The Lazy Happens-Before Relation: Better Partial-Order Reduction for Systematic Concurrency Testing *

Paul Thomson and Alastair F. Donaldson
Imperial College London, UK
{paul.thomson11,afd}@imperial.ac.uk

Abstract

We present the *lazy happens-before relation* (lazy HBR), which ignores mutex-induced edges to provide a more precise notion of state equivalence compared with the traditional happens-before relation. We demonstrate experimentally that the lazy HBR has the potential to provide greater schedule reduction during systematic concurrency testing with respect to a set of 79 Java benchmarks.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords Partial-order reduction; systematic concurrency testing

1. Introduction

Partial-order reduction (POR) [2] can be used to reduce the number of schedules explored during *systematic concurrency testing* (SCT) [3], a technique for dynamic analysis of concurrent programs implemented in tools such as Verisoft [3] and CHESS [5], with the guarantee that the reduction cannot lead to error states being missed. A potential issue with POR is that all interleavings of operations on the same mutex must be explored; in general, such interleavings may lead to different states. However, it is often unnecessary to explore interleavings of critical sections that access disjoint data, or access common data in a read-only fashion; this is common in the presence of *coarse-grained locking*.

We present the *lazy happens-before relation* (lazy HBR) that enables schedule reduction even in the presence of coarse-grained locking, allowing well-engineered concurrent software that has been deliberately designed to use a simple locking discipline to also reap the benefits of SCT-based analysis.

2. Example

We illustrate partial-order reduction and our key contribution via an example. Figure 1 shows an execution trace of a program—that is, a list of operations executed by the two threads, T1 and T2; T1 first locks a mutex m, reads a variable x, unlocks the mutex m and writes to a variable y. T2 then executes a similar sequence of operations. We refer to the operations as events and the list (a total-order of the events) as a schedule. Most POR techniques use the happens-before relation, which is a partial-order of the events in the schedule. The happens-before relation is a subset of the total-order of the schedule and a common definition is as follows: e_1 happens-before e_2 iff e_1 is before e_2 in the schedule and one of the following holds: (a) e_1 and e_2 are from the same thread; (b) e_1 and e_2 both access the same variable/mutex and at least one access is a modification; (c) there exists an event e such that e_1 happens-before e and e happens-before e_2 (the relation is transitively closed). Figure 1 depicts an example HBR; inter-thread edges are shown as arrows and we omit intra-thread edges and edges that can be obtained via transitivity. The theorem for the HBR is well-known:

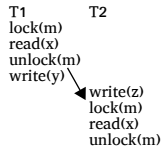

Figure 1: Schedule of a program and the inter-thread edges of the happens-before relation (edges implied by transitivity are omitted).

Theorem 2.1. *All linearizations of an HBR correspond to feasible schedules (they can be executed) and all will reach the same state.*

Thus, it is only necessary to explore one schedule from each equivalence class (partial-order) in order to detect safety property violations, such as data races, deadlocks and assertion failures. For example, note that the writes to y and z are unordered; swapping these events gives a schedule that can be executed and will reach the same state. The write to z can be swapped with the event above it several more times without violating the partial-order; all these schedules reach the same state. In contrast, the lock and unlock events cannot be swapped without changing the partial-order. Thus, a POR technique would only need to consider two schedules for this example: a schedule for the HBR shown, and a schedule in which T2 locks m before T1 (which has a different partial-order). Theorem 2.1 can be proven by showing that swapping two adjacent unordered events does not change the state reached.

In the *lazy happens-before relation* (lazy HBR), we modify condition (b) of the definition to become: (b) e_1 and e_2 both access the same *non-mutex* variable and at least one access is a modification. Thus, lock and unlock events do not introduce inter-thread edges. As such, the arrow in Figure 1 is removed and so the partial-order captures *all* possible schedules. Thus, a partial-order algorithm would only need to explore a single schedule. Unfortunately,

*This work was supported by an EPSRC-funded PhD studentship.

[Copyright notice will appear here once 'preprint' option is removed.]

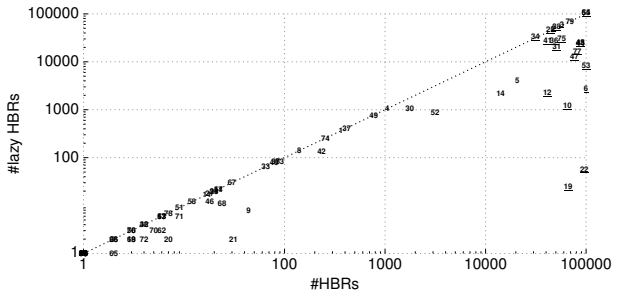

Figure 2: The number of regular vs. lazy happens-before relations explored within 100,000 schedules of DPOR.

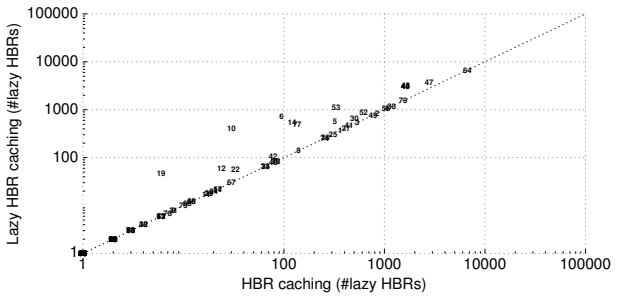

Figure 3: The number of lazy happens-before relations explored within 100,000 schedules of regular vs. lazy HBR caching.

not all linearizations of a lazy HBR are feasible schedules; that is, some schedules cannot be executed. For example, with respect to Figure 1, a schedule in which T2's lock event occurs between T1's lock and unlock events cannot be executed, as T2's lock event would be disabled (blocked) until T1 unlocks the mutex. This prevents straightforward use the of lazy HBR in certain POR techniques. Despite this, we contribute the following theorem:

Theorem 2.2. *Given two feasible schedules, S_1 and S_2, with the same lazy HBRs, S_1 and S_2 are guaranteed to reach the same state.*

The two schedules will not necessarily have the same HBR and so, in prior work, these schedules would be assumed to reach different states. The proof for this, which we now sketch, is not as straightforward as that of Theorem 2.1, as swapping adjacent events in a schedule that are unordered by lazy HBR does not always yield a feasible schedule. Instead, call our original program P; we consider an abstracted version P' in which lock and unlock are no-ops. An analogue of Theorem 2.1, using lazy HBR instead of HBR, now applies to schedules of P'. We complete the proof by showing that the states explored by each event of a schedule of P are the same as those explored when executing the schedule on P', modulo the state of mutexes. The final state of the mutexes when executing S_1 and S_2 on P are shown to be the same by a counting argument (both schedules contain the same lock and unlock events). Thus, the schedules reach the same state.

Lazy HBR caching Re-exploration of redundant states during SCT can be avoided using *happens-before relation caching* (HBR caching), a simple form of partial-order reduction: each HBR is stored after each executed event and if the same HBR is reached during a later execution, then the schedule is redundant and is not explored further [4]. An immediate benefit of the lazy HBR is that it yields *lazy HBR caching*, where the lazy HBR is used in place of the HBR to identify equivalent states more effectively.

3. Evaluation

We have implemented an SCT tool for Java programs called LAZYLOCKS and collected 79 open source multithreaded Java benchmarks, which we use to evaluate the lazy HBR. The tool and benchmarks are available at:

`http://sites.google.com/site/lazyhbr/`

Redundant HBRs We evaluate the potential reduction possible from the lazy HBR and its use in more precisely identifying unique states by testing each benchmark with a schedule limit of 100,000 using dynamic partial-order reduction (DPOR) [1], a POR technique that uses the *regular* HBR. We record the number of schedules, terminal HBRs and terminal lazy HBRs explored. We show that many unique HBRs explored actually lead to equivalent states (and so are redundant), which we know because they have the same lazy HBR. For a given benchmark we have the following inequality for the number of states, HBRs, etc. explored:

#states $\leq$ #lazy HBRs $\leq$ #HBRs $\leq$ #schedules $\leq$ 100,000

Figure 2 plots our results using a log scale. Each point is a benchmark id; an id at (x, y) indicates that DPOR explored x HBRs and y lazy HBRs. A benchmark is underlined if DPOR hit the schedule limit, in which case unexplored terminal states are likely to remain; otherwise all terminal states were explored.

There are 33 benchmarks below the diagonal; thus, for these benchmarks, redundant HBRs were explored according to the lazy HBR. Across the 33 benchmarks below the diagonal, 910,007 (80%) of the unique HBRs explored were found to be redundant.

Lazy HBR caching We evaluated lazy vs. regular HBR caching by comparing the number of lazy HBRs explored by both techniques within 100,000 schedules. Figure 3 plots our results using a log scale. Each point is a benchmark id; an id at (x, y) indicates the number of lazy HBRs explored by HBR caching and lazy HBR caching, respectively. As expected, regular HBR caching never explored more lazy HBRs. There are 18 benchmarks below the diagonal; across these benchmarks, lazy HBR caching explored a total of 8,969 (84%) more terminal lazy HBRs than regular HBR caching.

4. Future Work: Lazy DPOR

Because not every linearization of a lazy HBR is a feasible schedule, the *lazy* HBR cannot be immediately used in place of the regular HBR during DPOR. In future work we plan to investigate a new *lazy DPOR* algorithm that exploits the lazy HBR to provide a more significant reduction than regular DPOR and lazy HBR caching.

Acknowledgements

We are grateful to Jeroen Ketema for discussions and to Pavel Parizek for feedback related to this work.

References

[1] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.

[2] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*. Springer, 1996.

[3] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL*, pages 174–186, 1997.

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

[5] M. Musuvathi et al. Finding and reproducing Heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.