Extended Sequential Reasoning for Data-Race-Free Programs

Laura Effinger-Dean, Hans-J. Boehm, Dhruva Chakrabarti, Pramod Joisha

HP Laboratories
HPL-2011-57

Keyword(s):

program analysis, optimization, threads, data races

Abstract:

Most multithreaded programming languages prohibit or discourage data races. By avoiding data races, we are guaranteed that variables accessed within a synchronization-free code region cannot be modified by other threads, allowing us to reason about such code regions as though they were single-threaded. However, such single-threaded reasoning is not limited to synchronization-free regions. We present a simple characterization of extended interference-free regions in which variables cannot be modified by other threads. This characterization shows that, in the absence of data races, important code analysis problems often have surprisingly easy answers. For instance, we can use local analysis to determine when lock and unlock calls refer to the same mutex. Our characterization can be derived from prior work on safe compiler transformations, but it can also be simply derived from first principles, and justified in a very broad context. In addition, systematic reasoning about overlapping interference-free regions yields insight about optimization opportunities that were not previously apparent. We give preliminary results for a prototype implementation of interference-free regions in the LLVM compiler infrastructure. We also discuss other potential applications for interference-free regions.

External Posting Date: May 6, 2011 [Fulltext] Approved for External Publication
Internal Posting Date: May 6, 2011 [Fulltext]

Extended Sequential Reasoning for Data-Race-Free Programs [*]

Laura Effinger-Dean
University of Washington [†]
effinger@cs.washington.edu

Hans-J. Boehm Dhruva Chakrabarti
Pramod Joisha
Hewlett-Packard Laboratories
{hans.boehm, dhruva.chakrabarti,
pramod.joisha}@hp.com

Abstract

Most multithreaded programming languages prohibit or discourage data races. By avoiding data races, we are guaranteed that variables accessed within a synchronization-free code region cannot be modified by other threads, allowing us to reason about such code regions as though they were single-threaded. However, such single-threaded reasoning is not limited to synchronization-free regions. We present a simple characterization of extended *interference-free regions* in which variables cannot be modified by other threads.

This characterization shows that, in the absence of data races, important code analysis problems often have surprisingly easy answers. For instance, we can use local analysis to determine when lock and unlock calls refer to the same mutex. Our characterization can be derived from prior work on safe compiler transformations, but it can also be simply derived from first principles, and justified in a very broad context. In addition, systematic reasoning about overlapping interference-free regions yields insight about optimization opportunities that were not previously apparent.

We give preliminary results for a prototype implementation of interference-free regions in the LLVM compiler infrastructure. We also discuss other potential applications for interference-free regions.

1. Introduction

Programming languages that support multiple concurrent threads communicating via shared variables must provide a *memory model*. This model specifies whether variables may be concurrently accessed, when an update by one thread becomes visible to another, and so on.

Many mainstream programming languages now have memory models that disallow data races; that is, they completely prohibit concurrent accesses to a shared variable unless both accesses are reads. The prohibition against data races has arguably been the dominant rule for decades. Both Posix threads [7] and Ada [13] have explicitly disallowed data races from their inception. The revisions of the C and C++ standards [2–5] currently being finalized are more precise but equally explicit on this point.

Here we assume such a prohibition against data races. We do not directly address languages like Java [10], which also discourage, but do not completely disallow data races. Java attempts to provide well-defined semantics for data races, but recent work [1, 12] points out serious deficiencies in the approach, and calls for a reexamination of the problem. At this point, we can only speculate whether our work will eventually be applicable to Java in some form.

It is well-known that, in the absence of data races, when a variable x is accessed, its value cannot be changed by another thread between the immediately preceding and immediately following synchronization operations. If another thread could modify the variable between those two points (i.e., within a *synchronization-free region*), then there would be an execution of the program in which the two accesses of x are concurrent and therefore form a data race. Similarly, if x is written within a synchronization-free region, it cannot be read by another thread within that region.

These observations are fundamental to code optimization in current compilers. They allow compilers for languages like C and C++ to largely ignore the presence of threads when transforming synchronization-free code. No other thread can detect such transformations without introducing a data race. As long as synchronization operations are treated as "opaque" (i.e., as potentially modifying any memory location), and no speculative operations, and hence no new data races, are introduced, safe sequential transformations remain correct in the presence of multiple threads.

In this paper, we introduce a simple but powerful extension to synchronization-free regions called *interference-free regions*. The observation we make is that when a variable x is accessed, its value cannot be changed by another thread between the immediately preceding *acquire*[1] synchronization and the immediately following *release*[2] synchronization. This is strictly more general than synchronization-free

[*] This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-0718124.

[†] Some of this work was done while at HP Labs.

[1] E.g., mutex lock, thread join, volatile read.

[2] E.g., mutex unlock, thread create, volatile write.

regions, which do not discern between different types of synchronization operations. A nice consequence of this generalization is that the interference-free regions for different accesses to the same variable may overlap, revealing previously unrecognized optimization opportunities.

Our contributions are as follows:

- We define interference-free regions over execution traces, and present several interesting examples for which reasoning about interference-freedom, as we have defined it, allows for easy answers to tricky problems.
- We present a compiler analysis that identifies conservative approximations of interference-free regions, allowing other compiler analyses to remove redundant memory operations across synchronization boundaries.

Our analysis is straightforward, but general and applicable to a number of interesting cases. We hope that the discussion sheds light on issues that have not been well understood prior to our work.

2. Interference-Free Regions

In this paper, we concern ourselves with regions of code that are *not* synchronization-free. Can we regain some sequential reasoning even in the presence of synchronization? To illustrate the problem we are trying to address, consider

```
lock(mtx_p);
...
unlock(mtx_p);
```

where mtx_p is a global, potentially shared, pointer to a mutex. We will also assume, for purposes of the examples, that the only operations on mutexes are lock() and unlock() operations taking a pointer to a mutex, and that ellipses represent synchronization-free code that does not modify any of the variables mentioned in the example. Is it the case that both instances of mtx_p always refer to the same lock? Or could another thread modify mtx_p in the interim?

The prohibition against data races often allows us to answer such questions, *without analyzing code that might be run by other threads*. Without such a prohibition, and without knowledge about other threads in the system, we clearly could not guarantee anything about concurrent updates of mtx_p. Moreover, reasoning about synchronization-free regions is insufficient in this case: the lock is acquired between the first and second load of mtx_p, and hence the two references are not in the same synchronization-free region. Nevertheless, we argue that the data-race-freedom assumption is strong enough to establish, for this example, that another thread cannot concurrently modify mtx_p.

We make an easily provable, but foundational, and apparently normally overlooked observation: the region of code during which other threads cannot modify a locally accessed variable may often be extended in both directions beyond the access's enclosing synchronization-free region. In particular, we can extend the boundary of a region backwards in the trace past any earlier release operations, such as mutex unlock() calls, and forwards through any later acquire operations, such as mutex lock() calls. To put it another way, the variable cannot be modified by other threads in the region between the most recent acquire operation and the next release operation. We call this extended region the *interference-free region* for an access.

Thus, in our example, the interference-free region corresponding to the initial load of mtx_p extends through the immediately following lock() call, and includes the second load of mtx_p, guaranteeing that both loads must yield the same pointer. Here we have separated the accesses to mtx_p from the synchronization operations and labeled the first load, but the idea is the same:

```
A: tmp = mtx_p;   ⎫
lock(tmp);        ⎬ interference-free
...               ⎪ region for A
tmp = mtx_p;      ⎭
unlock(tmp);
```

We believe that the notion of an interference-free region is a fundamental observation about the analysis of multithreaded code, in that it gives us a much better characterization of the applicability of single-threaded techniques. Note that we make these deductions with no specific information about other threads in the system; we are relying only on the data-race-freedom requirement imposed by the language.

2.1 Formalism

Memory models are delicate and must be reasoned about in a formal setting. We give a formal definition of our execution model and a proof that interference-free regions are correct in Appendix A. Briefly, if one thread reads a variable x and another thread modifies x, then the data-race-freedom guarantee requires that these accesses be ordered by the *happens-before* relation, a partial order on actions in the execution which is defined as the transitive closure of the *program order* and *synchronizes-with* relations. Because the two actions occur in different threads, and program order only orders actions from the same thread, at least one of the edges in the happens-before ordering must be a synchronizes-with edge. Given how we defined interference-free regions, it is not possible for there to be an incoming synchronizes-with edge in the region before the access, nor for there to be an outgoing synchronizes-with edge in the region after, so any write by another thread must happen-before or happen-after the entire interference-free region. The proof itself is elegant, and generic with respect to the types of synchronization operations used by the language.

2.2 Interference-Free Regions in Thread Traces

Given a memory access in an execution, we can infer the interference-free region for that access. In the execution trace for a single thread in Figure 1, the IFR for access X extends

```
lock(&mtx1);
A: ...
unlock(&mtx2);      no acquires
...
unlock(&mtx3);
...                 synchronization-    interference-
X: tmp = x;         free region         free region
...
lock(&mtx4);
...                 no releases
lock(&mtx5);
B: ...
unlock(&mtx6);
```

Figure 1. An interference-free region in a thread trace. Ellipses are synchronization-free code.

```
A: tmp = mtx_p;
lock(tmp);
...                 interference-free
lock (&mtx2);       region for A
B: ...
unlock(&mtx2);         interference-free
...                    region for C
C: tmp = mtx_p;
unlock(tmp);
```

Figure 2. The interference-free regions for accesses A and C overlap, despite the intervening critical section.

backwards through region A and forwards through region B.[3] Any conflicting write must happen-before the lock of mtx1 or happen-after the unlock of mtx6. The exact sequence of lock acquires and releases is irrelevant; we simply identify incoming and outgoing synchronizes-with edges.

2.3 Overlapping Regions

We extend our reasoning about interference-free regions by considering cases in which two or more regions for the same variable *overlap*. If x cannot be changed in either interval a or interval b, and a and b overlap, then clearly it cannot change in $a \cup b$.

For example, suppose there is a critical section nested between two accesses, as in Figure 2. In this case, the interference-free region for load A extends forwards into region B. The interference-free region corresponding to load C extends backwards, past the unlock into the region B. Thus mtx_p must be interference-free for the entire region, and we can conclude that all locks acquired are released.

```
A: tmp = p;
lock(&tmp->mtx);         interference-free
...                      region for A
lock(&mtx2);
B: ...
unlock(&mtx2);
...
C: tmp = p;           interference-free
local = tmp->data;    region for C
...
lock(&mtx3);
D: ...
unlock(&mtx3);           interference-free
...                      region for E
E: tmp = p;
unlock(&tmp->mtx);
```

Figure 3. Here, the load of p at line C means that p is interference-free during both nested critical sections.

```
while(...) {
  A: r1 = x;
  ...
  lock(&mtx);
  ...
  unlock(&mtx);
}
```

Figure 4. The access at line A is loop-invariant.

The above reasoning does not generally apply if there is more than one nested critical section in a row. However, there are cases for which we can derive similar results even then. Consider the common case in which, rather than mtx_p, we have a pointer p to a structure that includes both a mutex and some data, with the program as shown in Figure 3. The program includes three loads of p, at lines A, C, and E. The interference-free region from the load of p at line A extends forward through region B. The interference-free region from the load of p at line E extends backwards through region D. The interference-free region from the load of p at line C extends backwards through B and forwards through D. Thus we conclude that p cannot be modified by another thread.

More generally, we may conclude a variable x is interference-free along a section of the execution trace if it is accessed between every pair of release and subsequent acquire operations.

2.4 Loop Invariance

We can also use interference-free regions to determine loop-invariant references for loops that contain synchronization. The loop in Figure 4 is again not a simple synchronization-free region, so it is not immediately clear whether the load of x can be moved out of the loop. However, x is guaranteed to be accessed between every lock release and the next lock

[3] We assume a language in which the only synchronizes-with edges are from unlocks to locks of the same variable. The happens-before relation may be more complex; for example, if the language includes a trylock primitive that cannot spuriously fail, then lock acquires may have outgoing happens-before edges, reducing the size of the interference-free region. The exact definition of the happens-before relation is orthogonal to our analysis.

acquire operation. Hence the interference-free region for each access of x must overlap with the previous and next one, if they exist. Therefore, all loaded values of x must be the same, so it is safe to move the load out of the loop (taking care to guard the load so as not to introduce a data race).

Similar observations apply to loops that access C++0x [4] atomic objects. If we consider the loop below where a is an atomic variable, we can deduce that the loop contains only acquire operations, and therefore the interference-free region of any access in the loop includes all later iterations of the loop. Thus the read of x can safely be hoisted out of the loop:

```
do {
  r1 = x;
  ...
} while(a);
```

2.5 Barriers

A common form of synchronization is rendezvous-style barriers (e.g., the pthread_barrier_t type). Barriers allow threads to stop and wait for other threads to reach a certain point in their execution before continuing. Treating barriers as release and acquire actions is too imprecise, so we handle them specially. We have proved that if a variable is accessed *both before and after* a call to the "wait" function for a barrier, then we can extend the interference-free regions for both accesses through the call. This makes intuitive sense, as any remote write to the variable would have to happen either before or after the barrier call, and the write would therefore race with at least one of the two local accesses. A formal explanation of barriers and their relation to interference-free regions appears in Appendix B.

3. Compiler Analysis

Compilers can use the concept of interference-free regions to improve the scope of optimizations for data-race-free programs. We have implemented a pass in the LLVM compiler framework [9] that identifies interference-free regions. Because we do not know which path through a program a given execution will take, we must be conservative: we identify synchronization calls that, no matter which path is taken, fall into some interference-free region for a given variable. We then remove the variable from the set of variables modified by each identified synchronization call.

3.1 Algorithm

Compilers apply sequential optimizations to multithreaded code by assuming that synchronization functions are *opaque*: they might access any variable. If we can show that no concurrently running threads modify a particular variable at the moment a synchronization call is executed, it is sound to remove that variable from the call's set of modified variables.[4]

We identify synchronization calls whose modified sets may be pruned by exploiting two symmetric insights:

1. If, on a path through a function that passes through an acquire call C, there is an access A to a variable x such that A precedes C and there are no release calls between A and C, then C is in the interference-free region for A for that path. Therefore, if such an access A exists for every path through C, C does not modify x.
2. If, on a path through a function that passes through a release call C, there is an access A to a variable x such that A follows C and there are no acquire calls between C and A, then C is in the interference-free region for A for that path. Therefore, if such an access A exists for every path through C, C does not modify x.

Our analysis determines two pieces of information. First, for each acquire call, we need the set of variables that must have been *accessed since the last release call* (ASLR). Second, for each release call, we need the set of variables that must be *accessed before the next acquire call* (ABNA). The former is a simple forward dataflow analysis; the latter is a simple backward dataflow analysis. For acquire calls, we remove any variables in the ASLR set from the modified set for that call; for release calls, we remove any variables in the ABNA set. A call to pthread_barrier_wait is interference-free for a given variable if that variable appears in both the ASLR and ABNA sets for the call.

As an example, in Figure 2, mtx_p is removed from the modified sets for the first three synchronization calls, although not the last because it is a release action and its ABNA set is empty. A redundant load analysis will therefore find that the second load can be eliminated. Figure 4 is an example of conservatism in our analysis: x is not in the ABNA set for the unlock call (because there is a path that does not access x after the unlock), so access A will not be hoisted out of the loop.

In order to improve the accuracy of the analysis, we distinguish between read and write accesses in the implementation. For example, if a variable x must be *modified* (not just accessed) before an acquire call, then we may assume that the call neither reads nor writes x.

3.2 Data-Race-Freedom

Since we assume data-race-freedom, programs with data races may be transformed in hard-to-predict ways, complicating debugging. This is already true for current compilers; otherwise current sequential techniques could not even be applied within synchronization-free regions [3]. We are not qualitatively changing the situation, and it does not appear to be a major problem in practice. Even if a compiler does not use our analysis, programmers should debug using a data-race detector (e.g., [6]) in order to avoid unpredicatable be-

[4] The bodies of the synchronization functions may modify some variables—e.g., pthread_create, a synchronization operation with release behavior, modifies the new thread ID—so we must take care to distinguish these (non-racy) writes from writes in other threads.

```
int max, shared_counter;
pthread_mutex_t m;

void *f(void *my_num) {
  int n = *((int *) my_num);
  while (n <= max) {
    pthread_mutex_lock(&m);
    shared_counter++;
    pthread_mutex_unlock(&m);
    n += 2;
  }
}

int main() {
  pthread_t t1, t2;
  int n1 = 1;
  int n2 = 2;
  max = 10000000;
  shared_counter = 0;
  pthread_mutex_init(&m, NULL);
  pthread_create(&t1, NULL, f, (void *) &n1);
  pthread_create(&t2, NULL, f, (void *) &n2);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
}
```

Figure 5. A microbenchmark demonstrating the effectiveness of IFR-based optimizations. When combined with our analysis, GVN moves the load of `max` out of the loop in `f()`.

havior due to incorrect optimizations and to catch data races immediately rather than after data structure corruption.

We expect this analysis might be useful on an opt-in basis. Programmers could, for example, specify on the command line that "aggressive DRF optimizations" should be enabled. We envision an "-ODRF" flag in future C/C++ compilers, the documentation for which would make explicit that racy programs might have unexpected behavior.

3.3 Preliminary Results

Our LLVM implementation is still in progress, but our initial results are promising. We inserted our alias analysis into LLVM's link-time optimization pipeline just before the loop-invariant code motion (LICM) and global value numbering (GVN) transformations. The machine used for compilation and running the tests was a 4-core 2.8GHz Intel Xeon with 16GB of RAM, running Linux.

Microbenchmark Figure 5 shows a small C program that contains a redundant load: every iteration of the loop in function `f()` checks the value of `max`, which is constant. But because the loop contains synchronization, the standard LLVM alias analysis pass assumes the synchronization calls may modify `max`. Our analysis removes `max` from the modified sets for the synchronization calls, allowing GVN to hoist the load out of the loop. Running the program under valgrind shows that the optimization is indeed effective: the optimized program performs 10 million fewer loads. However, the program performs over 400 million memory operations in total, so there is no noticable performance improvement.

SPLASH-2 Benchmark	**Lines of code**	**Sync calls tightened**	**Loads deleted (GVN)**
lu-n	678	14	16
radix	833	27	22
fft	899	12	13
lu-c	911	17	19
water-n	2,063	27	24
water-s	2,670	27	30
barnes	2,864	20	11
ocean-n	3,046	34	53
volrend	4,204	31	25
fmm	4,325	37	36
ocean-c	4,774	95	79
cholesky	5,139	83	19
raytrace	10,649	19	14
radiosity	11,760	93	45

Table 1. SPLASH-2 results.

Realistic applications We compiled the SPLASH-2 benchmarks [14] using our analysis. The analysis did not affect the LICM pass (we suspect because LICM only handles function calls that do not modify any memory locations), but the GVN pass found numerous opportunities to remove redundant loads (Table 1).[5] The third column in Table 1 lists the number of synchronization calls which were found to be in the interference-free region of at least one variable. The last column lists the difference in the number of loads deleted by GVN when run with and without our analysis.

Although the analysis exposed a number of redundant loads, we have had little success in terms of actually extracting performance from these optimizations. The benchmarks either have similar performance on both versions of the code, or our "optimized" version is slightly worse. One problem is that the loads may not be located on hot paths. Another possibility is that the optimizations increased the live ranges of variables, resulting in more loads as register variables are spilled to the stack (perhaps due to the low number of callee-saved registers on the x86 architecture).

4. Other Applications

Interference-free regions are useful for understanding the behavior of functions that contain internal synchronization. For example, the C++ draft specification defines `malloc` and `free` as acquire and release synchronization operations, respectively [4]. We can use interference-free regions to

[5] Theoretically, our analysis should also be useful for dead store elimination, but we did not observe any improvement in LLVM's dead store elimination pass, so we concentrate on loads here.

establish that global variables p and q are properly allocated and deallocated in the following code:

```
p = malloc(...);
q = malloc(...);
free(p);
free(q);
```

This kind of reasoning is applicable not just to compilers, but also to static analysis tools, where reasoning about properties such as deadlock freedom or memory allocation often requires knowledge about variables that might conceivably be changed by other threads. We should be able to use our existing alias analysis to improve the accuracy of existing analyses, when it is sound to assume data-race-freedom.

5. Related Work

Our analysis is related to a compiler transformation known as *roach-motel reordering*. This transformation increases the size of critical sections by moving actions either past a lock acquire or before a lock release. In some cases, it is possible to use this line of reasoning to infer interference-free regions by repeatedly swapping an access until it reaches the end of a region. Sevcik established that this transformation is legal for data-race-free programs [11]. We believe our characterization is independently useful, particularly since we make very minimal assumptions about the language and synchronization primitives and we avoid complex reasoning about syntactic code transformations. In particular, we know of no prior presentation of a similar compiler analysis, nor any discussion of the consequences of overlapping regions. Going from roach-motel reordering to the optimizations in Figures 2, 3 and 4 is nontrivial; our analysis considerably simplifies the necessary deductions.

Prevous work by several of the authors of this paper describes a framework for enabling sequential optimizations in multithreaded programs [8]. They identify paths on which variables are "siloed" by iteratively refining a graph of the program. (This "siloed" property is essentially the same as our notion of interference-freedom.) Like us, their implementation refines the modified/referenced sets for synchronization calls. Our work is complementary to this work, as our analysis could likely be incorporated neatly into their framework as an "interference-type refinement."

6. Conclusion

We have presented an approach to inferring interference-free regions in data-race-free programs. Since our analysis is based heavily on data-race-freedom, it requires only information about a single thread. We have implemented these ideas as an alias analysis pass in an optimizing compiler, and we have also discussed how this idea could apply to static analysis of multithreaded code. Our approach enables new kinds of inferences about program behavior that had not previously been considered.

References

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

[2] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2011.

[3] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[4] C++ Standards Committee, Pete Becker, ed. Programming Languages - C++ (final committee draft). C++ standards committee paper WG21/N3092=J16/10-0082, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2010/n3092.pdf`, March 2010.

[5] C Standards Committee (WG14). Working Draft: Programming Languages — C. C standards committee paper WG14 N1425, `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1425.pdf`, November 2009.

[6] C. Flanagan and S. Freund. FastTrack: Efficient and precise dynamic race detection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[7] IEEE and The Open Group. *IEEE Standard 1003.1-2001*. 2001.

[8] P. Joisha, R. S. Schreiber, P. Banerjee, H.-J. Boehm, and D. R. Chakrabarti. A technique for the effective and automatic reuse of classical compiler optimizations on multithreaded code. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2011.

[9] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, 2004.

[10] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2005.

[11] J. Sevcik. *Program Transformations in Weak Memory Models*. PhD thesis, University of Edinburgh, 2008.

[12] J. Sevcik and D. Aspinall. On the validity of program transformations in the Java memory model. In *European Conference on Object-Oriented Programming*, 2008.

[13] *Reference Manual for the Ada Programming Language: ANSI/MIL-STD-1815A-1983 Standard 1003.1-2001*. United States Department of Defense, 1983. Springer.

[14] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *International Symposium on Computer Architecture*, 1995.

A. Analysis

The idea of our analysis is as follows: for each memory access in a program, we define an *interference-free region* around the access by identifying points where another thread cannot asynchronously modify the accessed variable without causing a data race. This interference-free region may include release actions that occur prior to the memory access, as well as acquire actions that occur after the memory access. We observe that any conflicting access in other threads must happen-before or happen-after the memory access by the data-race-free assumption, so conflicting accesses are impossible if there are no release actions before the access or acquire actions after the access.

A.1 Model

Define an execution to be a set of memory actions, together with three relations on that set. The *sequenced-before relation* totally orders all actions in a thread, and does not order actions from different threads. The *synchronizes-with relation* defines an ordering between synchronization operations. Finally, the *happens-before order* is the reflexive transitive closure of the union of the sequenced-before and synchronizes-with orders. Formally, we say that an execution E is a quadruple $(A, \leq_{\mathsf{sb}}, <_{\mathsf{sw}}, \leq_{\mathsf{hb}})$ where:

- A is a set of actions tagged with a thread ID t and a unique ID u: (t, a, u). Actions include reads and writes to normal shared variables, as well as other unspecified actions:

$$a \quad ::= \quad \mathsf{read}(x) \mid \mathsf{write}(x) \mid \ \dots$$

- $\leq_{\mathsf{sb}}$ (sequenced-before) is a partial order over UIDs that defines a total order over actions performed by the same thread:
  - If $(t_1, a_1, u_1), (t_2, a_2, u_2) \in A$ and $u_1 \leq_{\mathsf{sb}} u_2$, then $t_1 = t_2$.
  - If $(t_1, a_1, u_1), (t_1, a_2, u_2) \in A$, then either $u_1 \leq_{\mathsf{sb}} u_2$ or $u_2 \leq_{\mathsf{sb}} u_1$.
- $<_{\mathsf{sw}}$ (synchronizes-with) is a relation over UIDs. We do not define this relation precisely, as the details are irrelevant to our analysis; presumably, lock releases synchronize-with acquires of the same lock, writes to volatile variables synchronize-with reads of the same variable, and so on.
- $\leq_{\mathsf{hb}}$ is a partial order over UIDs constructed as follows: $u_1 \leq_{\mathsf{hb}} u_2$ if
  - $u_1 \leq_{\mathsf{sb}} u_2$,
  - $u_1 <_{\mathsf{sw}} u_2$, or
  - there exists u_3 such that $u_1 \leq_{\mathsf{hb}} u_3$ and $u_3 \leq_{\mathsf{hb}} u_2$.[6]

We omit the values being read and written (or, equivalently, a writes-seen function). Because we only consider data-race-free programs, every normal read sees the value most recently written to the same location in the happens-before relation.

Next we define the notion of incoming and outgoing edges: happens-before edges that start in one thread and end in another. An action has an *incoming* edge if an action in another thread synchronizes with that action. An action has an outgoing edge if it synchronizes with an action in another thread. Incoming and outgoing edges must be synchronizes-with edges, as sequenced-before edges always go between actions of the same thread. The intuition here is that by establishing that a subsection of a single thread's execution does *not* have any incoming or outgoing edges, we can convince ourselves that no other thread could interfere with that thread's execution.

Definition 1 (Outgoing Edge)**.** *Let* $(t_1, a_1, u_1) \in A$. u_1 *has an* outgoing happens-before edge *if there exists* $(t_2, a_2, u_2) \in A$ *such that* $u_1 <_{\mathsf{sw}} u_2$ *and* $t_1 \neq t_2$.

Definition 2 (Incoming Edge)**.** *Let* $(t_2, a_2, u_2) \in A$. u_2 *has an* incoming happens-before edge *if there exists* $(t_1, a_1, u_1) \in A$ *such that* $u_1 <_{\mathsf{sw}} u_2$ *and* $t_1 \neq t_2$.

Outgoing happens-before edges usually correspond to lock releases, thread spawns, or writes to volatile variables. Incoming happens-before edges usually correspond to lock acquires, thread start actions, or reads of volatile variables.

Next, we prove that if there is a happens-before edge between two actions in two different threads, then there must be an action in the first thread with an outgoing happens-before edge.

Lemma 1 (Existence of Outgoing Edge)**.** *Let* (t_1, a_1, u_1), $(t_2, a_2, u_2) \in A$ *such that* $u_1 \leq_{\mathsf{hb}} u_2$ *and* $t_1 \neq t_2$. *Then there exists* u_3 *such that* $u_1 \leq_{\mathsf{sb}} u_3$*,* $u_3 \leq_{\mathsf{hb}} u_2$*, and* u_3 *has an outgoing edge.*

Proof. As happens-before is the closure of the sequenced-before and synchronizes-with relations, clearly any chain of happens-before edges that crosses threads must include an outgoing synchronizes-with edge. Formally, we can prove this by induction on $u_1 \leq_{\mathsf{hb}} u_2$:

- Clearly $\neg(u_1 \leq_{\mathsf{sb}} u_2)$, as $t_1 \neq t_2$.
- If $u_1 <_{\mathsf{sw}} u_2$, then let $u_3 = u_1$. As $t_1 \neq t_2$, u_1 has an outgoing edge. By reflexivity, $u_1 \leq_{\mathsf{sb}} u_1$. By assumption, $u_1 \leq_{\mathsf{hb}} u_2$.
- Now suppose $u_1 \leq_{\mathsf{hb}} u_4$ and $u_4 \leq_{\mathsf{hb}} u_2$, where $(t_4, a_4, u_4) \in A$. Either $t_1 = t_4$ or $t_1 \neq t_4$.
  - If $t_1 = t_4$, then either $u_1 \leq_{\mathsf{sb}} u_4$ or $u_4 \leq_{\mathsf{sb}} u_1$ by the definition of $\leq_{\mathsf{sb}}$. If $u_4 \leq_{\mathsf{sb}} u_1$, then $u_4 = u_1$ because $\leq_{\mathsf{hb}}$ is antisymmetric, so $u_1 \leq_{\mathsf{sb}} u_4$ as well. Use the

[6] The actual C++0x happens-before relation is not transitively closed, so that `memory_order_depends` can be supported. The effect is to prevent $\leq_{\mathsf{sb}}$ from contributing to $\leq_{\mathsf{hb}}$ in certain contexts. This does not affect our arguments.

inductive hypothesis to find u_3 such that $u_4 \leq_{\mathsf{sb}} u_3$, $u_3 \leq_{\mathsf{hb}} u_2$, and u_3 has an outgoing edge. Then u_3 is the witness because $u_1 \leq_{\mathsf{sb}} u_3$ by transitivity.

- If $t_1 \neq t_4$, then use the inductive hypothesis to find u_3 such that $u_1 \leq_{\mathsf{sb}} u_3$, $u_3 \leq_{\mathsf{hb}} u_4$, and u_3 has an outgoing edge. By transitivity, $u_3 \leq_{\mathsf{hb}} u_2$. □

Symmetrically, if there is a happens-before action between two actions in two different threads, the second thread must have an action with an incoming happens-before edge.

Lemma 2 (Existence of Incoming Edge). *Let* (t_1, a_1, u_1), $(t_2, a_2, u_2) \in A$ *such that* $u_1 \leq_{\mathsf{hb}} u_2$ *and* $t_1 \neq t_2$. *Then there exists* u_3 *such that* $u_1 \leq_{\mathsf{hb}} u_3$, $u_3 \leq_{\mathsf{sb}} u_2$, *and* u_3 *has an incoming edge.*

A.2 Interference-Freedom

We now establish our key result: if a region of code after a normal memory access has no outgoing happens-before edges, then any writes must happen-after that region of code. The usefulness of this result depends crucially on the fact that in an execution of a data-race-free program, reads and writes to the same location are ordered by happens-before.

Theorem 1 (Forwards Interference-Free). *Let* $(t_1, \mathsf{read}(x), u_1), (t_1, a_2, u_2) \in A$ *such that* $u_1 <_{\mathsf{sb}} u_2$. *Furthermore, for all* u_3 *such that* $u_1 \leq_{\mathsf{sb}} u_3 <_{\mathsf{sb}} u_2$, u_3 *does not have an outgoing edge. Finally, there is some write* $(t_4, \mathsf{write}(x), u_4)$ *such that* $u_1 \leq_{\mathsf{hb}} u_4$ *and* $t_1 \neq t_4$. *Then* $u_2 \leq_{\mathsf{hb}} u_4$.

Proof. By Lemma 1, there exists u_3 such that $u_1 \leq_{\mathsf{sb}} u_3$, $u_3 \leq_{\mathsf{hb}} u_4$, and u_3 has an outgoing edge. Clearly $u_2 \leq_{\mathsf{sb}} u_3$, because otherwise u_3 would violate an assumption. Then $u_2 \leq_{\mathsf{hb}} u_4$ by transitivity of happens-before. □

A similar result applies when the initial action is a write. Theorem 1 provides an invariant over the value of x for a subset of a trace starting with a read (assuming t_i does not write to x). We say that x is *interference-free* for thread t_i between u_1 and u_2.

Symmetrically, we can show that a location is interference-free before a read for a region in which there are no incoming happens-before edges.

Theorem 2 (Backwards Interference-Free). *Let* $(t_1, \mathsf{read}(x), u_1), (t_1, a_2, u_2) \in A$ *such that* $u_1 <_{\mathsf{sb}} u_2$. *Furthermore, for all* u_3 *such that* $u_1 <_{\mathsf{sb}} u_3 \leq_{\mathsf{sb}} u_2$, u_3 *does not have an incoming edge. Finally, there is some write* $(t_4, \mathsf{write}(x), u_4)$ *such that* $u_4 \leq_{\mathsf{hb}} u_2$ *and* $t_1 \neq t_4$. *Then* $u_4 \leq_{\mathsf{hb}} u_1$.

Again, the same observation applies if the access is a write. Theorems 1 and 2 allow us to reason about the invariance of a location in the presence of multithreading. In particular, we can work forwards and backwards from each read, defining an *interference-free region* in which the value read cannot be written by other threads. The region begins just after the first possible incoming edge before the read (e.g., a lock acquire or the start of the thread); the region ends just before the first possible outgoing edge after the read (e.g., a lock release or a spawn). If another thread writes to the variable of interest, that write must either happen-before or happen-after the memory access by the data-race-freedom assumption. By applying Theorems 1 and 2, we can conclude that any write must happen-before or happen-after the entire interference-free region.

B. Barriers

In this section we extend our analysis to cover rendezvous-style barriers. Our key observation is that even though barriers act like a release operation followed by an acquire action, effectively cutting off interference-free regions in both directions, we can still reason about interference-freedom if there are accesses to a variable both before and after a given invocation of a barrier.

Assume that the possible types of actions include notify and wait actions on barrier identifiers b:

$$a \quad ::= \quad \mathsf{notify}(b) \mid \mathsf{wait}(b) \mid \ \ldots$$

We assume that notify and wait actions have the following behavior:

Notify synchronizes-with wait: If $(t_1, \mathsf{notify}(b), u_1), (t_2, \mathsf{wait}(b), u_2) \in A$, then $u_1 <_{\mathsf{sw}} u_2$.

Notify and wait only synchronize-with each other: If $(t_1, a_1, u_1), (t_2, a_2, u_2) \in A$ and $u_1 <_{\mathsf{sw}} u_2$, then $a_1 = \mathsf{notify}(b)$ if and only if $a_2 = \mathsf{wait}(b)$.

Notify and wait are called in the proper order with no intervening synchronization: If a thread t calls notify and wait on a barrier b ($(t, \mathsf{notify}(b), u_1), (t, \mathsf{wait}(b), u_1') \in A$), then notify is called before wait ($u_1 \leq_{\mathsf{sb}} u_1'$), and the thread does not perform any synchronization actions between the calls to notify and wait. (Formally, if there exist u_2 and u_3 such that $u_1 \leq_{\mathsf{sb}} u_2 \leq_{\mathsf{sb}} u_1'$ and $u_2 <_{\mathsf{sw}} u_3$, then $u_2 = u_1$; if $u_1 \leq_{\mathsf{sb}} u_2 \leq_{\mathsf{sb}} u_1'$ and $u_3 <_{\mathsf{sw}} u_2$, then $u_2 = u_1'$.)

Note that we assume that each barrier is invoked only once per thread. Without this assumption, it would not always be the case that a notify always synchronizes-with a wait on the same barrier, nor that notify is sequenced-before wait with no intervening synchronization. Although real programs may wait on the same barrier multiple times, this is simply a convenience; it is possible to allocate a new barrier for every invocation. Moreover, we could easily make our formalism closer to reality by tagging every notify and wait action with a "barrier generation" that indicates how many times a thread has invoked this particular barrier, but such a change would not increase the expressiveness of the model.

The key insight of our analysis is that *nothing can happen-between a notify and a wait*. First, we prove two lemmas. The first lemma establishes that, if an action in another thread happens-after a notify, then either the action also

```
r1 = x;
...
pthread_mutex_lock(...);
...
pthread_mutex_lock(...);
...
// This call performs both the notify and
// wait actions.
pthread_barrier_wait(b);
...
pthread_mutex_unlock(...);
...
pthread_mutex_unlock(...);
...
r2 = x;
```

Figure 6. An example of an interference-free region around a barrier wait call.

happens-after the subsequent wait, or the happens-before chain between the notify and the remote action includes a synchronizes-with edge originating from the notify. This follows from our assumptions that the notify and wait are called in the proper order, and that a wait is never the source of an outgoing synchronizes-with edge. As this conclusion is reasonably obvious and the proof is a straightforward inductive case analysis, we omit the details of the proof.

Lemma 3 (Inversion of Happens-After-Notify)**.** *Suppose* $(t_1, \mathsf{notify}(b), u_1), (t_1, \mathsf{wait}(b), u_1'), (t_2, a, u_2) \in A$ *such that* $t_1 \neq t_2$ *and* $u_1 \leq_{\mathsf{hb}} u_2$*. Then either (1)* $u_1' \leq_{\mathsf{hb}} u_2$ *or (2) there exists* u_3 *such that* $u_1 <_{\mathsf{sw}} u_3 \leq_{\mathsf{hb}} u_2$ *.*

A symmetric lemma establishes that if a remote action happens-before a wait, then either the action also happens-before the notify, or the happens-before chain includes a synchronizes-with edge terminating at the wait.

Lemma 4 (Inversion of Happens-Before-Wait)**.** *Suppose* $(t_1, \mathsf{notify}(b), u_1), (t_1, \mathsf{wait}(b), u_1'), (t_2, a, u_2) \in A$ *such that* $t_1 \neq t_2$ *and* $u_2 \leq_{\mathsf{hb}} u_1'$*. Then either (1)* $u_2 \leq_{\mathsf{hb}} u_1$ *or (2) there exists* u_3 *such that* $u_2 \leq_{\mathsf{hb}} u_3 <_{\mathsf{sw}} u_1'$*.*

Given these two lemmas, the key theorem follows easily:

Theorem 3 (Nothing Happens-Between Notify and Wait)**.** *Suppose* $(t_1, \mathsf{notify}(b), u_1), (t_1, \mathsf{wait}(b), u_1'), (t_2, a, u_2) \in A$ *such that* $t_1 \neq t_2$*. Then it is not the case that* $u_1 \leq_{\mathsf{hb}} u_2 \leq_{\mathsf{hb}} u_1'$*.*

Proof. Assume $u_1 \leq_{\mathsf{hb}} u_2 \leq_{\mathsf{hb}} u_1'$. We will establish a contradiction by proving that the happens-before relation is no longer acyclic.

By Lemma 3, either $u_1' \leq_{\mathsf{hb}} u_2$ or there exists u_3 such that $u_1 <_{\mathsf{sw}} u_3 \leq_{\mathsf{hb}} u_2$. The first case is clearly a violation of the acyclicity of happens-before, as we have that $u_2 \leq_{\mathsf{hb}} u_1'$, and we know $u_1' \neq u_2$ because they are from different threads. Similarly, we can rule out the first case for Lemma 4. Therefore (using u_4 as the witness first the second case of Lemma 4), we have that $u_1 <_{\mathsf{sw}} u_3 \leq_{\mathsf{hb}} u_2 \leq_{\mathsf{hb}} u_4 <_{\mathsf{sw}} u_1'$. By assumption, u_3 must correspond to an action $\mathsf{wait}(b)$, and u_4 must correspond to an action $\mathsf{notify}(b)$. Therefore $u_4 <_{\mathsf{sw}} u_3$, which creates a cycle in the happens-before relation: $u_3 \leq_{\mathsf{hb}} u_4$ and $u_4 \leq_{\mathsf{hb}} u_3$. We know that $u_3 \neq u_4$ as they are notify and wait actions, respectively. Therefore we have established a contradiction, so the theorem is proved. □

We can combine Theorem 3 with Theorems 1 and 2 to infer larger interference-free regions. For instance, consider the code in Figure 6. The call to pthread_barrier_wait performs both the notify and wait actions, satisfying the requirement that these actions occur in the proper order with no intervening synchronization. Suppose a remote write to x were to happen-after the first read and happen-before the second. By Theorem 1, the write must happen-after the notify; by Theorem 2, the write must happen-before the wait. Therefore, by Theorem 3, no such write exists, so the two reads see the same value. In effect, the interference-free region for x extends across the call to pthread_barrier_wait—but only because there are reads of x before and after the barrier. Else the remote write could happen-before or happen-after the barrier.