lower-bound probability of finding a bug depending on the program length and number of context switch points.

2.4 Sleep sets

Sleep sets are an important optimization to DPOR [6]. Where DPOR carefully attempts to introduce preemptions only when truly necessary, sleep sets make sure that introduced preemptions actually lead to new, inequivalent traces.

When preempting a thread, DPOR adds the preempted thread to the sleep set. Threads in the sleep set are not allowed run until removed from the sleep set, and threads are removed only when a transition dependent with the thread's next transition executes.

If DPOR were to execute a thread still in the sleep set, the resulting trace would be equivalent to an already explored trace: the sleeping thread's next transition could be moved back in the trace until right after the original prefix because there would be no dependent transitions in between, and DPOR has already explored all traces with the sleeping thread's transition right after the prefix!

This reordering argument requires the model checker to have explored all possible reordered traces starting with the original prefix. CHESS does not explore all traces starting with each prefix because of the preemption limit, and so CHESS is not compatible with sleep sets as described above [12].

2.5 Bounded partial-order reduction

The BPOR model checker combines DPOR with sleep sets with bounded exploration, such as a context bound [5]. When BPOR uses a preemption bound, the model checker handles conflicting memory accesses differently from DPOR. Like DPOR, BPOR schedules the exploration of a trace where the original memory access executes after the later conflict. Unlike DPOR, BPOR also schedules the exploration of a trace where the entire original thread executes after the later conflict to not increase the number of preemptions. BPOR uses sleep sets only for preemptions introduced by the original DPOR algorithm.

In practice, CB-DPOR functions similar to BPOR with a preemption bound. Like BPOR, CB-DPOR uses sleep sets only for preemptions. CB-DPOR inserts preemptions after finding conflicting memory accesses, just like BPOR.

Chapter 3

Design

In this chapter we present the CB-DPOR algorithm with pseudocode and introduce preemption-only sleep sets compatible with CB-DPOR. An example exploration of a simple program with three threads by CB-DPOR can be found in Figure 3-1.

3.1 CB-DPOR

CB-DPOR attempts to enumerate inequivalent traces with a limited number of preemptions. CB-DPOR's main mechanism for exploring different traces is the introduction of *context switch points*. After a prefix that is marked as a context switch point, CB-DPOR will explore every runnable thread. There are two types of context switch points: those explicitly introduced by later dependencies, and those implicitly introduced by the trace. Implicit context switch points are introduced whenever the previous thread is no longer runnable, because it ended or blocked, and at the beginning of the trace.

In the example exploration in Figure 3-1 the first implicit context switch point is the leftmost vertical line at the beginning of the program. Other implicit context switch points can be found in trace (a) when thread A finishes running and threads B and C are attempted. A context switch point might consider only one thread, such as in trace (a) after thread B finishes and only C is left.

Whenever a transition depends on an earlier transition, CB-DPOR explicitly introduces a preemption as a new context switch point. The goal of such a preemption is to construct a

```
Thread A:        Thread B:        Thread C:
w=1              y=z              z=4
x=2              x=y
y=3
```

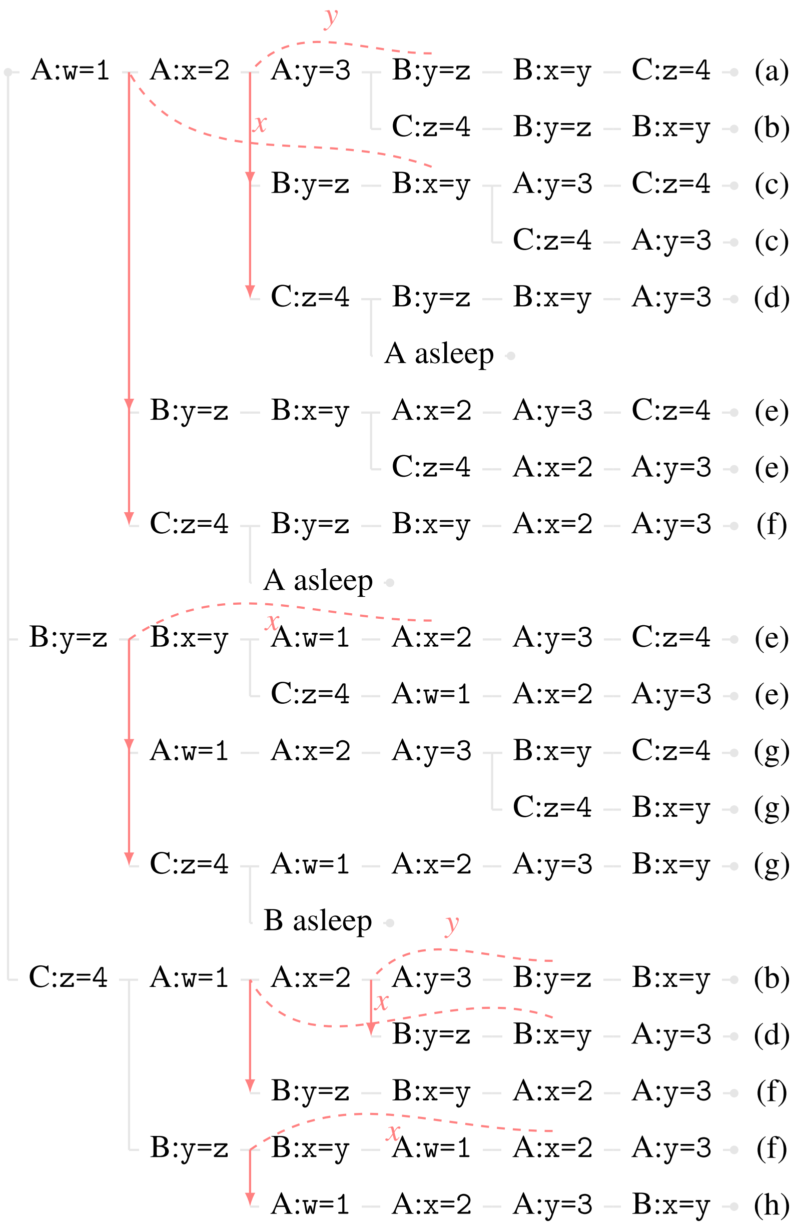
y
A:w=1 — A:x=2 — A:y=3 — B:y=z — B:x=y — C:z=4 (a)
x
C:z=4 — B:y=z — B:x=y (b)
B:y=z — B:x=y — A:y=3 — C:z=4 (c)
C:z=4 — A:y=3 (c)
C:z=4 — B:y=z — B:x=y — A:y=3 (d)
A asleep
B:y=z — B:x=y — A:x=2 — A:y=3 — C:z=4 (e)
C:z=4 — A:x=2 — A:y=3 (e)
C:z=4 — B:y=z — B:x=y — A:x=2 — A:y=3 (f)
A asleep
x
B:y=z — B:x=y — A:w=1 — A:x=2 — A:y=3 — C:z=4 (e)
C:z=4 — A:w=1 — A:x=2 — A:y=3 (e)
A:w=1 — A:x=2 — A:y=3 — B:x=y — C:z=4 (g)
C:z=4 — B:x=y (g)
C:z=4 — A:w=1 — A:x=2 — A:y=3 — B:x=y (g)
B asleep
y
C:z=4 — A:w=1 — A:x=2 — A:y=3 — B:y=z — B:x=y (b)
x
B:y=z — B:x=y — A:y=3 (d)
B:y=z — B:x=y — A:x=2 — A:y=3 (f)
x
B:y=z — B:x=y — A:w=1 — A:x=2 — A:y=3 (f)
A:w=1 — A:x=2 — A:y=3 — B:x=y (h)

Figure 3-1: A simple program along with its exploration by CB-DPOR. All preemptions are marked with red arrows and have a dotted line connecting the dependent transition to the preemption point. Each finished trace is labeled; equivalent traces have the same label.

trace in which the later dependency runs before the preempted transition. Such a preemption is introduced only if the later dependency can be reordered with the earlier transition.

An example of a preemption can be found in (a), introduced by the dependency B:y=z on A:y=3. In the traces (c) and (d) constructed after that preemption B:y=z executes before the preempted transition A:y=3. Note that in trace (a) the dependency of B:x=y on A:x=2 does not yet introduce a context switch point before transition A:x=2, as they are both ordered by the intermediate transition A:y=3.

3.2 Preemption-only sleep sets

CB-DPOR uses preemption-only sleep sets instead of DPOR's sleep sets. Like DPOR, CB-DPOR removes threads from the sleep set whenever it executes a dependent transition, but unlike DPOR, CB-DPOR adds threads only after a preemption, and not at other context switch points. The idea behind preemption-only sleep sets is that CB-DPOR can guarantee that a sleeping preempted transition can be moved backwards in the trace to the point where it was preempted and added to the sleep set without increasing the number of preemptions.

Preemption-only sleep sets allow CB-DPOR to skip an equivalent trace while exploring (f) in the example. CB-DPOR does not run thread A after C: z=4, as A was preempted and not yet removed from the sleep set. If CB-DPOR were to execute A:x=2 before the conflict B:x=y, the preempted A:x=2 could be moved backwards in the trace without increasing the total number of preemptions, and result in a trace equivalent to either (b) or (d).

CB-DPOR does not add the first transition A:w=1 to the sleep set, even though it tries other transitions in its place. If CB-DPOR were to add the transition to the sleep set, then CB-DPOR would not explore the trace (h): although (h) could start with A:w=1, that would require adding a preemption and violating the preemption bound.

3.3 Pseudocode

Pseudocode for CB-DPOR can be found in Figure 3-2. The main recursive search function is Explore(prefix, sleepset, preemptions). Explore starts with an empty prefix, an empty

```
function ConsiderPreemptions(prefix, thread)
    foreach $t \in$ prefix dependent with Next(thread) do
        if thread's last transition does not happen after t
        then
            Mark the prefix ending with t

function Attempt(prefix, sleeping, preemptions, thread)
    Remove threads whose next transition is dependent with
        Next(thread) from sleeping
    ConsiderPreemptions(prefix, thread)
    Explore(prefix + Next(thread), sleeping, preemptions)

function Explore(prefix, sleeping, preemptions)
    $l \leftarrow$ the last thread in prefix
    if $l$ is still runnable then
        Attempt(prefix, sleeping, preemptions, $l$)
        if prefix has been marked and preemptions < limit
        then
            sleeping $\leftarrow$ sleeping $\cup \{l\}$
            preemptions $\leftarrow$ preemptions $+ 1$
            foreach $T \in$ threads $-$ sleeping do
                Attempt(prefix, sleeping, preemptions, $T$)
    else
        foreach $T \in$ threads $-$ sleeping do
            Attempt(prefix, sleeping, preemptions, $T$)
```

Figure 3-2: Pseudocode for CB-DPOR.

sleep set, and a count of zero preemptions.

Explore first determines if it needs to introduce an implicit context switch point, and if so it runs all threads not in the sleep set (this is the lower branch). Otherwise, Explore initially attempts only the previous thread.

Explore extends a prefix by a thread by calling Attempt(prefix, sleeping, preemptions, thread). First, Attempt updates the sleep set by removing dependencies of the next transition. Then Attempt considers the introduction of explicit preemption context switch points: all dependencies of the next transition that can be reordered have their prefix marked. Finally, Attempt calls Explore to recursively explore the new prefix.

Whenever a recursive search of prefix with no implicit context switch point finishes, Explore checks if the prefix has been marked by a later transition. When Explore finds a conflict, and the number of preemptions does not yet exceed the limit, Explore introduces an explicit context switch point and runs all available threads. Explore ignores marks on prefixes with implicit context switch points.

3.4 Linearizability testing

By themselves, model checkers that enumerate different execution traces will trigger bugs, but not detect them: although a program might return an invalid result, a model checker such as CB-DPOR has no knowledge of correct or incorrect results. Instead, CB-DPOR by itself will find only crash-bugs and deadlocks.

To find all concurrency bugs using CB-DPOR, CODEX use a modified version of the Line-Up linearizability checker [3] to test lock-free data structures. Line-Up takes traces generated by a model checker, and checks them for linearizability. A limitation of the original Line-Up algorithm is that it does not guarantee the same result on two equivalent traces. CODEX includes a modified version of Line-Up that considers all equivalent traces at the same time.

To test for linearizability, Line-Up groups actions by threads in function calls. In a trace these functions might run concurrently, for example, if CB-DPOR preempts individual functions. However, for a trace to be linearizable, there must be another trace in which the individual functions do not run concurrently, yet return the same results. Line-Up exhaustively enumerates different orderings of the function calls, or linearizations, to find a *correct* linearization with the same results as in the trace.

Not all orderings of function calls are valid linearizations: for example, if two functions do not run concurrently, so that the second the function starts in the trace after the first one ends, then any valid linearization must respect that ordering.

The original Line-Up enforces such a relative order between two function calls only if the earlier function finishes completely before the second function starts. However, this means that Line-Up has different behavior on equivalent traces. For example, consider the two equivalent traces labeled (c) in Figure 3-1. If the entirety of thread A and thread C are each a single function call, then Line-Up would force C to run after A in the first trace (c), yet not enforce any ordering in the second trace (c). This might hide a bug, if running C before A gives a linearization with the same outputs.

The original Line-Up misses some bugs because Line-Up does not consider the actual the dependencies between the two threads, instead considering only when the threads' transitions run. CODEX's modified version of Line-Up considers the dependencies between threads, and allows two threads to be reordered only if they both have a dependency on each other. Additionally, if there are no dependencies between two threads, CODEX requires that both relative orderings are correct. Because dependencies are the same for equivalent traces, CODEX will not miss linearizability violations.

For example, for trace (c), CODEX will allow A and B to be reordered, as B: `x=y` happens after A:`y=3`, and A:`y=3` happens after B:`x=y`. At the same time, C must run after B as C:`z=4` happens after B:`y=z`, and no transition of B happens after a transition of C. Finally, in a correct linearization, changing the relative order of A and C should not make the linearization incorrect, as A and C have no dependent transitions.

Chapter 4

Implementation

We have implemented CB-DPOR and other model-checking algorithms in CODEX to evaluate CB-DPOR's performance and to find bugs in real-world lock-free data structures. In this chapter we describe the implementation of CODEX as well our extended version of Line-Up to find linearizability bugs.

4.1 Controlling trace execution

CODEX bridges the gap between the interface expected by CB-DPOR and other algorithms to manipulate program execution and the reality of unmodified C++ code. Model checkers operate on an abstract program that consists of multiple threads whose state consists of an extensible prefix of transitions executed so far.

CODEX provides this API by intercepting all memory accesses, exposing them as transitions, and carefully controlling the scheduler to execute the desired transitions. Using the LLVM compiler framework, CODEX replaces all memory accesses in the target program with calls to the CODEX scheduler. The CODEX scheduler pauses the calling thread and records the upcoming transition. Once the scheduler has collected upcoming transitions for all threads, it lets the model checker decide what thread to run next.

CODEX does not intercept library calls, such as calls to a mutex primitive. Instead, CODEX provides several such primitives using compare-and-swaps. Lock-free data structures usually use few existing primitives, and so CODEX's implementation suffices for the tested libraries. Programmers can supply a runnability predicate through a RequireResult function. For example, Codex's lock is implemented with RequireResult:

```
void Acquire() {
  bool old = false;
  RequireResult(false);
  while (!held.compare_exchange_weak(old, true)) {}
}
```

When run normally, the while loop will spin until the lock is released. When run under Codex, the loop never spins: Codex knows that the function will progress only when the held variable is false, and will not run the Acquire function until the lock is available.

4.2 Generating test cases

To find bugs using Codex we generated random programs consisting of 3 threads with 3 function calls per thread, as described in the Line-Up paper. After Codex found a linearizability violation, we manually constructed a smallest reproducing test case by removing function calls from the initial test case until the test case no longer caused a linearizability violation. Afterwards, we inspected the trace to find the cause of the linearizability violation.

4.3 Efficient implementation

CB-DPOR has two possibly computationally expensive parts: finding all dependent transitions in ConsiderPreemptions, and determining if two transitions happen after each other. Both can be implemented efficiently using the techniques from DPOR [6], so that the overhead per executed transition is linear in the number of threads.

Codex limits transitions to memory accesses that modify only a single location in memory. Because of this simplification it is easy to find all dependent transitions that a thread has not yet happened after: for each memory location, we keep a list of memory reads and all memory writes, and an index in both lists for each thread indicating the latest seen transition. A new transition then needs to consider only transitions not considered before, and the amortized runtime for ConsiderPreemptions is linear in the number of threads. This optimization is similar to the stack traversal optimization as used by DPOR.

The happens-before relation is maintained using vector clocks [8], stored for each thread, each memory address, and each past transition in the prefix. Updating vector clocks requires time linear in the number of threads for each transition, and comparing two vector clocks runs in constant time, again similar to DPOR.

All operations on sets of threads are implemented using bit sets: CODEX allows a maximum of 64 threads, and encodes sets of threads as 64 bit integers, so all common set operations run in constant time.

Chapter 5

Example bug

CODEX reproduced a bug in a development version of Boost's lock-free queue [1], which is an implementation of the Michael and Scott queue [10]. Figure 5-1 contains the implementation of the enqueue and dequeue function, and Figure 5-2 shows a trace triggering the bug. The trace contains 5 threads, with three threads performing an enqueue, and two threads performing a dequeue. The output of CODEX normally lists addresses instead of variable names (as CODEX's instrumentation works on addresses). However, CODEX does list the line that caused each write, allowing a careful programmer to determine what variables were written, and reconstruct a trace as in Figure 5-2.

In this specific trace, Boost's FIFO accidentally resets the pointer version of A.NEXT in step 30. Boost's FIFO uses tagged pointers to prevent compare-and-swaps from spuriously succeeding; however, the node allocator contained a bug that reset the tag on the NEXT pointer, as it invoked the default constructor which reset all fields to zero. In Figure 5-2 a pointer's tag is written in subscript, so that A_1 represents a pointer to A with tag 1. On a high level, the problem in this trace is as follows: first, thread 0 starts an enqueue and plans on attaching its node to node A, by reading A.NEXT on line 3. However, before it can change A.NEXT. Then, thread 2 starts another enqueue, and attaches its node B to A on step 11. Now thread 3 performs a dequeue, removing A as it is the first element in the list. It stores A for reuse on step 26. Then thread 1 starts an enqueue and retrieves A on step 27. Before thread 1 uses A, it first calls the constructor, spuriously resetting the tag on A.NEXT on step 30. Then thread 1 is preempted, and now thread 0 attaches its node, C, to A on step 35. Thread

0 does not realize that *A* has changed while it was preempted; from thread 0's point of view, *A*.NEXT has been NULL_0 the whole time. Then thread 0 completes, and thread 4 begins a dequeue. This dequeue fails, because although thread 0 has added a new node to *A*, the node *A* is not yet part of the queue. This is a linearizability violation, as thread 0 has successfully enqueued 1234, no one has dequeued 1234, and yet thread 4 believes the queue is empty.

This example illustrates that CODEX can reproduce well-hidden bugs, and that even with a powerful model checker, actually understanding the buggy line of code is still a challenge, as lock-free data structures have complicated execution traces.

```
void enqueue(T const & t) {
    node * n = alloc_node(t);
    for (;;) {
        atomic_node_ptr tail (tail_);
        memory_barrier();
        atomic_node_ptr next (tail->next);
        memory_barrier();
        if (likely(tail == tail_)) {
            if (next.get_ptr() == 0) {
                if ( tail->next.CAS(next, n) ) {
                    tail_.CAS(tail, n);
                    return;
                }
            } else tail_.CAS(tail, next.get_ptr());
        }
    }
}

bool dequeue (T * ret) {
    for (;;) {
        atomic_node_ptr head (head_);
        memory_barrier();
        atomic_node_ptr tail(tail_);
        node * next = head->next.get_ptr();
        memory_barrier();
        if (likely(head == head_)) {
            if (head.get_ptr() == tail.get_ptr()) {
                if (next == 0) return false;
                tail_.CAS(tail, next);
            } else {
                if (next == 0) continue;
                *ret = next->data;
                if (head_.CAS(head, next)) {
                    dealloc_node(head.get_ptr());
                    memory_barrier();
                    return true;
                }
            }
        }
    }
}
```

Figure 5-1: Enqueue and dequeue code from Boost's FIFO.

Thread 0 starts ENQUEUE(1234):

1. Read FREELIST = NULL$_0$
2. Read TAIL = A_0
3. Read A.NEXT = NULL$_0$
4. Read TAIL = A_0

Thread 0 is preempted, and thread 2 starts ENQUEUE(1236):

5. Read FREELIST = NULL$_0$
6. Write B.NEXT = NULL$_0$
7. Write B.VALUE = 1236
8. Read TAIL = A_0
9. Read A.NEXT = NULL$_0$
10. Read TAIL = A_0
11. Swap A.NEXT from NULL$_0$ to B_1

Thread 2 is preempted, and thread 3 starts DEQUEUE:

12. Read HEAD = A_0
13. Read TAIL = A_0
14. Read A.NEXT = B_1
15. Read HEAD = A_0
16. Swap TAIL from A_0 to B_1
17. Read HEAD = A_0
18. Read TAIL = B_1
19. Read A.NEXT = B_1
20. Read HEAD = A_0
21. Read B.VALUE = 1236
22. Swap HEAD from A_0 to B_1
23. Read FREELIST = NULL$_0$
24. Read A.NEXT = B
25. Write A.NEXT = NULL$_1$
26. Swap FREELIST from NULL$_0$ to A_1

Thread 3's DEQUEUE returns 1236, and thread 1 begins ENQUEUE(1235):

27. Read FREELIST = A_1
28. Read A.NEXT = NULL$_1$
29. Swap FREELIST from A_1 to NULL$_2$
30. **Write A.NEXT = NULL$_0$**
31. Read TAIL = B_1
32. Read B.NEXT = NULL$_0$
33. Read TAIL = B_1

Thread 1 is preempted, and thread 0 continues:

34. **Swap A.NEXT from NULL$_0$ to C_1**
35. Failed to swap TAIL from A_0 to C_1; was B_1

Thread 0's ENQUEUE finished, and thread 4 begins DEQUEUE:

36. Read HEAD = B_1
37. Read TAIL = B_1
38. Read B.NEXT = NULL$_0$
39. Read HEAD = B_1

Thread 4's DEQUEUE failed, and thread 1 continues:

40. Swap B.NEXT from NULL$_0$ to C_1
41. Swap TAIL from B_1 to C_2

Thread 1's ENQUEUE finished, and thread 2 continues:

42. Failed to swap TAIL from A_0 to B_1; was C_2

Thread 2's ENQUEUE finished, and the program is finished.

Figure 5-2: Trace with bug in Boost's FIFO, an implementation of the Michael and Scott queue, with 5 threads. Although thread 0 finishes before 4 starts, and 1234 thus must be enqueued, thread 4 fails to dequeue any element from the list. Boost's FIFO uses tagged pointers; each pointer's version is noted in subscript. This implementation incorrectly resets the version of A.NEXT on step 30.

Chapter 6

Evaluation

This chapter evaluates CB-DPOR experimentally to answer the following questions:

- Does CB-DPOR find bugs, and, in particular does it find new bugs?
- Is CB-DPOR faster than other model checkers?
- Is CB-DPOR's design the reason that it finds bugs faster?

6.1 Experiment setup

To evaluate CB-DPOR's performance we used CODEX to test several pre-existing lock-free C++ data structures. The majority of the data structures come from the open-source Concurrent Data Structures library `libcds` 3.1 [9]. The library is an interesting target to test because it has been under development since 2007, had not exhaustively been tested with model checkers before, and is based on proven-correct data structures.

CODEX does not wrap higher-level constructs such as the `pthread` library because we assume that lock-free code provides its own implementation. To test `libcds` we implemented a simple lock and thread-local storage abstraction using built-in atomics.

We also test an old version of Boost's lock-free queue, as mentioned in chapter 5.

Another tested data structure is a development version of `crange`, an attempt at a lock-free range set from `sv6` [2]. In some interleavings a logic bug would cause `crange` to lose elements.

	Guaranteed number of bugs							Expected number of bugs		
Data structure	**CB-DPOR**	**DPOR**	**CHESS**	**-atomic**	**PCT 3**	**PCT 5**	**PCT 7**	**PCT 3**	**PCT 5**	**PCT 7**
`boost fifo`	2/2	1/2	2/2	1/2	2/2	2/2	2/2	2.00/2	2.00/2	2.00/2
`cds basketqueue`	2/2	0/2	2/2	2/2	2/2	2/2	2/2	2.00/2	2.00/2	2.00/2
`cds lazylist`	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1.00/1	1.00/1	1.00/1
`cds moirqueue`	2/2	1/2	2/2	2/2	2/2	2/2	2/2	2.00/2	2.00/2	2.00/2
`cds msqueue`	3/3	1/3	3/3	3/3	3/3	3/3	3/3	3.00/3	3.00/3	3.00/3
`cds optimisticqueue`	2/2	0/2	0/2	0/2	0/2	0/2	0/2	0.44/2	0.82/2	0.91/2
`cds skiplistset`	3/3	1/3	0/3	0/3	1/3	1/3	1/3	1.18/3	1.52/3	2.15/3
`crange`	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2.00/2	2.00/2	2.00/2
`refcache`	1/1	1/1	1/1	0/1	0/1	1/1	1/1	0.00/1	1.00/1	1.00/1
	18/18	8/18	13/18	11/18	13/18	14/18	14/18	13.62/18	15.34/18	16.06/18

Figure 6-1: The number of test cases out of the total number of test cases for each data structure in which model checkers found bugs. PCT is non-deterministic; a bug is counted as guaranteed if PCT finds the bug with probability at least 99%. The bugs in `libcds` were not previously known.

The last tested data structure is `refcache`, a conflict-free reference counter from `sv6`. The published version `refcache` can incorrectly free an element.

Generating test cases CODEX uses Line-Up's test case generation algorithm to create testable programs for each data structure. We test the core methods exposed by each data structure, such as queue, enqueue and empty for queues and stacks, as well as insert, delete and find for sets. Some data structures also expose other methods, but they are mostly implemented as a series of calls to already tested core methods and thus not linearizable.

Running codex CODEX is aimed at programmers that quickly want to find bugs in their parallel code. We assume that a programmer is willing to wait 30 minutes to run a test. If no bug is found after half an hour, we stop the test case. Without a time limit, all model checkers could run for years on certain bug-free test cases.

All our benchmarks ran on an 80 core machine with eight 2.4 GHz 10 core Intel E7-8870 chips and 256 GB of RAM, running 80 tests in parallel with a single thread for each model checker.

6.1.1 Model checker configuration

The CODEX implementations of CB-DPOR, PCT, CHESS all require some configuration. We have configured each algorithm to find bugs as quickly possible, and provide results for multiple alternatives if we could not find a single best configuration.

PCT. PCT's only configuration option is the number of priority switch points it introduces in each trace. We have not found an optimal number of priority switch points for all our test cases, so we test three representative configurations of PCT with 3, 5 and 7 priority switch points. Because of PCT's non-deterministic nature, we ran PCT for several hours to accurately determine the probability that PCT would find a bug in 30 minutes, as well the expected time to find a bug.

CB-DPOR and CHESS. Both CB-DPOR and CHESS run in iterative-deepening mode to quickly find bugs with a low preemption bound and to provide actual coverage guarantees after running for 30 minutes.

CHESS-atomic CODEX also runs CHESS modified to branch only on atomic instructions as mentioned in CHESS's original description. This modified version of CHESS runs faster, but misses some bugs due to the difficulty in identifying atomic instructions in C++ programs.

6.1.2 Metrics

We use two metrics to compare the algorithms. The first metric is the time until a bug is found. If no bug is found within 30 minutes, we stop the test case. If there is no bug, all algorithms could run almost indefinitely.

The second metric is the efficiency of the algorithms when exploring up to a specific context bound on cases with and without bugs. Efficiency of an algorithm is measured as the number of distinct traces it explored out of all the traces explored.

6.2 Does CB-DPOR find bugs?

For each data structure we generated three test cases with Line-Up. A summary of the bugs found for each data structure can be found in Figure 6-1. For each data structure and model checker the table lists the number of test cases in which the model checker found a bug, out of the total number of test cases with at least one bug. CB-DPOR is the only model checker that reliably finds bugs in all test cases. CB-DPOR reproduces the known bugs in `crange` and Boost's queue, and finds several new bugs in `libcds`.

DPOR finds bugs in only 8 out of 17 test cases because it does not have enough time to explore all inequivalent traces, and spends time enumerating traces without bugs. Both CHESS and CHESS-atomic find bugs on more test cases, though not on the same: CHESS-atomic misses one `boost fifo` test case while CHESS misses a `crange` test case.

PCT finds 14 out of 18 test cases reliably, and might find more: the expected number of bugs found is slightly higher.

6.3 How fast is CB-DPOR?

We compare the performance of model checkers by measuring the time it takes for a model checker to find a bug. For all data structures except `refcache`, the test cases are numbered 1 to 3. Relative runtime for all algorithms compared to CB-DPOR on all test cases with a bug, as well as absolute runtimes for CB-DPOR, can be found in Figure 6-2. The runtime for PCT is the expected runtime, calculated using the average time to run a trace and the probability of a trace finding a bug after running for several hours. All deterministic model checkers are stopped after 30 minutes.

We do not compare runtime on test cases without bugs because the algorithms have no measurable ending point on such cases. Instead, we compare efficiency in Section 6.4.

In almost all cases CB-DPOR is faster than the other algorithms. In the `optimistic-queue` and first two `skiplistset` cases CB-DPOR is the only algorithm expected to find the bug within 30 minutes. In the `boost fifo 2` case the non-deterministic PCT algorithms are faster CB-DPOR. In the `skiplistset` 3 case DPOR is faster than CB-DPOR. CB-DPOR is designed to quickly explore relevant sub traces, and sometimes this exploration is unlucky.

CB-DPOR's performance gain is bigger for the `libcds` data structures than for `boost fifo` or for `crange`. This is because `libcds`'s data structures are more heavyweight than the other data structures: for example, the garbage collection in `libcds` uses hazard pointers that require significant maintenance, while `crange` and `boost fifo` provide no garbage collection. These more complicated programs have longer traces where CB-DPOR's efficiency becomes valuable.

In a couple of cases the modified version of CHESS that branches only on atomic instructions is faster than the version of CHESS branching on all instructions, because it skips traces that in some cases do not contain a bug. In other cases however, the modified version misses bugs because of this optimization. Because CB-DPOR tracks which memory accesses lead to conflicts, CB-DPOR correctly introduces context switch points only when

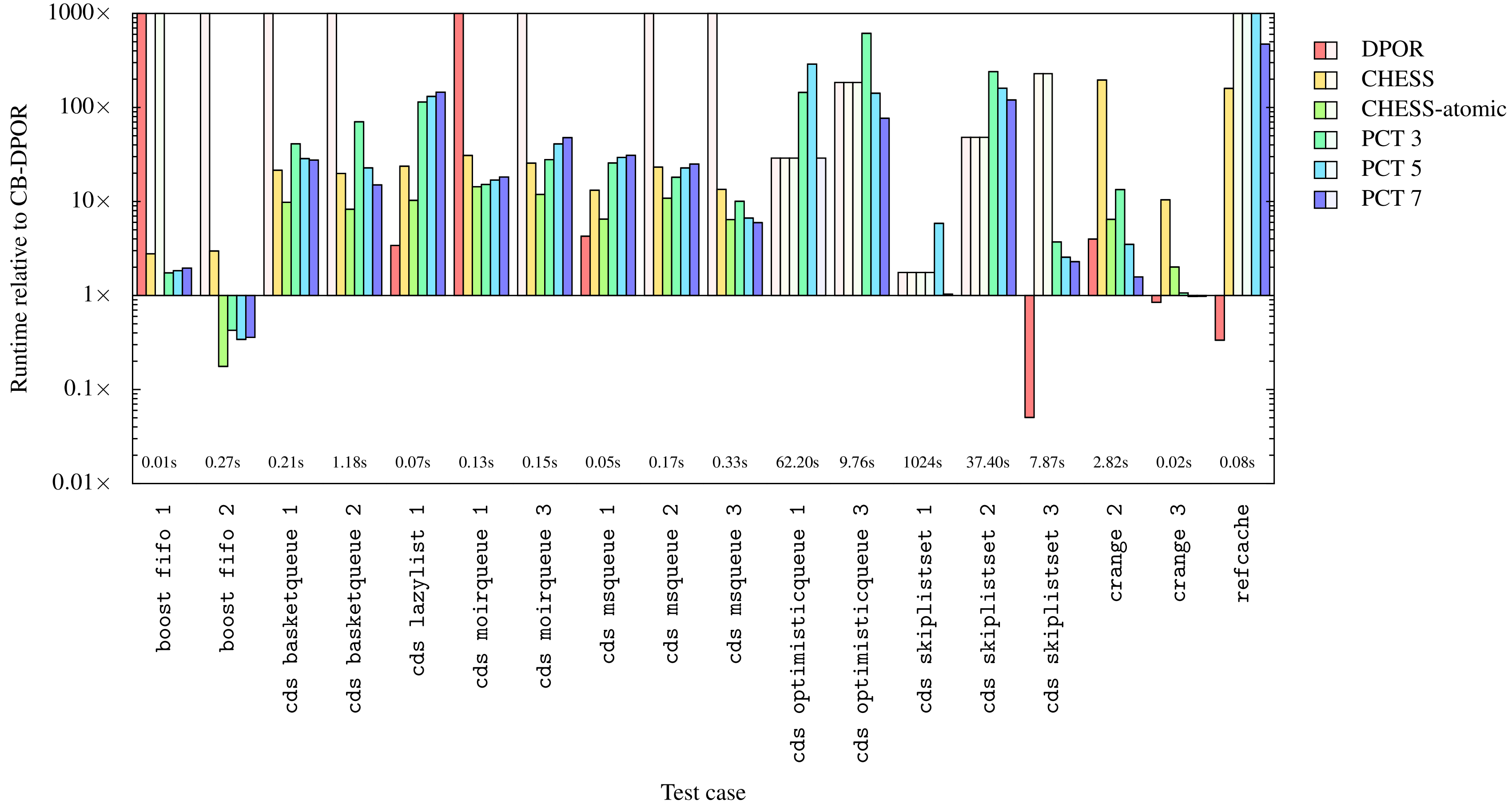

Figure 6-2: Time until bug found for previous algorithms compared to CB-DPOR on several randomly generated test cases for each data structure. The absolute runtime for CB-DPOR is shown at the bottom. If a bug is missed by an algorithm its bar is lightly shaded and extends to the 30 minute mark.

necessary, and can skip context switch points on atomic instructions that lead to conflicts that CHESS atomic cannot.

6.4 Why does CB-DPOR find bugs faster than other model checkers?

CB-DPOR is designed to combine the targeted search of CHESS with the efficiency of DPOR, and so CB-DPOR quickly explores a relevant set of traces without wasting time on the exploration of equivalent traces. To evaluate the efficiency of CB-DPOR, we measured the number of inequivalent traces explored as a function of the total number of traces explored.

Figure 6-3 illustrates the exploration behavior over time of CB-DPOR on the test case `cds skiplistset 2`, where all model checkers are run for a total 75000 traces. The total number of inequivalent traces varies significantly: CHESS explores fewer than 500 inequivalent traces, while CB-DPOR enumerates almost 65000 inequivalent traces. An optimal model checker would explore no equivalent traces, indicated by the red line in the graph. In this example, CB-DPOR is closer to optimal than DPOR; in some cases, DPOR cannot finish a trace when all runnable threads are sleeping, which we count as equivalent traces. Although DPOR is more efficient than PCT, DPOR does not find the bug, while all instances of PCT do.

The CHESS and PCT algorithms do not attempt to explore inequivalent traces, resulting in a large number of equivalent traces explored. CHESS and CHESS-atomic in particular are especially vulnerable to unnecessary context switches (for example, at a thread-local memory access), after which CHESS explores an entire prefix without enumerating any new inequivalent traces.

We empirically compared CB-DPOR's efficiency with the other model checkers' on all test cases. For each test case, we ran all model checkers for 30 minutes, and calculated each model checker's inefficiency: the total number of traces explored divided by the number of inequivalent traces, or the average number of equivalent traces. Figure 6-4 contains

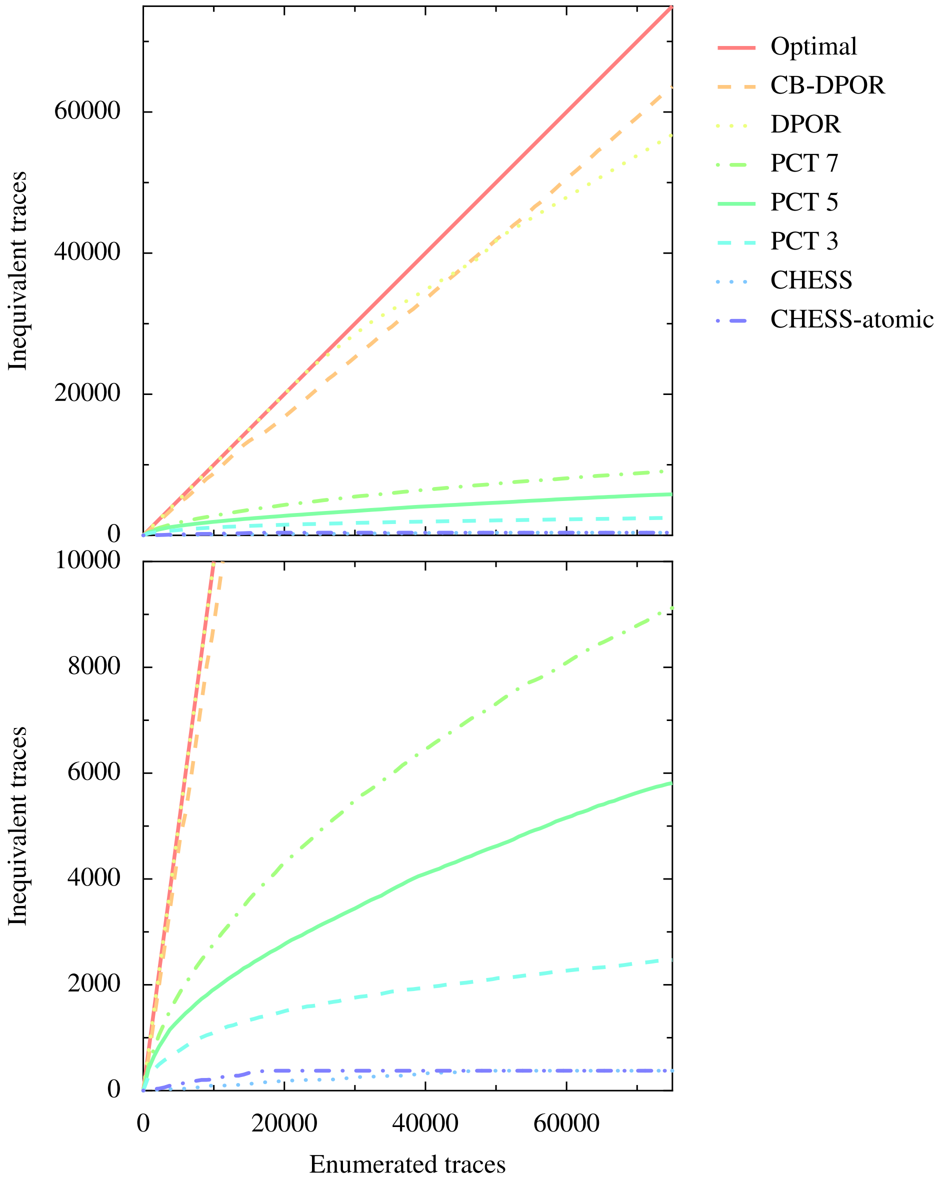

Figure 6-3: The number of inequivalent traces as a function of the number of traces explored so far for all model checkers on the test case `cds skiplistset 2`; an optimal model checker would explore only inequivalent traces. The bottom graph is a close-up of the top graph. Only CB-DPOR, PCT 3, PCT 5, and PCT 7 find the bug.

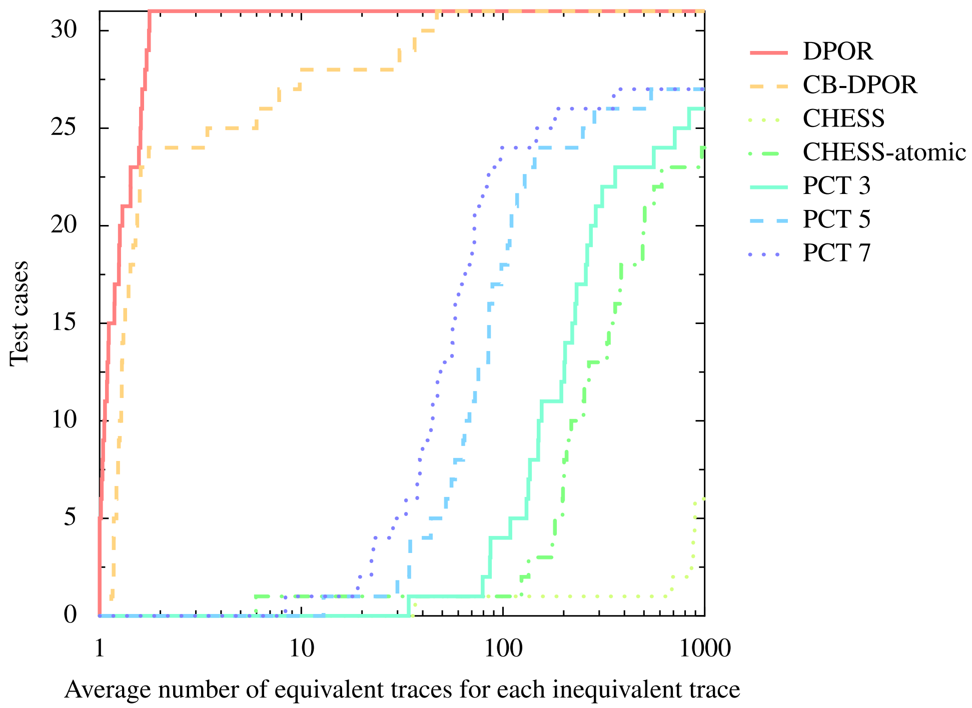

Figure 6-4: Cumulative distribution of model checker inefficiency over all randomly created test cases after running for 30 minutes.

a summary of all inefficiencies as a CDF over test cases. For example, DPOR has an inefficiency of no more than 2 in all test cases.

CB-DPOR is not as efficient as DPOR overall, as CB-DPOR does explore equivalent traces. However, CB-DPOR is significantly more efficient than PCT and CHESS on a majority of the test cases. PCT's efficiency increases as the number of priority change points increases, even though its bug finding effectiveness no longer grows: PCT 5 is just as good at finding bugs as PCT 7, yet PCT 7 is more efficient. PCT 7 does not improve upon PCT 5 because none of the test cases require 7 priority change points, and so while adding priority change points makes traces more likely to be inequivalent, it does not make PCT more likely to find bugs.

Chapter 7

Conclusion

This thesis presented CB-DPOR, a fast model checking algorithm for finding bugs in lock-free data structures, and its implementation in CODEX. CB-DPOR shows that CHESS and DPOR can be effectively combined in a single model checker that quickly finds bugs. CB-DPOR finds complicated bugs, and does so faster than earlier model checkers. CB-DPOR is faster than earlier model checkers because it is efficient, exploring few equivalent traces, and targeted, exploring only traces up to a given context bound. CODEX found previously unknown bugs in `libcds`, and reproduced bugs in Boost and data structures from `sv6`.

Bibliography

[1] Boost C++ libraries. `http://www.boost.org/`.

[2] sv6. `https://github.com/aclements/sv6`.

[3] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: A complete and automatic linearizability checker. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 330–340, Toronto, Canada, June 2010.

[4] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 167–178, Pittsburgh, PA, Mar. 2010.

[5] K. E. Coons, M. Musuvathi, and K. S. McKinley. Bounded partial-order reduction. In *Proceedings of the 2013 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 833–848, Indianapolis, IN, Oct. 2013.

[6] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM Symposium on Principles of Programming Languages*, pages 110–121, Long Beach, CA, Jan. 2005.

[7] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[8] F. Mattern. Virtual time and global states of distributed systems. In *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989.

[9] Max Khiszinsky. Concurrent data structures (libcds). `http://libcds.sourceforge.net/`.

[10] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 267–275, Philadelphia, PA, May 1996.

[11] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on*

Programming Language Design and Implementation, pages 446–455, San Diego, CA, June 2007.

[12] M. Musuvathi and S. Qadeer. Partial-order reduction for context-bounded state exploration. Technical Report MSR-TR-2007-12, Microsoft, Feb. 2007.

[13] M. Musuvathi and S. Qadeer. Fair stateless model checking. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 362–371, Tucson, AZ, June 2008.

[14] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 267–280, San Diego, CA, Dec. 2008.