2011

Accentuating the Positive: Atomicity Inference and Enforcement Using Correct Executions

Dasarath Weeratunge
Purdue University, dweeratu@cs.purdue.edu

Xiangyu Zhang
Purdue University, xyzhang@cs.purdue.edu

Suresh Jagannathan
Purdue University, suresh@cs.purdue.edu

Report Number:
11-002

Accentuating the Positive: Atomicity Inference and Enforcement Using Correct Executions

Dasarath Weeratunge Xiangyu Zhang Suresh Jaganathan

Department of Computer Science, Purdue University

{dweeratu,xyzhang,suresh}@cs.purdue.edu

Abstract

Concurrency bugs are often due to inadequate synchronization that fail to prevent specific (undesirable) thread interleavings. Such errors, often referred to as *Heisenbugs*, are difficult to detect, prevent, and repair. In this paper, we present a new technique to increase program robustness against Heisenbugs. We profile correct executions from provided test suites to infer fine-grained atomicity properties. Additional deadlock-free locking is injected into the program to guarantee these properties hold on production runs. Notably, our technique does not rely on witnessing or analyzing erroneous executions.

The end result is a scheme that only permits executions which are guaranteed to preserve the atomicity properties derived from the profile. Evaluation results on large, real-world, open-source programs show that our technique can effectively suppress subtle concurrency bugs, with small runtime overheads (typically less than 15%).

1. Introduction

Debugging concurrent programs is challenging because of non-determinism induced by schedulers and unintended racy behavior. Consequently, bugs that do manifest are often not easily reproduced – even though a program may yield many different results on the same input, only a small fraction of these may be erroneous; these failures are called *Heisenbugs*. There has been much recent work devoted to discovering these bugs. For example, logging and replay techniques [13, 22] monitor program executions and allow replay when failures do occur, albeit at the expense of sometimes substantial space and time overheads during normal execution. Alternatively, one could enrich testing strategies to discover specific interleavings that lead to such failures; these techniques perform a directed and bounded search over program executions [23, 26], but assume that the failures have been observed and the failure inducing inputs are provided. Unfortunately, discovering such inputs during testing for large code bases is often problematic.

Even if a specific interleaving can be automatically discovered to consistently reproduce a failure, the onus for understanding the reason for the failure, and more importantly preventing it, remains squarely on the programmer. Heisenbugs are triggered by race conditions, atomicity violations, and unintended dependence ordering violations. In general, these conditions do not directly and immediately lead to failures, but gradually contaminate program state; thus, the actual point where a failure is triggered may be far removed from the primary cause. Even if the root cause can be identified, preventing the bug from occurring may require substantial non-local reasoning. For example, preventing an atomicity violation by adding locks may have the unintended consequence of introducing a deadlock, or unnecessarily limiting concurrency.

We believe that expecting programmers to repair Heisenbugs by tedious analysis and meticulous reasoning of failed executions is untenable for complex real-world applications. An alternative approach explored in this paper takes advantage of the fact that Heisenbugs occur rarely – thus, the expected behavior of an execution is *not* to manifest the error. This observation leads us to focus on correct executions (the common case) to infer apparent atomicity properties. For our purposes, these properties capture pairs of thread-local accesses to a shared variable that occur without interleaved remote accesses from other threads. Significantly, we do not require that these pairs be related lexically; thus, they can cross different function boundaries and conditional branches. Our technique considers these pairwise accesses as a set of constraints. A path-sensitive locking scheme is derived by solving these constraints. Acquisitions and releases of a set of new locks, generated by our analysis, are subsequently injected into the program. The locking scheme is safe as it only allows a subset of executions that are allowed in the original program - this subset is guaranteed to produce behavior consistent with the atomicity properties observed in the profile. Because we infer atomic accesses that are substantially more fine-grained than what programmers can easily express using lexical bracketing of atomic code regions, the approach enables effective *suppression* of Heisenbugs with little loss of concurrency, and without requiring programmers to employ the low-level and non-modular reasoning that would otherwise be necessary to remedy the error.

Thus, the distinguishing feature of our technique is that it does *not* require the availability of erroneous executions to prevent concurrency bugs that arise because of an atomicity violation. Instead, the injected instrumentation effectively only admits executions that respect the atomic regions inferred by profiling correct runs. Notably, our approach is not replay-based: the instrumented program allows different interleavings from those witnessed in the profile with respect to non-atomic regions, only guaranteeing the absence of interleaved accesses in those that are. Moreover, the technique is safe: any execution realizable under the instrumented program is also realizable under the original; a corollary of our safety condition is that our injected instrumentation does not introduce deadlocks.

Our contributions are summarized as follows:

- We propose a novel profile-based program analysis that can be used to suppress Heisenbugs for complex, large-scale concurrent applications.
- Central to our approach is the inference of salient atomicity properties from profiles of benign executions generated from test suites, without requiring the availability of failure-inducing inputs.
- Our instrumentation algorithm injects *path-sensitive* lock acquisitions and releases to tolerate atomic sections that are not lexical (e.g., because the atomic region may span different lexically-delimited scopes).
- We evaluate our technique on a set of large, real-world open-source programs. Benchmark results show that our technique can effectively suppress subtle hard-to-identify and repair concurrency bugs. Performance evaluation on realistic workloads indicate that the runtime overhead of our technique is small, typically on the order of 15%.

The remainder of the paper is structured as follows. The next section presents additional motivation and some details of the benchmark corpus we used in our study. Section 3 examines one of these benchmarks as a case study to illustrate some of the technical challenges our technique must address. Section 4 gives an overview of our solution using this case study as a representative example. We present the scope of our technique with respect to the kinds of concurrency bugs it can suppress in Section 5. We present details of the profiler in Section 6. The lock placement algorithm and instrumentation mechanism is given in Section 7. The deadlock resolution algorithm is described in Section 8. Extensions to our approach are required to deal with conditional synchronization and shared heap objects; these extensions are described in Section 9. We formalize the safety properties of our solution in Section **??**. An evaluation study with respect to overhead and effectiveness is given in Section 11. Related work is given in Section 12 and conclusions are provided in Section 13.

2. Motivation

Existing concurrency bug detection/fixing techniques [26, 18] often rely on exploiting *negative* information - given the observation of a known or likely concurrency bug, acquire an input that triggers the bug. The implication is that bug remediation first requires either manifestation of the bug or identification of likely sources. In contrast, our technique infers potential atomic regions from observed correct executions, and enforces atomicity in these regions, thereby guaranteeing that atomicity invariants witnessed in these executions are automatically preserved; by doing so, it prevents bugs that arise because an execution fails to preserve these invariants without the need for having a failure-inducing input.

We conducted a detailed evaluation study to support our intuition. Our study looked at 13 concurrency bugs from 6 widely used concurrent applications including the `apache` web server and the `mysql` database server. (Table 1) `Mysql` comes with an extensive regression test suite, as well as several widely used database performance evaluation benchmarks such as `sysbench-oltp` and `tpcc`. `Apache` and the other programs do not have a their own test suites, but there are commonly used static/dynamic workloads that can be used as representative inputs. The test inputs used for profiling each benchmark are shown in Table 2.

These programs run correctly with the test inputs. We collected atomicity profiles from these runs and analyzed whether enforcing these properties would suppress future bugs, i.e. bugs reported after the program and the test suites were released. Our atomicity criteria was based on observed *pairwise* atomic sections: a code region (not necessarily lexical) bracketed by two accesses to the same variable in the same thread, without any intervening access to that variable by another thread.

We observed that the test suites collectively provide coverage over the faulty statements, even though they do not trigger the failures. Indeed, the key atomic pairs are easily observed from the passing test cases; these regions induce a failure only with certain inputs that do not happen to be used in the regressions and benchmarks, or under certain unforeseen schedules. By enforcing atomicity in these regions (i.e., by injecting suitable locks), we were able to successfully prevent these bugs: inputs identified from the bug reports that were previously able to trigger the bug no longer did, and the inserted locks prohibited previously allowed faulty schedules. Thus, the transformed program suppresses these bugs from occurring in the first place. Many of these bugs such as `mysql` 12848, 38739, 16333, 20850, and `apache` 44402 were difficult to deterministically reproduce and diagnose when they were reported, due to the special inputs required, the large number of threads involved and the long execution time necessary to reach the failure. Indeed, several of the bugs we consider have not been studied in the research literature to the best of our knowledge. Note that we focus on reported bugs only because it simplifies validation.

program	bug, date	module affected, description	atomic pair
aget, 0.4	-	A data race between threads downloading a web page and SIGINT signal handler. When the data race occurs, the downloaded file is corrupted. The shared variable bwritten is consistently protected by lock bwritten_mutex except inside the signal handler.	bwritten needs read lock in save_log() in Resume.c:41.
pbzip2, 2.094	-	An order violation between the main thread and threads doing file compression (consumer threads) on mutex `fifo->mut`. The main thread may delete the mutex before all consumer threads are done using it.	`fifo->mut` needs a read lock in func. consumer() in pbzip2.cpp:873-994
nszip, 1.8	342577, 23/6/2006	A data race in Mozilla nsZipArchive::SeekToItem on nsZipItem::flags. The data race corrupts the decompressed file.	nsZipItem::flags needs write lock in func. nsZipArchive::SeekToItem() in nsZipArchive.cpp:1381-1408
apache, 2.2.6	44402, 2/12/2008	In worker multi-processing module (mpm), the recycled pools list gets corrupted under high concurrency causing server to crash. First observed while running specweb99 static content workload with 1000 simultaneous connections and 500 threads per worker. The server would run for anything between 10 minutes to 4 hours before it would crash.	recycled_pool::next in fdqueue.c:104-107, func. ap_queue_info_set_idle() needs read lock.
apache, 2.0-head	25520, 15/12/2003	In mod_log_config, log lines are corrupted at high volumes in access log.	buffered_log::outcnt needs write lock in mod_log_config.c:1432-1469, in func. ap_buffered_log_writer()
spider-monkey, 1.5	133773, 27/3/2002	An order violation bug in Mozilla JavaScript engine in func. js_DestroyContext() on JSRuntime::state leads to crash.	JSRuntime::state needs read lock in func. main().
mysql, 4.0.12	791, 4/7/2003	In log.cc (binlog), when SQL FLUSH LOGS is executed concurrently with another SQL statement, e.g. SQL INSERT, the latter may not be recorded in binlog.	MYSQL_LOG::log_type needs write lock in log.cc:867-869, func. MYSQL_LOG::new_file()
mysql, 4.0.12	12848, 29/8/2005	In sql_cache.cc (query cache), access to query cache while the cache is been resized caused server to crash. The bug was reproduced by running query_cache.test (size: 450 lines) in mysql-test suite for 1-2min.	Query_cache::bins needs write lock in sql_cache.cc:733-755, in func. Query_cache::resize()
mysql, 5.0.16	14747, 8/9/2005	In the index tree adaptive search module in Innobase database engine, two threads trying to drop a hash index could race with one another, leading to a server crash. After the bug was first reported, the developers added diagnostics to the code and there was no activity for several months until it was reported again. Even then there was no way to reliably reproduce it except for the observation that crash happened often on big queries.	buf_block_struct::index needs write lock in btr0sea.c:893-1015 in func. btr_search_drop_page_hash_index()
mysql, 6.0.6	38739, 12/8/2008	Transaction Manager, Falcon database engine; In TransactionManager::startTransaction() it was possible for two threads to pick up the same transaction object for reuse simultaneously leading to an assertion failure. The developers used 30 client threads that execute single select statement for a random record from a single table to reproduce this bug.	Transaction::state needs write lock in TransactionManager.cpp:119-120, in func. startTransaction()
mysql, 6.0.6	35714, 31/3/2008	A data race between THD::awake() and thd_scheduler::thread_detach() on varaible THD::mysys_var led to a server crash.	THD::mysys_var needs read lock sql_class.cc:859-893 in THD::awake().
mysql, 5.0.19	16333, 10/1/2006	In the safe_mutex API, assertion failure occured in safe_mutex_assert_not_owner() due to inconsistent values in fields `count` and `thread` in safe_mutex objects. The crash was reproduced by running oltp test in sysbench benchmark. The server would crash approx. 18min into the run when using 8 client connections and a test database having 1 million records.	A multi-variable atomicity violation. safe_mutex_t::count and safe_mutex_t::thread need write lock in safe_mutex_lock() in thr_mutex.c:163-170
mysql, 5.0.24	20850, 4/7/2006	A data race occured between server termination code in end_slave, and slave threads on variable Master_info. The func. end_slave could destroy the Master_info object before all slaves were done using it. To reproduce the bug the developers ran MySQL replication tests (size: 80 lines) in a loop with each iteration restarting the server. It many take as many as 36 iterations before the server would crash and the crash happens during server shutdown.	Master_info::run_lock needs read lock in func. handle_slave_io() in slave.cc:3414-3745

Table 1. Some known data race/atomicity violation/order violation bugs and how they can be prevented by enforcing pairwise atomicity observed in passing runs. Bugs 44402, 14747, 38739, 16333 have not been discussed in literature before. Many of these bugs such as `mysql` 12848, 38739, 16333, 20850, and `apache` 44402 were difficult to deterministically reproduce and diagnose according to the bug reports, due to the special inputs required, the large number of threads involved, and the long execution time necessary to trigger the failure. Please refer to Table. 2 for inputs used for profiling.

```
foo()
{
  if (<COND1>){
    P = NULL;
  }

  if (P){
   ... = *P;
  }
}
```

```
foo()
{
  if (<COND1>){
    acquire(L)
    P = NULL;
    release(L)
  }

  acquire(L)
  if (P){
    ... = *P;
  }
  release(L)
}
```

Figure 1. An input dependent atomicity violation. The original code is shown on the left; the version instrumented by our technique is on the right.

To make our discussion concrete, consider the example in figure 1, which is an abstraction of several real atomicity violations we have observed. Here, P is a shared pointer which is not protected consistently with a lock. The function foo() is called by multiple threads, leading to interleaving of accesses to P. The assignment that sets P to NULL is guarded by the condition COND1. For all test inputs found in the regression test suite, COND1 is always false. Hence, it is not possible to induce the bug even after exhaustively exploring the possible schedules for the test runs.

Consequently, we do not observe any writes to P from other threads; our profile, on the other hand, indicates that the two reads to P are pairwise atomic – they occur with no intervening access to P by another thread. Since the reads are not protected, our technique transforms the program to put these accesses within an atomic section. In addition, even though we have not seen the execution of the assignment to P in our profile, enforcing the atomicity of the two reads requires us to also instrument the write access to ensure that it is not allowed to be interleaved with the reads. Our lock injection mechanism guarantees that even when COND1 is true (under some new input provided later during a production run), two threads will never be interleaved within foo() such that one sets P to NULL between the two read accesses to P.

3. Technical Challenges

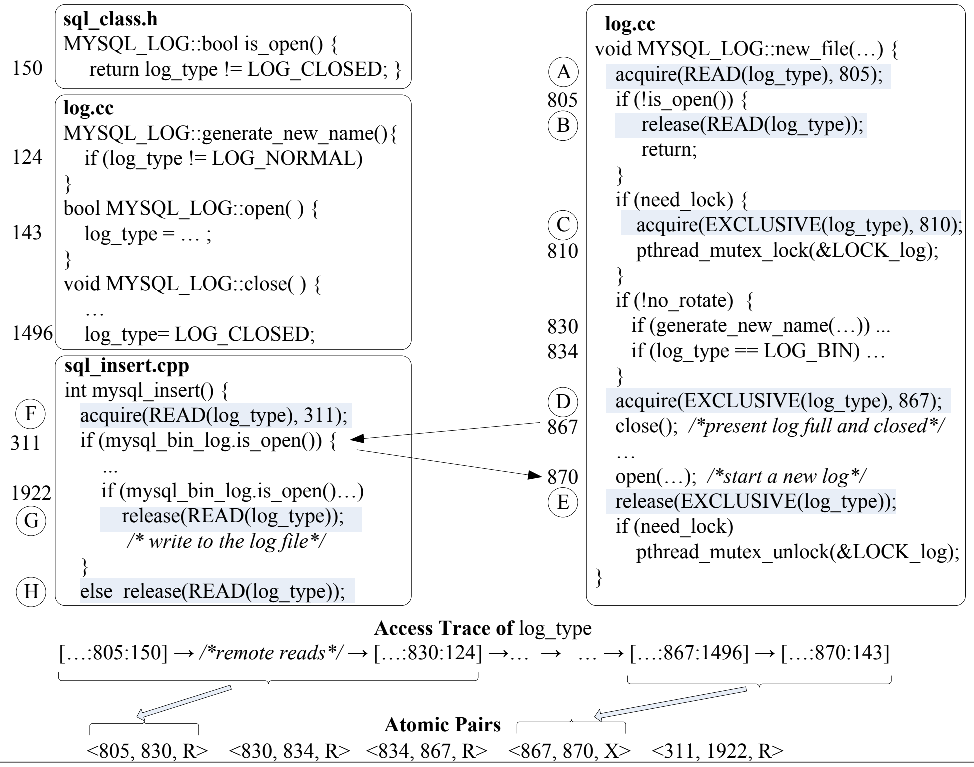
Realizing our ideas for real world programs is challenging. We use a concurrency bug found in MySQL-4.0.12 (bug id 791) to illustrate these challenges. This non-deterministic bug results in SQL queries not being properly logged. Logging failure is problematic beause mysql relies on its log to revoke and replay queries in the presence of transactional commit failures. The failure is caused by an atomicity violation. The relevant code snippets are shown in Fig. 2. Ignore all the highlighted statements for the moment; these are statements inserted by our technique, and are not present in the original program. At line 867 in function new_file() (the right-hand side code fragment), the current log file reaches its size limit and is closed. A new log file is supposed to be created at line 870 to continue logging. Although there is a program lock LOCK_log that protects the main body of new_file(), the lock is not consistently held in other accesses of log_type. For instance, in function mysql_insert() (shown on the left-hand side), the call to mysql_bin_log.is_open() at line 311 entails accessing log_type (see the definition of is_open() on the top-left of the figure), yet the access is not protected. In the failure-inducing schedule as shown by the arrows between the lines 311, 867 and 870, the value is set to LOG_CLOSED by the invocation to close() first at line 867, the predicate at 311 is evaluated before the value is set again at line 870 and thus takes the false branch, with the insert query not logged.

Like most Heisenbugs, this error illustrates two significant complexities: (a) finding the bug is challenging, and not likely to be easily exposed by typical testing strategies; and (b) even after the bug is found, there is a deep causal chain of dependencies that must be followed in order to determine how best to repair it. For example, given knowledge of the bug and its cause, we might be tempted to lock the call to is_open() at line 311. However, non-trivial reasoning must be brought to bear to ensure that such a change is in fact correct:

- It is unclear if deadlocks would be introduced.
- The body of new_file() is protected by multiple locks (not shown in the figure). There may be many other accesses to log_type just like that at line 311, with some of them already protected by some subset of the locks in new_file(). It is hence difficult to determine which lock(s) should be held in order to provide consistent protection.
- Wrapping the access at line 311 with lock LOCK_log would still not resolve the bug because there is another access to log_type in the call to is_open() at line 1922. Atomicity must be preserved between these two accesses as well.
- We might choose to simply protect the whole body of mysql_insert() with lock LOCK_log. However, doing so likely limits concurrency unnecessarily since only the two accesses to the log need to be protected. The execution of the rest of the function body can be interleaved with other threads.

4. Overview of Our Solution

Our technique is divided in three phases: a *profiling* phase collects pairwise atomicity information; a *lock placement* phase derives a locking scheme that satisfies the atomicity constraints collected by the profiler; and a *deadlock resolution* phase resolves possible deadlocks induced by the in-



```
sql_class.h
     MYSQL_LOG::bool is_open() {
150      return log_type != LOG_CLOSED; }

log.cc
     MYSQL_LOG::generate_new_name(){
124    if (log_type != LOG_NORMAL)
     }
     bool MYSQL_LOG::open( ) {
143    log_type = … ;
     }
     void MYSQL_LOG::close( ) {
       …
1496   log_type= LOG_CLOSED;

sql_insert.cpp
     int mysql_insert() {
Ⓕ      acquire(READ(log_type), 311);
311    if (mysql_bin_log.is_open()) {
         …
1922     if (mysql_bin_log.is_open()…)
Ⓖ          release(READ(log_type));
           /* write to the log file*/
       }
Ⓗ      else  release(READ(log_type));
```

```
log.cc
     void MYSQL_LOG::new_file(…) {
Ⓐ      acquire(READ(log_type), 805);
805    if (!is_open()) {
Ⓑ        release(READ(log_type));
         return;
       }
       if (need_lock) {
Ⓒ        acquire(EXCLUSIVE(log_type), 810);
810      pthread_mutex_lock(&LOCK_log);
       }
       if (!no_rotate)  {
830      if (generate_new_name(…)) ...
834      if (log_type == LOG_BIN) …
       }
Ⓓ      acquire(EXCLUSIVE(log_type), 867);
867    close();  /*present log full and closed*/
       …
870    open(…);  /*start a new log*/
Ⓔ      release(EXCLUSIVE(log_type));
       if (need_lock)
         pthread_mutex_unlock(&LOCK_log);
     }
```

Access Trace of log_type

[…:805:150] → /*remote reads*/ → […:830:124] →… → … → […:867:1496] → […:870:143]

Atomic Pairs

<805, 830, R> <830, 834, R> <834, 867, R> <867, 870, X> <311, 1922, R>

Figure 2. A real bug in MySQL-4.0.12. The arrows between statements show the failure inducing schedule before applying our technique. The shared variable is `log_type`. The access trace of the variable is shown below the code snippets. Each entry of the trace is an access with context information. The pairwise atomicity profile is presented in the bottom. Each profile entry is a triple. The first two elements are a pair of intraprocedural program points with atomicity. The last element is the atomicity type. R means *read atomicity*, i.e., remote reads have been observed to be interleaved with this pair. X means *exclusive atomicity*, i.e. no interleaving remote access has been observed in the profile. The shaded statements are instrumentations added by our technique. Primitive `acquire(READ(log_type),805)` means acquiring a read lock for variable `log_type`, starting at 805.

serted locks. We illustrate these steps using the example in Fig. 2 in the remainder of the section.

Profiling. Our technique first collects a set of traces. To simplify our discussion, we only present a trace relevant to variable `log_type` below the code snippets. Each trace entry represents an access annotated with its context. For instance, the first entry $[... : 805 : 150]$ means that the access is the read at line 150 in function `is_open()`, which was invoked at line 805, and so on.

We then consider each pair of consecutive thread-local accesses to see if they have ever been interleaved with accesses from other threads. For a given pair of local accesses, we identify an intraprocedural path such that locking that path ensures atomicity of these accesses. Thus, our instrumentation phase does not need to inject locking over arbitrary inter-procedural paths.

In the example, the common function body of the first two local entries is `new_file`, and the two accesses to `log_type` are interleaved with remote reads. These accesses are thus aggregated to the atomic pair $\langle 805, 830, \mathit{READ}\rangle$, permitting read atomicity (i.e. remote reads are allowed but remote writes are not) between lines 805 and 830. Similarly, since no interleaved remote accesses have been observed between entries $[... : 867 : 1496]$ and $[... : 870 : 143]$, which correspond to the accesses in `close()` and `open()` respectively, the atomic pair $\langle 867, 870, \mathit{EXCLUSIVE}\rangle$ is derived as shown, representing exclusive atomicity (i.e. no remote reads and writes are allowed).

Lock placement. Given a profiled trace, we infer synchronizations that satisify all pairwise constraints. The highlighted statements are the synchronizations introduced. Ignore the lock acquisition at Ⓒ for the moment. A read-lock is acquired at Ⓐ to ensure read atomicity from 805 to 830. The lock is released at Ⓑ since taking the false branch at 805 indicates line 830 is no longer reachable. The read atomic pairs $\langle 830, 834, \mathit{READ}\rangle$ and $\langle 834, 867, \mathit{READ}\rangle$ do not lead to new acquisitions as they can be considered as a continuation of the pair from 805 to 830. The lock is upgraded to exclusive at Ⓓ based on $\langle 867, 870, \mathit{EXCLUSIVE}\rangle$. It is released at Ⓔ, right after the access in 870. This avoids unnecessarily limiting concurrency.

Similarly, the atomic pair $\langle 311, 1922, \mathit{READ}\rangle$ entails the acquisition at Ⓕ and the release at Ⓖ and Ⓗ. The latter release is because 1922 is not reachable. Observe that our acquisition primitives take the line number as a parameter to achieve intra-procedural path-sensitive locking, explained in more detail in Section 7.

Deadlock resolution. Observe that our technique augments but does not remove existing synchronization actions. Correctness arguments dictate that instrumentation should not alter the original program semantics. However, adding locks may interfere with existing ones or may interfere (e.g., cause deadlock) with locks added for other variables. Our technique resolves deadlocks by injecting additional acquisitions. Consider the synchronizations added to function `new_file()` by our technique (ignoring Ⓒ for now). There is a potential deadlock due to the interference with program lock `LOCK_log`. In particular, the program lock may be acquired (at 810) when the read lock for `log_type` is held (at Ⓐ); and in a different thread, the exclusive lock may be acquired (at Ⓓ) when the program lock is held (at 810). Our technique breaks the cycle by adding an extra `acquire()` (at Ⓒ)[1]. This operation ensures that the exclusive lock for `log_type` is always held when the program lock is acquired. It is straightforward to see that the resulting instrumented program effectively prevents the bug.

5. Scope

Many common kinds of concurrency bugs, such as data races and order violations, often involve violation of atomicity invariants. As long as a bug manifests such a violation, it is feasible to apply our technique to prevent it.

```
thread1() {                  thread2() {
   acquire(L)
A: p = ...;
   release(L)
                             B:  p = null;

   acquire(L)
C: *p;                       }
   release(L)
}
```

Figure 3. A data race involving an atomicity violation.

We have observed that many bugs reported as data races entail violation of pairwise atomicity. Fig. 3 presents a simplified, yet typical, example. Here, executing statements `A`, `B` and `C` in that order results in a null pointer dereference. It might be assumed that the bug arises because of a race between `B` and `C`. However, if correct executions exhibit pairwise atomicity between the access to `p` at `C` and its preceding access at `A`, enforcing atomicity between these two statements would effectively suppress the bug. `Mysql` bugs 14747, 35714, and 20850 in Table 1 belong to this category.

Order violations are a type of concurrency bug that often involve atomicity violations as well. An order violation occurs when accesses to a shared variable in different threads fail to execute these accesses in the specific order in which they were intended to be executed. For example, a socket may be intended to be first created and initialized in one thread, and then used in another. We observe that in many cases, ensuring a specific order can be preserved by enforcing atomicity. In the case of sockets, this would mean that the creation and initialization of the socket should be atomic. The `spidermonkey` bug and the `pbzip-2` bug in Table 1 belong to this category.

So far, we have assumed pairwise atomicity invariants are defined with respect to the same variables. But in practice, there are cases in which atomicity is required to hold across multiple variables. For example, a flag that must be set if the corresponding pointer becomes null is an instance in which atomic accesses to two variables is required. Breaking such atomicity can lead to failures. We leverage existing techniques (e.g., [19]) to identify correlated shared variables (essentially those that are accessed together in most cases), representing them as a single abstract shared variable.

Other concurrency bugs that do not necessarily involve atomicity violations, such as those that lead to deadlocks, are not handled by our technique.

6. Profiling Atomic Pairs

TRACE-LANGUAGE $\mathcal{L}$

$$\begin{array}{rcl}
S \in \mathcal{L} & ::= & \langle v, T\rangle;\ S \ \mid\ \epsilon \\
T \in VarTrace & ::= & \langle t, rw, c, is\rangle;\ T \ \mid\ \epsilon \\
t \in Thread & ::= & \{t_1, t_2, ...\} \\
c \in Context & ::= & \bar{l} \\
rw \in AccessType & ::= & \{\texttt{Rd}, \texttt{Wr}\} \\
v \in Var & ::= & \{x, y, z, ...\} \\
is \in LockInstanceSet & ::= & \mathcal{P}(Lock \times Int) \\
Lock & ::= & \{k_1, k_2, ...\} \\
l \in Label & ::= & \{\texttt{entry}, \texttt{exit}, l_1, l_2, ...\}
\end{array}$$

Figure 4. Trace Syntax

The first component of our technique is a profiler that identifies atomic access pairs. We say two local accesses to a variable observed by a profile are *exclusive-atomic* if no remote (i.e., non-thread-local) accesses occur between them. Any execution which respects the behavior witnessed by the profile ensures that an exclusive lock on that variable is acquired prior to the first access, and released after the second. Two local acceses are *read-atomic* if only remote reads occur between the two accesses, without the presence of an intervening remote write. To mimic the conditions of the profiled run on subsequent executions, we require that a read lock be held for this pair. If a remote write was ever observed between the two accesses in the profile, then no atomicity condition holds (denoted as *no-atomic*). We distinguish *read-atomic* from *exclusive-atomic* to allow

[1] Our runtime allows a lock to be acquired multiple times.

more concurrency. It is an important design choice given the non-trivial number of atomic invariants we need to enforce.

Traditonally, atomicity is defined lexically by the programmer – thus, we typically reason about atomicity in terms of methods or code blocks. Transplanting such notions into a profiling context is not straightforward, however. Lexically-scoped atomic regions are difficult to identify and profile because (a) multicore architectures permit real concurrency, allowing any code region to execute concurrently with code regions in other threads, confounding easy identification of lexical region interleavings; and (b) it is hard to aggregate region-based profiles; the same region may occur in the profile multiple times, exhibiting atomicity in some instances, but not others. We cannot simply mark the whole region as not being atomic because sub-regions (before and after a given remote access) may still be atomic. Thus, identifying atomicity using lexically-scoped code regions could be sub-optimal if atomicity is only needed in a small portion of such regions.

Profiling atomic pairs, instead of code regions, addresses these issues. The variable sensitive locking that can be derived by discovering these pairs allows arbitrary (non-lexical) interleavings of atomic regions. In this sense, atomic pairs provide atomicity at fine granularity, and facilitate aggregation of larger atomic regions from smaller atomic pairs.

To realize our design, we need to overcome a number of technical challenges. Most importantly, atomic pairs are not lexically well formed; for example, two accesses could occur in different calling contexts. For instance, function $A()$ might call $B()$, with the first access occurring in $B()$; after $B()$ returns, $A()$ might subsequently call $C()$, which performs the second access. Moreover, existing synchronization present in the program may contribute to the atomicity properties observed in the profile. Hence, we should exclude identifying atomic accesses that are already guaranteed based on program structure.

We instrument programs to generate traces, which are then analyzed to identify atomic pairs. The trace syntax is presented in Fig. 4. A trace S is a sequence of variable subtraces, which define the access history of a shared variable. The reason for such a design is that there is a sequential order for accesses to the same variable instance while such an order may not be available across variables. An entry in the variable trace represents a shared variable access, consisting of the thread id t, the calling context c, the access type rw, and the lock instance set is. The lock instance set describes the set of lock instances held at the access point. A lock may have multiple instances at runtime, with an instance generated by an acquisition and destroyed by the corresponding release. Hence, we use a pair $\langle k, i\rangle$ to represent the ith instance of lock k.

The profiling rules are presented in Fig. 5. The profiler takes a trace as input and generates a set of access pairs for each shared variable. Each pair is associated with a type that could be no-atomic ($NONE$), read-atomic ($READ$), exclusive-atomic ($EXCLUSIVE$), or top ($\top$); the ordering relationship among these types form a simple lattice as defined in Fig. 5. In order to determine if the atomicity of a pair is already guaranteed by the program, we introduce two relations: the pair lock set L describing the locks that guard both accesses; and, the lock set LS for a variable describing the set of common locks held by all accesses of the variable.

The rules are divided into three sets, the first describing the evaluation of a variable trace, the second computing variable lock sets, and the third computing the atomic pairs. The triple defining a trace evaluation configuration, $\langle T, L, Z\rangle$ contains a trace T, an access pair lockset L, and an access pair atomicity set Z. Rule *(Exclusive)* considers the case when two consecutive local accesses are not interleaved with any remote access. To ensure that subsequent executions respect this observed atomicity, we must ensure that the instrumented version of the program is (a) adequately augmented with locks to enforce atomicity, and (b) the locking protocol is consistent with all other observed accesses of this pair. To address (a), we appeal to the definition of *frontier* shown in the figure. Suppose the first access (within the procedure labeled l_4) occurs in context: $c_1 = [l_1 : l_2 : l_3 : l_4]$ and the second (within the procedure labeled l_6) occurs in context $c_2 = [l_1 : l_2 : l_5 : l_6]$. The frontier of these two contexts is defined to be $frontier(c_1, c_2) = \langle l_3, l_5\rangle$. Thus, the frontier identifies l_3 and l_5 as the call-sites found within l_2 that encapsulate the access pair. We use this information to insert an exclusive lock around the region that encloses l_3 and l_5 in procedure l_2 to provide atomicity guarantees without having lock acquisitions and releases span procedure boundaries. If the locks necessary to provide atomicity of the pair were not hoisted upto l_2, but instead acquired within l_4 and released within l_6, runtime disambiguation would be necessary to consider the different possible call paths to l_4 (and l_6), to ensure that the lock is only acquired in the profiled contexts. In the rule, the set s represents the locks that are held at both l_1 and l_2. By intersecting this lock set with $L(l_1, l_2)$, the set of locks that protect all occurrences of access pair $\langle l_1, l_2\rangle$, we define the minimum set of locks that are used to protect this pair. Consider two occurrences of pair $\langle l_1, l_2\rangle$ in which there is no intervening remote operation between the first pair, but there is one between the second. The atomicity requirement on this pair is naturally dictated by the *weakest* observed action; in this case, the observed remote operation between the second pair of accesses precludes treating the access pair bracketed by l_1 and l_2 as atomic.

Rule $(Read)$ defines the conditions under which an intervening remote read operation occurs between an access pair, and rule $None$ defines the conditions under which an intervening remote write operation takes place. In the latter case, no additional synchronization is required to enforce executions faithful to the profile; in the case of remote reads, read locks can be used to permit executions to admit inter-

DEFINITIONS

$$\alpha \in \mathit{AtomicType} ::= \{\mathit{NONE},\ \mathit{READ},\ \mathit{EXCLUSIVE}, \top\},\ \top >\ \mathit{EXCLUSIVE} >\ \mathit{READ} >\ \mathit{NONE}.$$

$$Z \in \mathit{Pairs} \quad ::= \mathit{Label} \times \mathit{Label} \longrightarrow\ \mathit{AtomicType} \qquad L \in \mathit{PairLockSet} \quad ::= \mathit{Label} \times \mathit{Label} \longrightarrow\ \mathcal{P}(\mathit{Lock})$$

$$A \in \mathit{VarPairs} \quad ::= \mathit{Var} \longrightarrow\ \mathcal{P}(\mathit{Label} \times \mathit{Label} \times \mathit{AtomicType}) \qquad \mathit{LS} \in \mathcal{P}(\mathit{Lock})$$

$$\mathit{frontier}(c_1 \cdot l_1, c_2 \cdot l_2) = \begin{cases} \langle l_1, l_2 \rangle & \texttt{if}\ c_1 = c_2; \\ \mathit{frontier}(c_1 \cdot l_1, c_2) & \texttt{if}\ c_1 \subset c_2; \\ \mathit{frontier}(c_1, c_2 \cdot l_2) & \texttt{if}\ c_2 \subset c_1; \\ \mathit{frontier}(c_1, c_2) & \texttt{otherwise}. \end{cases} \qquad \begin{array}{l} \mathcal{L} = \{k_1, \ldots, k_n | \texttt{new\_lock()}^{k_i} \in \mathit{Prog}\} \\ L_0(l_1, l_2) = \mathcal{L} \\ Z_0(l_1, l_2) = \top \end{array}$$

DESCRIPTION

Input: A trace S.
Output: A mapping A that maps a variable to a set of atomic pairs.

PROCESSING PER-VARIABLE TRACE

$$\frac{\boxed{\langle l_1, l_2 \rangle = \mathit{frontier}(c_1, c_2) \quad s = \{k | (k, i) \in is_1 \cap is_2\} \quad L' = L[\langle l_1, l_2 \rangle\ \mapsto\ L(l_1, l_2) \cap s]}^{(1)} \quad Z' = Z[\langle l_1, l_2 \rangle\ \mapsto\ \mathit{min}(Z(l_1, l_2), \mathit{EXCLUSIVE})]}{\langle T; \langle t, rw_1, c_1, is_1 \rangle;\ \langle t, rw_2, c_2, is_2 \rangle,\ L,\ Z \rangle \overset{\mathit{inst}}{\Longrightarrow} \langle T; \langle t, rw_1, c_1, is_1 \rangle,\ L',\ Z' \rangle} \quad (\mathit{Exclusive})$$

$$\frac{T_2 \neq \texttt{nil} \quad \neg\exists \langle t, rw_x, c_x, is_x \rangle \in T_2 \quad \neg\exists \langle t', \texttt{Wr}, c', is' \rangle \in T_2 \quad t \neq t' \quad \text{Condition (1) from rule } (\mathit{Exclusive}) \quad Z' = Z[\langle l_1, l_2 \rangle\ \mapsto\ \mathit{min}(Z(l_1, l_2), \mathit{READ})]}{\langle T_1; \langle t, rw_1, c_1, is_1 \rangle;\ T_2;\ \langle t, rw_2, c_2, is_2 \rangle,\ L,\ Z \rangle \overset{\mathit{inst}}{\Longrightarrow} \langle T_1; \langle t, rw_1, c_1, is_1 \rangle;\ T_2,\ L',\ Z' \rangle} \quad (\mathit{Read})$$

$$\frac{T_2 \neq \texttt{nil} \quad \neg\exists \langle t, rw_x, c_x, is_x \rangle \in T_2 \quad \langle t', \texttt{Wr}, c', is' \rangle \in T_2 \quad \text{Condition (1) from rule } (\mathit{Exclusive}) \quad Z' = Z[\langle l_1, l_2 \rangle\ \mapsto\ \mathit{NONE}]}{\langle T_1; \langle t, rw_1, c_1, is_1 \rangle;\ T_2;\ \langle t, rw_2, c_2, is_2 \rangle,\ L,\ Z \rangle \overset{\mathit{inst}}{\Longrightarrow} \langle T_1; \langle t, rw_1, c_1, is_1 \rangle;\ T_2,\ L',\ Z' \rangle} \quad (\mathit{None})$$

LOCKSET

$$\frac{}{\langle \langle t, rw, c, is \rangle; T,\ LS \rangle \overset{ls}{\Longrightarrow} \langle T,\ LS \cap \{k | (k, i) \in is\} \rangle} \quad (\mathit{LockSet})$$

ATOMIC PAIRS

$$\frac{\langle T, \mathcal{L} \rangle \overset{ls}{\Longrightarrow} \ldots \overset{ls}{\Longrightarrow} \langle \epsilon, LS \rangle \quad \langle T, L_0, Z_0 \rangle \overset{\mathit{inst}}{\Longrightarrow} \ldots \overset{\mathit{inst}}{\Longrightarrow} \langle \epsilon, L, Z \rangle \quad X = \{\langle l_1, l_2, Z(l_1, l_2) \rangle \mid L(l_1, l_2) \cap LS = \phi\ \wedge Z(l_1, l_2) \neq \top\}}{\langle \langle v, T \rangle; S,\ A \rangle \overset{\mathit{atom}}{\Longrightarrow} \langle S, A[v \mapsto X] \rangle} \quad (\mathit{Pairs})$$

Figure 5. Profiling rules.

vening remote read operations. Rule *(LockSet)* specifies the computation of the variable lock set. Rule *(Pairs)* generates the atomic pairs for a variable. Specifically, a profiled atomic pair is admitted only if the pair and all other accesses of the same variable are not consistently protected by a program lock; these pairs represent regions that are potential targets for subsequent lock instrumentation.

7. Lock Placement

We now discuss how appropriate synchronization can be inserted into a program to respect profiled pairwise atomicity constraints. Recall that atomic pairs need not be lexically-scoped. To avoid limiting concurrency, we need to release locks that protect atomic pairs at the earliest possible point, taking into consideration their non-lexical organization. Moreover, atomic pairs may overlap: two atomic pairs may span the same code region.

Any synchronization injection protocol must be cognizant of these issues. In this section, we consider how to inject appropriate synchronization for a single shared variable; in Section 8, we consider extensions that resolve conflicts between inserted locks for multiple variables or between an inserted lock and a program lock.

The profiling semantics identifies an atomic pair as a pair of local accesses to the same variable with no intervening remote access. To ensure safety, we over-approximate profiled information; specifically, *we consider all paths between two accesses that comprise an atomic pair as exhibiting the same atomicity characteristics with respect to the variable that defines the pair.* Our treatment is an over-approximation because the profile is necessarily incomplete, i.e., not all paths between the two accesses are guaranteed to be exercised by the profiling input suite. On the other hand, the approximation is safe and adding additional synchronization only affects the degree of concurrency realized.

We define an intraprocedural flow analysis to decide where additional synchronization must be inserted. For each control flow node, represented by a label, we compute the set of locks that ought to be held upon entry to that node and then decide the required synchronization instrumentation accordingly. If no profile is available for a pair of accesses, we assume there is no atomicity that must be enforced between that pair.

The rules defining our analysis, presented in Fig. 6, are divided into two sets. The first set computes the locks that must be held at a program point. The second set derives the instrumentation that must be injected according to derived lock information. The computation is based on the atomic pairs for a variable. The relevant definitions are presented in the beginning. Function `lc` is a mapping from a label to a set that may include *NONE* and triples $\langle l_i, l_j, \alpha \rangle$, referred to as the *lock constraint set*. The triple signifies that an α lock should be held, during the execution from l_i to l_j, where α is either a read lock or an exclusive lock. Having *NONE* in the constraint set means that it is possible no lock is held at this point (along some path). The remaining definitions are for auxiliary functions used in the rules.

The rules are given by judgements of the form, $P \Longrightarrow \mathcal{C}$ where P is a predicate and C is either a lock constraint operation or an instrumentation effect performed when P is true. Rule $(Init)$ specifies that the constraint set of a program point l be a singleton holding the *NONE* lock, meaning no lock is needed, as long as l is not on any path of any atomic pair. Note that the *NONE* constraint may be further propagated to other nodes through flow edges. Rule (Gen) specifies that if node l is the head (the first access) of an atomic pair with read or exclusive lock type, the lock constraint set for l must include the atomicity type of all pairs that have l as the head, and must not include *NONE*. Intuitively, the rule requires that we hold some lock starting from l defined by the properties of the atomic pair. Rule $(Join)$ describes the propagation of constraints along flow edges. Specifically, $\texttt{lc}(l)$ is computed from the lock constraint sets of its predecessors. The $notreachable(l_p, l)$ set is computed as the set of constraints of a predecessor l_p such that the corresponding atomic pairs cannot witness l. Recall that a triple $\langle l_i, l_j, \alpha \rangle \in \texttt{lc}(l_p)$ means that a α lock should be held as dictated by an atomic pair from l_i to l_j. Intuitively, if l is not on any path from l_i to l_j, the constraint should be invalid and excluded from the constraint set of l. The $conflict(l_p, l)$ set is computed as the set of constraints of l_p that conflict with the program's control-flow and hence should be excluded from l' lock constraint set. Given a constraint $\langle l_i, l_j, \alpha \rangle$, l_i and l_j must be two consecutive accesses along an execution path. Hence, if l entails paths along which a different access l_k to the variable must be encountered before l_j, the atomicity of the variable under consideration must have been profiled by the pair l_i and l_k instead of l_i and l_j. Hence, all constraints regarding l_i and l_j should be excluded from $\texttt{lc}(l)$. Finally, if l has multiple predecessors, the entailment operation in the consequent effectively defines a union of their lock constraint sets.

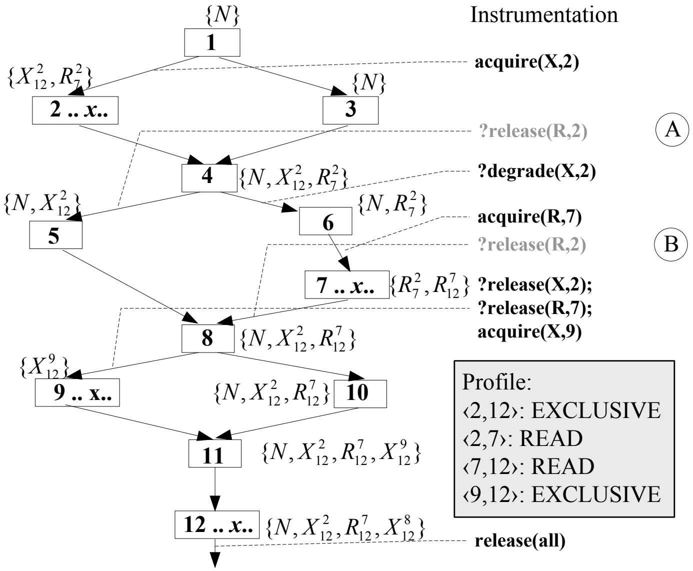


Figure 7. Example for lock placement. Nodes are annotated with their `lc`() set. Symbol X^2_{12} is a shorthand for $\langle 2, 12, \textit{EXCLUSIVE} \rangle$. Symbols R and N denote *READ*(x) and *NONE*(x), respectively. Instrumentations on edges are shown on the right. The shaded ones are optimized away. The question mark before a release primitive is to test if the lock is held. The **degrade()** primitive degrades a lock (from exclusive to read).

Example. Consider the example in Fig. 7. The CFG is presented on the left. Variable x is accessed at nodes 2, 7, 9 and 12. Each node is annotated with its lock constraints. According to rule $(init)$, $\texttt{lc}(1) = \texttt{lc}(3) = \{N\}$ (for $NONE$). Node 2 has the constraint set $\{X^2_{12}, R^2_7\}$ according to rule (Gen). Note that there are two atomic pairs starting with 2 in the profile; the notation X^2_{12} is an abbreviation for the lock constraint $\langle 2, 12, EXCLUSIVE \rangle$. $\texttt{lc}(4)$ is the union of $\texttt{lc}(2)$ and $\texttt{lc}(3)$, dictated by rule $(Join)$. For node 5, $noreachable = \{R^2_7\}$ as 5 is not on any path from 2 to 7, hence $\texttt{lc}(5) = \texttt{lc}(4) - \{R^2_7\}$. For node 6, $conflict = \{X^2_{12}\}$ because control-flow through node 6 implies that an access to x at node 7 will be encountered before execution reaches node 12. Hence the constraint X^2_{12} must not have been observed along the path following 6 and should be excluded. Computation of the remaining constraints follows similarly. □

The second set of rules derives appropriate instrumentation from the lock constraints. The instrumentation is mainly lock acquisition and release. We allow a lock to be acquired

DEFINITIONS

$\mathtt{lc} \in \mathit{LockConstraint} ::= \mathit{Label} \longrightarrow \mathcal{P}((\mathit{Label} \times \mathit{Label} \times \mathit{AtomicType}) \bigcup \{\mathit{NONE}\})$

$\mathtt{edge}(l, l') : \ l \to l'$ is a flow edge

$\mathtt{atomhead}(l) = \exists l', \ \top > Z(l, l') > \mathit{NONE}$

$\mathtt{onpath}(l, l_i, l_j)$: l is on at least one path from l_i to l_j.

$\mathtt{access}(l)$: the variable under consideration is accessed at l.

$\mathtt{mustonpath}(l, l_i, l_j)$: l must be on all paths from l_i to l_j.

$\mathtt{maxlock}(S) = \ \alpha$ iff $\forall \langle l, l', \alpha' \rangle \in S, \ \alpha \geq \alpha'$

$\mathtt{notreachable}(l_p, l) = \{t \mid (t = \langle l_i, l_j, \alpha \rangle) \in \mathtt{lc}(l_p) \wedge \neg\mathtt{onpath}(l, l_i, l_j)\}$

$\mathtt{conflict}(l_p, l) = \{t \mid (t = \langle l_i, l_j, \alpha \rangle) \in \mathtt{lc}(l_p) \ \wedge \exists l_k \neq l_i, l_j \ \ \mathtt{access}(l_k) \wedge \mathtt{mustonpath}(l_k, l, l_j)\}$

LOCK CONSTRAINTS

$$\begin{array}{rcll}
\forall \top > Z(l_i, l_j) > \mathit{NONE}, \ \neg\mathtt{onpath}(l, l_i, l_j) & \Longrightarrow & \mathtt{lc}(l) = \{\mathit{NONE}\} & (\mathit{Init}) \\
\mathtt{atomhead}(l) & \Longrightarrow & \mathtt{lc}(l) \supseteq \{\langle l, l_i, Z(l, l_i) \rangle \mid \top > Z(l, l_i) > \mathit{NONE}\} \ \wedge & \\
 & & \mathtt{lc}(l) \not\supseteq \{\mathit{NONE}\} & (\mathit{Gen}) \\
\mathtt{edge}(l_p, l) & \Longrightarrow & \mathtt{lc}(l) \supseteq \mathtt{lc}(l_p) - \mathit{notreachable}(l_p, l) - \mathit{conflict}(l_p, l) & (\mathit{Join})
\end{array}$$

INSTRUMENTATION

$$\begin{array}{rcll}
\mathtt{atomhead}(l) \ \wedge \ \alpha = \mathtt{maxlock}(\mathtt{lc}(l)) & \Longrightarrow & \text{instrument before } l \text{ with ``}\mathit{acquire}(\alpha(x), l)\text{''} & (\mathit{INAcquire}) \\
\mathtt{edge}(l_p, l) \ \wedge \ \Delta = \mathtt{lc}(l_p) - \mathtt{lc}(l) & \Longrightarrow & \text{for each } \langle l_i, l_j, \alpha \rangle \in \Delta : & (\mathit{INRelease}) \\
 & & \quad \text{if } (\exists \langle l_i, l_k, \alpha' \rangle \in \mathtt{lc}(l), \ \alpha > \alpha') & \\
 & & \quad\quad \text{instrument } l_p \to l \text{ with ``}?\mathit{degrade}(\alpha(x), l_i)\text{''} & \\
 & & \quad \text{else} \ \ \text{instrument } l_p \to l \text{ with ``}?\mathit{release}(\alpha(x), l_i)\text{''} &
\end{array}$$

Figure 6. Lock Placement. The analysis is parameterized on the variable x being considered.

multiple times but released only once. Both read or write capabilities may be acquired for a lock. Rule $(\mathit{INAcquire})$ specifies lock acquisition. Locks are only acquired at a node l that is the start of at least one atomic pair. If there are multiple constraints associated with l, the lock with the maximum strength is acquired. Intuitively, if one constraint demands a read lock and another one demands a write lock, a write lock is acquired in case the path inducing the stronger constraint is taken. The lock state is enhanced with label l to indicate the acquisition point of the current lock. Rule $(\mathit{INRelease})$ specifies the conditions under which an injected lock may be released. Along a control flow edge from predecessor l_p to node l, the difference between constraint sets at l_p and l defines the locks that need to be released at l. Depending upon whether there is a weaker lock retained by $\mathtt{lc}(l)$, the instrumentation mechanism degrades the lock or releases it. Note that such operations are always guarded by a comparison to test the origin of the lock being released/degraded.

Example (continued). Continuing with the example in Fig. 7, observe that edge $1 \to 2$ is instrumented with the acquisition of an exclusive lock according to rule $(\mathit{INAcquire})$. Edge $4 \ \to \ 6$ is instrumented with a degradation operation that weakens the exclusive lock to a read lock, since $\Delta = \mathtt{lc}(4) - \mathtt{lc}(6) = \{X_{12}^2\}$. Intuitively, if the execution comes from path $2 \ \to \ 4 \ \to \ 6$, the exclusive lock is not needed under execution $4 \to 6$ even though the read lock is still required.

If the path $2 \to 4 \to 6 \to 7$ is taken, an exclusive lock for x is acquired at node 2; the lock is degraded to a read lock prior to execution of node 6. The (re)-acquistion of the read lock prior to executing node 7 trivially succeeds since the lock is already held; the instrumentation that injects the read lock here is required because the control-flow path that reaches node 6 could have been $1 \to 3 \to 4 \to 6$. Along the path from 2 to 12 passing 7, our instrumentation aggregates the two atomic pairs $\langle 2, 7 \rangle$ and $\langle 7, 12 \rangle$ to a larger read atomic region, preventing any intervening remote write along the path. □

There are also rules that insert locks to protect single accesses that are not in any atomic pairs, in order to provide consistent protection. The rules are elided here.

Removing redundant instrumentation. We can optimize the instrumentation rules shown in Fig. 6. The basic idea is that the release of a lock is redundant if we can be sure the lock cannot be held at the point where the release was injected. Assume the release is caused by a constraint $c = \langle l_i, l_j, \alpha \rangle$ in Δ along edge $l_p \to l$.

- If there is a constraint in $\mathtt{lc}(l)$ denoting a pair starting at the same point l_i and the lock type is stronger than α. The stronger lock must have been acquired at l_i but not α. The release is redundant. This optimization allows us to remove release Ⓐ in Fig. 7.
- Suppose there is a constraint in $\mathtt{lc}(l)$ that starts at l_j, the end of c, the release is not necessary as a new lock must have been acquired at l_j. The optimization allows us to remove release Ⓑ; thus it has the effect of aggregating two atomic pairs to a larger atomic region.

Similarly, acquisitions can be optimized. The idea is that an acquisition at l is redundant if we are sure the same type of lock must be held at l. We omit the details here.

8. Deadlock Resolution

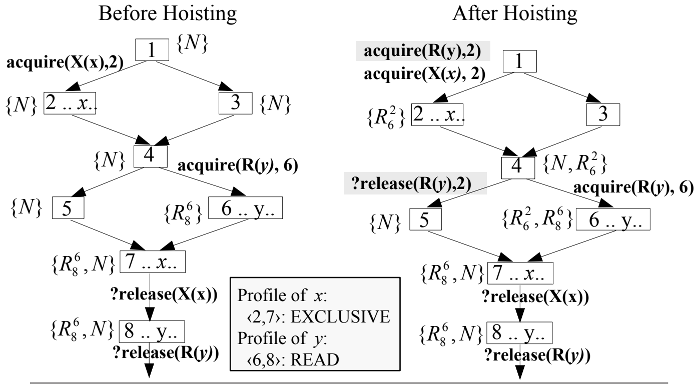


Figure 8. Example for lock hoisting. The goal is to reverse the lock order of variables x and y. In (a), each node is annotated with the `lc` set regarding variable y. In (b), modified `lc` sets are presented, which entail the highlighted instrumentation.

The lock placement algorithm presented in Section 7 does not consider undesirable interference among locks. In particular, deadlocks may form between the locks inserted by our algorithm for different variables or between inserted locks and locks already present in the program. In this section, we explain how we identify and resolve such deadlocks. We have to guarantee deadlock-free for all possible inputs even though the profile is acquired from only a small subset. Hence, we have to resolve all possible deadlocks at compile time. We first construct a static lock-order graph that identifies lock dependences. Strongly connected components in the graph indicate potential deadlocks at runtime. We then break cycles by manipulating where lock insertion takes place. To minimize impact on program structure, we do not relocate program locks.

Constructing Lock Order Graph. In the lock order graph, a node represents a lock, which could be an inserted lock or a program lock. An edge is introduced from lock k_1 to k_2 if, at some program point, k_2 is acquired when k_1 is held. The locks held at a program point can be conservatively computed through a standard dataflow analysis. The analysis is analogous to a reaching definition analysis with lock acquisitions considered as *GEN()* and the corresponding releases as *KILL()*. The computation rule applies to both inserted locks and program locks. The analysis is interprocedural and summary based. The summary of a function is the set of locks that may be acquired in the function. Suppose a function *foo()* is invoked at a program point l. Let the set of locks that may be held at l be S. Edges are added to the lock order graph from each lock in S to each lock in the summary of *foo()*. Cycles in the graph indicate potential runtime deadlocks.

Recall that our design distinguishes read and exclusive locks for the same variable. In a lock order graph, the read and exclusive locks of the same variable, however, are represented as the same node. Intuitively, the mutually exclusive nature between a read and an exclusive lock of the same variable could lead to a deadlock. For example, assume thread t_1 holds a program lock k_1 and acquires a read lock on variable x, denoted as *READ*(x), and t_2 holds an exclusive lock of x, denoted as *EXCLUSIVE*(x) and acquires k_1. A deadlock is thus formed.

On the other hand, if a lock dependence edge is exclusively due to a *READ* lock on a variable x acquired while a *READ* lock on a different variable y is held, the edge can be safely removed.

For example, assume $k \rightarrow$ *READ*$(x) \rightarrow$ *READ*$(y) \rightarrow k$ is a cycle in the graph. It may correspond to the case that thread t_1 holds k and acquires *READ*(x), t_2 holds *READ*(x) and acquires *READ*(y), and t_3 holds *READ*(y) and acquires k. The non-exclusive nature of the read locks allows t_1 and t_2 to proceed and eventually t_3 to proceed. Thus, the edge can be safely removed.

Lock Hoisting. Strongly connected components (SCC) in the lock order graph indicate potential deadlocks. We need to resolve these SCCs statically in order to avoid runtime deadlocks. There are two possible solutions: (1) merge the locks in a SCC to a single lock, or (2) reverse lock order edges. Because lock merging is likely to be too conservative, we propose to reverse lock order edges. Reversing lock edges is tantamount to hoisting lock acquistions. For example, if reversing an edge from k_1 to k_2 can break a cycle, this implies we could hoist the acquisition of k_2 to before the acquisition of k_1. Since we do not change the original program semantics, only inserted locks are subject to hoisting.

The challenge to reversing lock edges lies in the fact that inserted locks are not lexically scoped. To reverse the lock order from k_1 to k_2, hoisting the acquisition of k_2 to right before the acquisition of k_1, often unnecessarily limits concurrency, and is even problematic in many cases. In particular, the inserted lock k_2 is supposed to protect atomicity along a subset of paths P between a pair of accesses as dictated by the profile. Ideally, hoisting k_2 to before k_1 should be able to *protect a superset of* P*, (call it* P'*), such that originally along each path in* P'*,* k_1 *is acquired and then* k_2 *is acquired, but the lock order is reversed after hoisting*. However, since the placement of k_1 is not lexical, simply moving the acquisition of k_2 to right before that of k_1 may result in protecting a superset of P' that, while safe, may unnecessarily limit concurrency, or a subset of P, which minimizes the impact on concurrency, but may violate the observations defined by the profiled execution.

Consider the example in Fig. 8 (a). It shows the lock placement of the given profile following the rules in Fig. 6. The lock order is *EXCLUSIVE*$(x) \rightarrow$ *READ*(y). Assume we want to reverse this order to resolve potential deadlocks. Moving the acquisition at node 6 to node 2 is both inefficient and problematic. If the execution follows the path 2, 4, 5, 7 and 8, lock *READ*(y) is unnecessarily held as the profile only demands locking the path 6, 7 and 8. If the execution follows the path 3, 4, 6, 7 and 8, the lock is not held while it is needed.

Our solution is as follows. To ensure that we always protect a superset of P, we do not relocate the acquisition of k_2. Instead, we add an extra acquisition of k_2 before the acquisition of k_1. We call it the *hoisted* acquisition to distinguish it from the *original* acquisition. The intuition is that we only need to reverse the lock order along paths that k_1 is first held and then k_2 is acquired. If a lock is already held as a result of a hoisted acquisition, reacquiring the lock on the original path is a benign action.

Note that the hoisted k_2 lock should be released if the control flow takes a path along which the original k_2 acquisition is not encountered. In order to systematically handle hoisted lock releases, we leverage our instrumentation rules in Fig. 6. In particular, we add lock k_2 to the `lc` set for all nodes that are on a path from the acquisition of k_1 and the original acquisition of k_2, meaning that we add a new constraint such that k_2 must be held along all paths from the acquisition of k_1 to the original acquisition of k_2. Instumentations can be derived from the `lc` sets following the instrumentation rules in Fig. 6. Note that we cannot simply add an atomic pair between the acquisition of k_1 and the acquisition of k_2 and let the rules to infer the lock constraint sets. The reason is that the lock constraint rules consider conflicts between atomic pairs (i.e., the $conflict$ set in rule ($Join$)) so that k_2 may not be held along some paths between k_1 and k_2 acquisitions (as dictated by a conflict) even though we introduce an atomic pair. As a result, the original acquisition of k_2 may not be a no-op and thus the lock order $k_1 \rightarrow k_2$ may not be reversed.

Consider the example in Fig. 8 (b). A read lock from 2 to 6, denoted as R_6^2, is inserted to the lock constraint sets for the path between the two original acquisitions, including nodes 2, 4 and 6. The hoisting and the new constraint set lead to the highlighted instrumentations along edge $1 \rightarrow 2$ and $4 \rightarrow 5$.

9. Condition Variables and Shared Heap Access

```
                          // thread 2
// thread 1               acquire(our_lock)
acquire(our_lock, ...)    release(our_lock)
...                       ... //non-trivial
cond_wait(C);             ... //program path
                          cond_signal(C);
```

Figure 9. A deadlock involving a condition variable and a lock added by our technique. Thread 1 blocks at the condition wait and thread 2 blocks at `our_lock`.

Condition variables: Our technique has so far not taken condition variables into consideration. Statically resolving deadlocks involving condition variables is challenging in general. Fig. 9 presents an abstraction of the cases we have encountered. Thread 1 blocks at the condition wait and thread 2 blocks at the added lock acquisition injected by our analysis. In practice, such deadlocks may involve other program locks and added locks. Statically resolving the deadlock is hard because the condition signal in thread 2 is not even in the region protected by `our_lock` and hence a lock dependence graph can not be easily constructed. Assuming a lock dependence exists between a lock acquisition and any reachable condition signal would be too conservative in practice. Because of these complexities, we resort to a runtime solution. The basic idea is to disable our locks when program execution blocks on a condition variable, until the condition is satisfied. In particular, we release all our locks held by a thread when it blocks on a condition variable. Due to transitive lock dependences, we also release our locks in other threads that transitively block on the condition variable. In the presence of condition variables, we refine our definition of pairwise atomicity appropriately: a pairwise atomic region is a code region (not necessarily lexical) bracketed by two accesses to the same variable, without any intervening access to that variable by another thread, *provided the region has no lock dependence with any thread currently blocked on a condition variable*.

Heap variables: Heap-allocated shared data also poses challenges for our technique. An ideal solution is to provide per-object locking so that maximal concurrency can be achieved. However, ensuring deadlock-freedom using per-object locking is difficult since static deadlock resolution often requires object lock acquisitions to be hoisted; oftentimes, we cannot guarantee that the object has even been allocated at this hoist point. Furthermore, static deadlock resolution has difficulty disambiguating different instances of the same heap type that will be protected by different locks at runtime. A conservative solution results in solving bogus deadlocks. We therefore currently use a conservative type-based locking strategy, which is trivially compatible with our deadlock analysis described earlier. Note, however, that even though we obtain locks on types, our profiler still tracks atomicity properties independently for each heap instance of a type. In other words, the type based abstraction does not affect the quality of the profiled atomicity invariants. Due to the large amount of shared heap types and some of them coming from libraries, we expect programmers to annotate the heap types of interest. In this paper, we focus on those for which bugs have been reported.

10. Safety

There are two fundamental safety properties guaranteed by our technique. First, the approach is *execution safe*: the injected instrumentation does not introduce additional behavior beyond what could be exhibited by the original program. Second, the approach is *atomicity safe* and respects the atomicity properties of the profiled execution: our instrumentation guarantees that the transformed program will not permit non-atomic behavior within atomic regions found in the profile.

We formalize these notions below. Let the original program be P and the instrumented program be P'.

THEOREM 1 (Execution Safety). *For any input I, executing P' on I produces the same result as executing P on I.*

Proof Sketch. Let an execution E be a sequence of state transitions $S^0 \Rightarrow S^1 \Rightarrow \ldots \Rightarrow S^n$ where a state S is a tuple $\langle s_0, s_1, ..., s_n \rangle$, representing the local states for the n threads. Consider an execution E' of P' instantiated from an input I. A simple inductive argument on the number of instrumented actions suffices to show that given an execution E' of P', an equivalent execution E can be derived for P on I.

The theorem holds trivially in the absence of any instrumentation. The inductive hypothesis assumes the property holds for $k-1$ steps. There are two interesting cases. For the first case, assume an inserted lock acquisition is encountered at the kth step in thread t_i in E' and the acquired lock is already held by t_i; in this case, the lock acquisition is a no-op, and execution can proceed. For the second case, assume the acquisition fails. This results in a context switch in E'. Assume the scheduler is able to pick up a different thread t_j to continue. Let $x < k$ be the last step performed by t_j in E', implying a context switch at step $x+1$. Suppose the context switch was not caused by blocking. By the induction hypothesis, t_j was not blocked at E, and it is safe to switch to t_j at step k in E. If the context switch at step $x+1$ in E' was caused by blocking, the fact that we can now switch back to t_j implies the lock on which t_j was previously blocked is now available due to some previous release at step y ($x < y < k$). By the induction hypothesis, this release was matched in E.

Now, assume the acquisition fails and E' can not pick up a different thread t_j to continue. This means that E' denotes a deadlocked schedule. In other words, there must be a cycle in the dynamic lock order graph on instrumented locks, which contradicts to the fact that we already have statically resolved all deadlocks. Note that a deadlock in E' can not involve conditional variables since a thread that blocks on a condition variable or a program lock with a lock dependence on such a condition variable releases all our locks held by it.

Inserted lock releases in E' can be considered as no-ops in E. The proof is similar and elided. □

THEOREM 2 (Atomicity Safety). *A pairwise atomic region identified in P will be executed atomically by any execution of P'.*

The proof follows trivially from the validity of the following two lemmas. Recall that the refinement of the definition of atomic pairs given in Sec. 6 excludes pairs where there exists possible lock dependences with threads blocked on a condition variable along some path between the encapsulating accesses; the lemmas are therefore defined without taking such accesses into account.

LEMMA 1. *The lock placement algorithm in Fig. 6 respects the atomicity constraints defined by the profile.*

Proof Sketch. We prove the lemma through contradiction. Assume $\langle l_1, l_2, \mathit{EXCLUSIVE} \rangle$ is an atomic pair in the profile, and that in one execution of P', a path from l_1 to l_2 is exercised in thread t_i. Variable x is accessed in both l_1 and l_2 and not accessed in other points along the path. Assume the path is interleaved with a write access to x at l' in a different thread t_j.

According to the rules in Fig. 6, an exclusive lock is acquired before l_1 and an exclusive lock must be held before l' in t_j. Since l' interleaved with the path from l_1 to l_2, the lock acquired in t_i must have been released along some edge $l_p \rightarrow l_q$ along the path. According to rule ($IN - Release$), this implies constraint $c = \langle l_1, l_2, \mathit{EXCLUSIVE} \rangle$ is in $\texttt{lc}(l_p)$ but not in $\texttt{lc}(l_q)$. According to rule ($Join$), c must be in the $notreachable$ set or the $conflict$ set for the computation of $\texttt{lc}(l_q)$. Since l_q is along the path from l_1 to l_2, c must not be in $notreachable$. Hence, it must be in $conflict$. There must be another access to x along all paths from l_q to l_2, contradicting the assumption.

Similar reasoning applies for other access configurations, e.g., the access at l' is a read and the pair is $\langle l_1, l_2, \mathit{READ} \rangle$. □

LEMMA 2. *The deadlock resolution algorithm in Section 8 respects the atomicity constraints in the profile.*

According to the deadlock resolution algorithm, deadlocks are resolved by hoisting locks. A lock is hoisted by adding entries to `lc` sets, leading to extra acquisitions of the lock. After hoisting, the paths on which the lock is held are a superset of the original paths. Hence, the atomicity constraints must be respected.

The atomicity safety property can be derived from these two lemmas. While our technique is safe, the rules in Fig. 6 ensure optimality as an inserted lock is always released at the earliest possible (atomicity-safe) program point.

11. Evaluation

Our system is implemented within LLVM. The profiler is implemented as an instrumentation pass that runs independently of the rest of the system. The lock placement and deadlock resolution are implemented as two passes that take the profile as input and insert deadlock free synchronizations. The overall implementation has 8K LOC in C. The experiments are conducted on a Intel Xeon CPU 2x4 core 1.84GHz machine with 4GB RAM. We create 8 threads in all executions.

program	LOC	func	profile input
aget-0.4	960	20	get a 100k file
pbzip-2.094	2041	121	compress 100k file with 1k block size
mozilla nszip-1.8	2935	45	extract 100 files
apache-2.2.8	42k	1.7k	10k random reqs
spidermonkey	51k	1098	compute MD5 hash
mysql-4.0.12	122k	5.7k	regression test suite and sysbench-oltp benchmark

Table 2. Programs and profile inputs.

11.1 Effectiveness

We evaluate the effectiveness of bug suppression by applying our technique on real bugs found in production-quality software. We have already previously discussed the applications we have studied (shown in Table 1); Table 2 presents the program characteristics and inputs used for collecting profiles for these programs. Column [`func`] presents the functions under our analysis. Column [`profile input`] presents the workload used for collecting atomicity profile. These programs are well-studied standard benchmarks for studying concurrency bugs [20]. In particular, `spidermonkey` is the JavaScript engine for Mozilla. `Mozilla-nszip` is the decompression component of Mozilla.

The real patches for these bugs are also available from the bug reports or their CVS repositories. We compared the patches with the synchronizations inserted by our technique manually and confirm that our technique prevents these bugs. We also ran the instrumented programs on the failure inducing inputs with an implementation of the CHESS [23] algorithm developed in our prior work [33], with the 1-preemption setting. Bugs that were previously exposed no longer occurred.

Note that even though our instrumentation mechanism is based on profiles that extract pairwise atomicity, our technique can also prevent concurrency bugs that are classified as data races or order violations. For example, the `pbzip` bug was considered as an order violation. The bug occurs because a read to a variable x in a thread T_1 must always happen before a re-definition of x in a different thread T_2. But, the program permits violation of such orderings, leading to a failure. Our technique observes atomicity between the read and its preceding local read. As existing program synchronizations preserve that the preceding read happens before the remote write, by locking the two reads, the order violation is suppressed.

Case Study. Next, we present a fragment from `apache`, to further illustrate the benefit of our approach. Bug#44402 shown in Fig. 10 was observed while running the `specweb99` static content workload. The server would crash typically after 10 minutes of starting the test with 500 active threads. [2] The bug took developers one week of investigation spanning more than 1000 lines of discussion before they were able to identify the faulty interleaving. The matter is further complicated by the fact that the crash happens at places far removed from the location where atomicity is violated and a failure may manifest at different places at different times.

```
ap_queue_info_set_idle(pool_t *pool)
{
   node = apr_palloc(pool, sizeof(*node));
   node->pool = pool;
   for (;;) {
      node->next = recycled_pools;
      if (apr_atomic_casptr(&recycled_pools,
L:          node, node->next) == node->next)
         break;
  }
  ...
}
```

Figure 10. Apache bug#44402: A bug that manifests under a rare complicated interleaving with multiple preemptions. The two reads to `node->next` in the conditional test should be executed atomically to prevent a race condition.

In the code shown in Fig. 10, worker threads in the web server, call the function `ap_queue_info_set_idle ()` to deposit a memory pool which is no longer in use in the list of recycled pools. Under normal conditions, the list `recycled_pools` is a linear list. However, if three different threads interleave within this function simultaneously (a rare but possible occurrence), the list can be modified to be circular with just one pool after four different preemptions. We refer the reader to the bug report for details. Consequently, all workers which should otherwise be using distinct memory pools for handling requests end up using the same memory pool, eventually leading to a server crash. Due to the higher number of threads and multiple preemptions required (in addition to the atomicity violation) and the fact that the crash manifests elsewhere in the server (necessitating testing the full server to expose the failure), it is difficult to both reproduce and understand this bug. However, by only looking at correct executions where the two reads to `node->next` at line `L` are atomic, and thus enforcing this property, the bug can be prevented easily. This is, in fact, the patch that was eventually provided by the developers.

11.2 Performance Overhead

Next, we present experimental results that highlight quantitative aspects on the cost of the analysis and implementation on these benchmarks. Columns 2-4 in Table 3 present the profiling results. Column [`shared vars/ unprotected`] presents the shared variables identified, and those that are not consistently protected by at least one lock. Columns 3 and 4 present the read and exclusive atomic pairs identified by the profiler and the average number of (static) statements that lie in between a pair (i.e. the numbers in parentheses). As these programs are mostly event-driven, executing the programs with various events (or event combinations) are equivalent to running the programs multiple times. Hence,

[2] https://issues.apache.org/bugzilla/show_bug.cgi?id=44402

we construct the profile inputs as follows: we combine and duplicate test inputs in a regression suite into a larger input consisting of all events from individual tests. We randomly insert `sleep()` calls at synchronization points to increase our schedule coverage. Observe that the number of shared variables that are not consistently protected is non-trivial. Our claim is that the statements enclosed by atomic pairs with respect to these variables are vulnerable to concurrency bugs. Also observe that the average number of such statements in an atomic pair is not large.

Columns under label [`inserted locks`] in Table 3 present the results of static lock placement. Column [`lock`] presents the number of inserted locks. Columns [`R acq`], [`X acq`], and [`degrade`] denote the numbers of read lock acquisitions, exclusive lock acquisitions and lock degradations, respectively. For each of these columns, we present the numbers before and after deadlock resolution. Columns under label [`program locks`] present the statistics of existing program locks for reference. We make the following observations.

- The number of injected locks is comparable to that of program locks. The number of injected synchronization primitives before deadlock resolution is also comparable to the number of existing synchronizations. There are two reasons for this: one is that our technique induces fine-grained guidance of executions based on profiled data – such guidance is controlled by lock acquires and releases; secondly, path-sensitive locking injects releases at many points, corresponding to different atomicity properties manifest along different paths. But, note that lock releases for the same variable are usually along different paths so that they are not typically encountered multiple times along one path during an execution.
- Deadlock resolution increases the number of injected synchronizations substantially. This is due to the conservative nature of the analysis.
- Comparing the numbers of the unprotected variables and profiled atomic pairs with the numbers before deadlock resolution, we observe that we do not need to insert locks to protect all unprotected variables because some are not marked as atomic. Some atomic pairs are aggregated to larger atomic regions. Some acquisitions are optimized away, which explains why the number of inserted synchronizations is smaller than the number of atomic pairs.

Table 4 presents runtime characteristics. We execute the instrumented programs on realistic workloads. The second column describes the workloads. The third column presents the total number of executed instructions. Column [`speedup`] presents the speedup of running the *original* program on 8 cores over 1 core. This provides a measure of the available concurrency in these workloads. Note that these numbers are collected without any involvement of our system. Column [`P acq`] presents the dynamic count of the number of program lock acquisitions. The next two columns show the ratio of instrumented locks with respect to this number. The last column shows overall runtime overhead. Observe that the overhead is low despite the large number of injected synchronizations, indicating our instrumentation rarely cause severe blocking. This is because we are locking on observed atomic pairs, whose executions are mostly atomic, and consequently likely to not be a source of contention. The overhead of our technique is sometimes related to the available concurrency in the workload. If available concurrency is low, the overhead of our technique is masked further. For example, we use two workloads for `mysql`. The first one is a standard workload called `sysbench`. The second is a reduced version of the `tpcc` workload, another standard workload. The tpcc workload is much more I/O bound. Consequently, the overhead of our technique is much lower. A similar workload configuration is used for `spidermonkey`. In contrast, note that both `pbzip` and `mozilla-nszip` get substantial speedup with very low overhead.

12. Related Work

Concurrent with our work, [18] also attempts to automatically repair concurrency bugs by inserting locks around inferred atomic regions. However, unlike our approach, which attempts to prevent bugs whose existence may not even be known, their technique attempts to repair existing ones. Their technique uses CTrigger [26] to identify atomic pairs related to the observed failure; our technique identifies atomic pairs by mining the pool of passing test cases. As a result, we face the challenge of enforcing a much larger number of atomic pairs, with the corresponding potential benefit of preventing unobserved bugs. To reduce overhead, our locking instrumentation uses both read locks as well as exclusive locks, whereas exclusive locks suffice in their implementation. Furthermore, the two techniques also differ in the way they handle deadlock. We guarantee deadlock freedom through static analysis.

In [35, 7], hardware-based techniques are proposed to ensure dependence integrity. Violations incur rollback and cause a different schedule to be explored. In [10], schedules are memoized with respect to inputs so that if the same input is encountered, the same schedule is reused. LOOM [32] allows user to put in explicit annotations to repair races. Like [10], it provides no safety guarantees, and deadlocks can be introduced.

CoreDet [2] guarantees deterministic outputs by allowing threads to run concurrently when they are not communicating. It tracks ownership and employs a deterministic commit protocol. In [6] authors present a language and a type system that support nondeterministic computation with a deterministic-by-default guarantee where nondeterminism must be explicitly requested via special parallel construct(s). None of these techniques attempt to suppress bugs or guide

program	shared var/ unprotected	atomic pair		program locks			inserted locks				
		R (stmt)	X (stmt)	lock	acq	rel	lock	R acq (before/after)	X acq (b/a)	degrade (b/a)	rel (b/a)
aget	5/ 3	0 (-)	7 (2)	1	2	2	3	0/ 9	4/ 4	0/ 0	4/ 12
pbzip	18/ 17	60 (12)	16 (6)	3	24	26	15	28/ 35	13/ 28	2/ 0	76/ 104
mozilla nszip-1.8	2/ 2	3 (1)	7 (9)	1	1	5	2	0/ 0	4/ 3	0/ 0	9/ 8
apache	15/ 13	11 (5)	28 (10)	9	21	29	13	6/ 13	17/ 20	1/ 1	52/ 65
spidermonkey	16/ 9	13 (37)	26 (11)	9	36	36	8	10/ 7	18/ 39	5/ 2	36/ 108
mysql	66/ 20	91 (61)	181 (7)	36	222	300	15	71/ 39	99/ 431	7/ 5	249/ 959

Table 3. Profiling and lock placement results.

program	workload	instr. (10^9)	speedup	P acq	R acq/ P acq	X acq/ P acq	overhead
aget	get a 648M file	0.045	2.58	147.4k	0	1.0	0%
pbzip	compress 1.8G file	798.9	5.33	14.9k	2.0	1.28	1.4%
mozilla nszip-1.8	extract 400k files	179.7	3.76	399k	0	2	0.0%
apache	2 million random reqs	158.5	4.2	14.2m	1.48	2.04	13.3%
spidermonkey	constraint solver	308.22	1.06	422m	0.0	0.0	2.1%
	null script	24.1	4.6	18m	0.11	0.44	17.3%
mysql	sysbench-oltp with 10k rows	258.5	4.34	7.76m	1.29	0.57	13.9%
	reduced tpcc	185.8	1.74	4.22m	1.48	0.61	2.0%

Table 4. Runtime overhead (8 threads). In the `P acq` column, units `m` and `k` mean million and thousand, resp.

schedulers to enforce correct executions. Isolator [28] guarantees isolation in well-behaved threads of a program that obey a locking discipline even in the presence of ill-behaved threads that disobey the locking discipline.

In [21, 9], locking is statically inferred from atomic region annotations so that atomicity can be guaranteed. Atomic regions are lexically scoped. Locksmith [27] statically associates locks with object abstractions and at run time, abstract locks are safely instantiated to different concrete locks under various contexts. Atomic set serializability [30] introduces the notion of data centric synchronization in which users annotate a set of data fields that should have similar consisteny properties. Synchronizations are automatically inferred to ensure serializability of operations on atomic sets in a method body. In [11], the authors describe a technique for infering locks from atomic section annotations. It is path-sensitive and uses read/write, multi-granular locks and also attempts to release locks as early as possible. Similarly, in [14], locks are allocated automatically in a multi-threaded program annotated with atomic sections. Their algorithm works in the presence of pointers and procedures, and sets up the lock allocation problem as a 0-1 ILP which minimizes the conflict cost between atomic sections while simultaneously minimizing the number of locks. Unlike these approaches which rely on static analysis and programming abstractions, our approach infers properties from correct profiled executions to inject suitable locking instrumentation. Our work could benefit from lock coarsening techniques such as [12] to further improve performance and eliminate potential deadlocks.

There has been extensive investigation on detection of various kinds of concurrency bugs, such as data races [29, 25, 3, 5, 24], atomicity violations [31, 16, 15, 34], order violations [20], and deadlocks [17, 1]. There has also been recent progress in devising techniques that can generate deterministic failure-inducing schedules [23, 26, 4]. These techniques systematically explore a bounded space of schedules, and are complementary to our work.

13. Conclusion

This paper proposes a novel technique to suppress Heisenbugs by inferring fine-grained atomicity properties from correct profiled executions. We describe a deadlock-free path-sensitive locking scheme to force program execution to adhere to these properties. Experimental evaluation on real-world workloads demonstrates that it can be used to successfully suppress subtle atomicity and order violation concurrency bugs with low overhead.

References

[1] R. Agarwal, L. Wang, and S. D. Stoller. Detecting Potential Deadlocks with Static Analysis and Runtime Monitoring. In *PADTAD'05*.

[2] T. Bergan, O. Anderson, J. Devietti, L. Ceze, D. Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *ASPLOS'10*.

[3] E. Bodden and K. Havelund. Racer: Effective Race Detection Using AspectJ. In *ISSTA'08*.

[4] S. Burckhardt, P. Kothari, M. Musuvathi and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS'10*.

[5] M. D. Bond, K. E. Coons and K. S. McKinley. Pacer: Proportional Detection of Data Races. In *PLDI'10*.

[6] R. Jr. Bocchino, S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc and T. Shpeisman, Safe nondeterminism in a deterministic-by-default parallel language, In *POPL'11*.

[7] L. Chew and D. Lie, Kivati: fast detection and prevention of atomicity violations, In *EuroSys'10*.

[8] F. Chen, T. F. Serbanuta, and G. Rosu. JPredictor: A Predictive Runtime Analysis Tool for Java. In *ICSE'08*.

[9] S. Cherem, T. M. Chilimbi, S. Gulwani. Inferring Locks for Atomic Sections. In *PLDI'08*.

[10] H. Cui, J. Wu, C. Tsai and J. Yang. Stable Deterministic Multithreading through Schedule Memoization. In *OSDI'10*.

[11] D. Cunningham, K. Gudka and S. Eisenbach, Keep off the grass: locking the right path for atomicity, In *CC'08/ETAPS'08*.

[12] P. Diniz and M. Rinard, Lock Coarsening: Eliminating Lock Overhead in Automatically Parallelized Object-Based Programs, In Journal of Parallel and Distributed Computing, 49(1):218–244, 1996,

[13] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *VEE'08*.

[14] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar. Lock allocation, In *POPL'07*.

[15] C. Flanagan and S. N Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *POPL'04*.

[16] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic Detection of Atomic-Set-Serializability Violations. In *ICSE'08*.

[17] P. Joshi, M. Naik, K. Sen, and D. Gay. An Effective Dynamic Analysis for Detecting Generalized Deadlocks. In *FSE'10*.

[18] Z. Jin, L. Song, W. Zhang, S. Lu, and S. B. Liblit, Automated Atomicity-Violation Fixing. In *PLDI'11*.

[19] S. Lu and S. Park and C. Hu and X. Ma, and W. Jiang and Z. Li and R. Popa, R. and Y. Zhou. MUVI: Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In SOSP'07.

[20] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: A Comprehensive Study on Real-World Concurrency Bug Characteristics. In *ASPLOS'08*.

[21] B. McCloskey, F. Zhou, D. Gay and E. Brewer. Autolocker: Synchronization Inference for Atomic Sections. In *POPL'06*.

[22] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay. In *ASPLOS'09*.

[23] M. Musuvathi and S. Qadeer. Fair Stateless Model Checking. In *PLDI'08*.

[24] J. P. Martin, M. Hicks, M. Costa, P. Akritidis, and M. Castro, Dynamically checking ownership policies in concurrent c/c++ programs, In *POPL'10*.

[25] R. O'Callahan and J.-D. Choi. Hybrid Dynamic Data Race Detection. In *PPoPP'03*.

[26] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from their Hiding Places. In *ASPLOS'09*.

[27] P. Pratikakis, J. S. Foster and M. Hicks. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI'06*.

[28] S. Rajamani, G. Ramalingam, V. P. Ranganath, and K. Vaswani, ISOLATOR: dynamically ensuring isolation in comcurrent programs, In *ASPLOS'09*.

[29] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comp. Sys.*, 1997.

[30] M. Vaziri, F. Tip and J. Dolby. Associating Synchronization Constraints with Data in an Object-Oriented Language. In *POPL'06*.

[31] L. Wang and S. D. Stoller. Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs. In *PPoPP'06*.

[32] J. Wu, H. Cui and J. Yang. Bypassing Races in Live Applications with Execution Filters. In *OSDI'10*.

[33] D. Weeratunge, X. Zhang and S. Jagannathan, Analyzing multicore dumps to facilitate concurrency bug reproduction, In *ASPLOS'10*.

[34] M. Xu, R. Bodik, and M. D. Hill. A Serializability Violation Detector for Shared-Memory Server Programs. In *PLDI'05*.

[35] J. Yu and S. Narayanasamy. A Case for an Interleaving Constrained Shared-Memory Multi-Processor. In *ISCA'09*.