

# Speculative Lock Insertion

Paper number 128, 14 pages

## Abstract

The recent move to multicore hardware means that synchronization bugs (e.g. race conditions, atomicity violations) will become increasingly common. In this paper, we introduce Speculative Lock Insertion (SLI), a new technique which can automatically fix some of these kinds of bugs in program binaries. SLI starts with a reproduction of the bug, uses a combination of static and dynamic analysis to characterize it, and finally generates and applies a binary patch which fixes the bug. We demonstrate the technique's effectiveness using both real and artificial bugs, and discuss some of the implementation challenges and limitations.

## 1. Introduction

The increasing availability of multi-core and multi-processor systems is driving a trend towards software with a greater degree of parallelism, but, while potentially paying dividends in improved responsiveness, throughput, and power consumption, multi-threaded programming has an unfortunate tendency to lead to very subtle bugs. Even worse, it is often difficult to trigger these bugs reliably, which means that they are less likely to be discovered by testing and harder to fix once they have been discovered. A number of techniques have been proposed for reducing the likelihood of such errors, including transactional memory[23] and automatic parallelization[2], but these cannot be transparently applied to the large body of existing concurrent software. There is therefore a need for techniques which can assist in fixing bugs in programs written using the currently widely-used shared memory model of concurrency. In this paper, we introduce SLI, or Speculative Lock Insertion, as one potential approach to this problem. SLI automatically fixes observed synchronization

bugs, given only the program binary and a reproduction of the bug, generating a modified binary whose behavior is identical to that of the original except that it no longer suffers from the bug. Furthermore, the fixes will usually have very low performance overhead, and the process of generating the fix itself takes only a moderate amount of time (ranging from seconds in simple cases to a few minutes in more complicated ones). SLI does not depend on programmer annotations or semantic knowledge of the program's intended behavior; it does, however, depend on having observed the bug, and on having captured it in a deterministic replay system (DRS).

SLI does not attempt to fix every possible synchronization bug. Instead, we consider bugs caused by one thread reading an in-memory structure while some other thread is in the process of updating it causing the reading thread to crash. There are several key challenges here:

- Extracting a higher level abstract “read operation” from the crashing thread. This consists of the memory loads which are relevant to the observed crash, plus the minimal amount of local computation required to tie them together (address computations, for instance).
- Identifying memory regions which constitute a structure. This might, for instance, be the first three entries in a linked list, or all of the nodes on a particular path through a DAG.
- Determining which stores issued by other threads might have raced with the read operation in a dangerous way.
- Embodying the information obtained as a fix which can be applied directly to the program binary.

The first challenge is key; given a characterization of the read operation and a DRS log, the other three are straightforward. We represent these abstract operations using state machines derived using a combination of static analysis, applied to the program text, and dynamic analysis, applied to the DRS log. These state machines characterize and approximate the parts of the crashing thread's behavior which are most relevant to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOSP 2011 date, City.

Copyright © ACM [to be supplied]. . . \$10.00

the observed crash, allowing SLI to extract the useful information from the vast sea of superfluous detail provided by the DRS, and hence to proceed to a useful fix.

## 2. Capturing the bug

Before SLI starts, the bug to be fixed must first be captured using a deterministic replay system. This work does not attempt to advance the state of the art in DRSEs, but does depend on them in order to be feasible, and so we discuss them briefly here. The only requirement we place on the choice of DRS is that it must allow us to replay the relevant fragment of execution as many times as necessary and produce the same sequence of memory accesses each time. This captured execution does not need to be precisely the same as the original crashing execution (although excessive imprecision here could lead to SLI fixing the wrong bug). The most obvious way of capturing an execution, used in our prototype, is to simply record every single memory access issued by the program, which is effective but has extremely high overhead. This could be reduced by using a more intelligent recording mechanism such as PRES[17] or ODR[1], both of which record just a few critical events and discover the rest only when they are needed during replay. This can reduce overhead to a level where it is sensible to run with recording enabled by default. ESD[25] is an extreme form of this approach, and logs nothing at all but instead attempts to recreate the path to the failure given just the state of the program at the time of the crash. In a slightly different context, an automatic program exerciser such as CHESS[14] could be used to detect unknown bugs, which could then be passed to SLI to be automatically characterized and fixed.

## 3. Building abstract read operations

The first phase of our algorithm is to identify the abstract read operation which the program was performing when it crashed. This is represented by a series of state machines, each corresponding to and approximating a fragment of the program starting at some instruction executed by the crashing thread and ending at the point of the crash. The state machine can be evaluated when one of the program's threads executes the starting instruction and predicts, given the state of memory and the thread's registers, whether the thread would crash if it were executed in isolation starting from that state. In that sense, the state machine captures the part of the crashing thread's behavior which is most relevant to the bug which is to be fixed.

The state machines can be regarded as small programs in a very simple graphical language, and the process of producing them as a kind of compilation. The language has two basic components, states and directed edges. States can be either internal, in which case they

have two outgoing edges and a predicate which controls which is taken in any given evaluation, or terminal, in which case they have no outgoing edges and are labelled with either `crash` or `no-crash`. Edges link states, and are labelled with a list of store operations. The predicates for internal states, and the address and value of store operations, are represented in a simple expression language which has forms for examining registers, such as `rax` or `rsp`, and for examining values loaded from memory. It also has the usual operators from C-like languages, like `+` and `==`, which have the standard semantics, and a few simple functions such as `BadAddr` (which tests whether a pointer can be safely dereferenced) and `CondEq` (which takes an x86 `eflags` condition code word and checks whether the `Z` flag is set, indicating that the values compared were equal).

Figure 2(a) shows the state machine generated for instruction A in Figure 1, and illustrates most of the important points of the scheme. The state machine first issues the store `load(global1@A) -> (rsp)`, which corresponds to instruction B and indicates that the value loaded from `global1` at instruction A is stored in the memory location pointed at by register `rsp`. It then evaluates `load(global1@A) == 0` and goes left if the value is false (corresponding to the branch at instruction D being taken) or right if it is true (corresponding to the branch at D not being taken). In either case, it performs one final test on the contents of memory, with the result determining whether or not the program crashes. The remainder of this section details our algorithm for producing these state machines automatically.

### 3.1 The proximal cause

The first step is to locate the first point in the log at which something has definitely gone wrong, and hence to obtain a direct cause of the crash and to nominate one thread as being directly responsible for it. A naive approach would simply use the point at which the program crashed. In principle, this is always correct, and for some simple bugs, such as `NULL`-dereferences or assertion failures, it works well. However, for more complicated classes of bugs, such as `use-after-frees`, there can be a significant lag between the first definitely bad behavior (such as the use of released memory) and the program crash, and this can reduce the effectiveness of later phases. This can be mitigated by applying a dynamic analysis tool, such as Valgrind[15], to the captured execution, which provides a more accurate starting point for the rest of the analysis. We have implemented a simple `use-after-free` detector as part of our prototype; combining this with other forms of analysis or with application-specific knowledge would be straightforward, and would allow other classes of bugs to be detected. The result of this initial analysis is generally

```

A: mov (global1) -> %rax
B: mov %rax -> (%rsp)
C: cmp $0, %rax
D: jne F
E: mov &fallback -> (%rsp)
F: mov (%rsp) -> %rcx
G: mov (%rcx) -> %rdx
H: add $48, %rdx
J: mov (%rdx) -> %rax
K: ret

```

(a) Thread 1

```

V: mov &struct1 -> (variable1)
W: mov &variable1 -> (global1)
...
X: mov $0 -> (global1)
Y: mov $0 -> (variable1)
...
Z: jmp V

```

(b) Thread 2

**Figure 1.** A buggy example of the privatize synchronization pattern. The author of thread 2 intended variable1 to be private after X, and so de-initialized it at Y, but thread 1 caches a pointer to it in a register at A which prevents it from being properly privatized. This leads to a bug: if A happens before X and Y happens before G, rdx will contain a bad pointer at instruction J, leading to an immediate crash.

a state machine with only one internal state, referred to as the proximal cause; for the example, it is shown in Figure 2(i).

### 3.2 Deriving earlier state machines

This proximal cause is accurate but not, by itself, sufficient to derive a fix, as by the time the crashing instruction is executed it is usually too late to attempt to avoid the bug. It is therefore useful to move the expression backwards through the captured execution, and hence to determine an equivalent expression which can be evaluated earlier. We do this inductively, moving back one instruction at a time through the DRS log and deriving a state machine for each. There are three main classes of relevant instructions: register-register arithmetic instructions, branches, and memory accesses; we consider each in turn (more complicated instructions can generally be treated as combinations of these basic classes).

#### 3.2.1 Arithmetic instructions

We assume that we have a state machine corresponding to the point immediately after the instruction in question, and we wish to construct one corresponding to a point immediately before it. If the register-register arithmetic instruction is regarded as a transformation on the program’s register state then this can be accomplished by applying the same transformation to the given state machine. For instance, in the example in Figure 1, the instruction preceding the proximal cause is `add $48, %rdx`. This transforms `rdx` into `rdx+48`. Applying this transformation to the proximal cause of the crash produces the state machine shown in Figure 2(h), which is valid at the start of instruction H. Other simple register-to-register arithmetic instructions can be handled in the same way, and hence the crash reason can be backtracked across any sequence of such instructions.

#### 3.2.2 Memory accesses

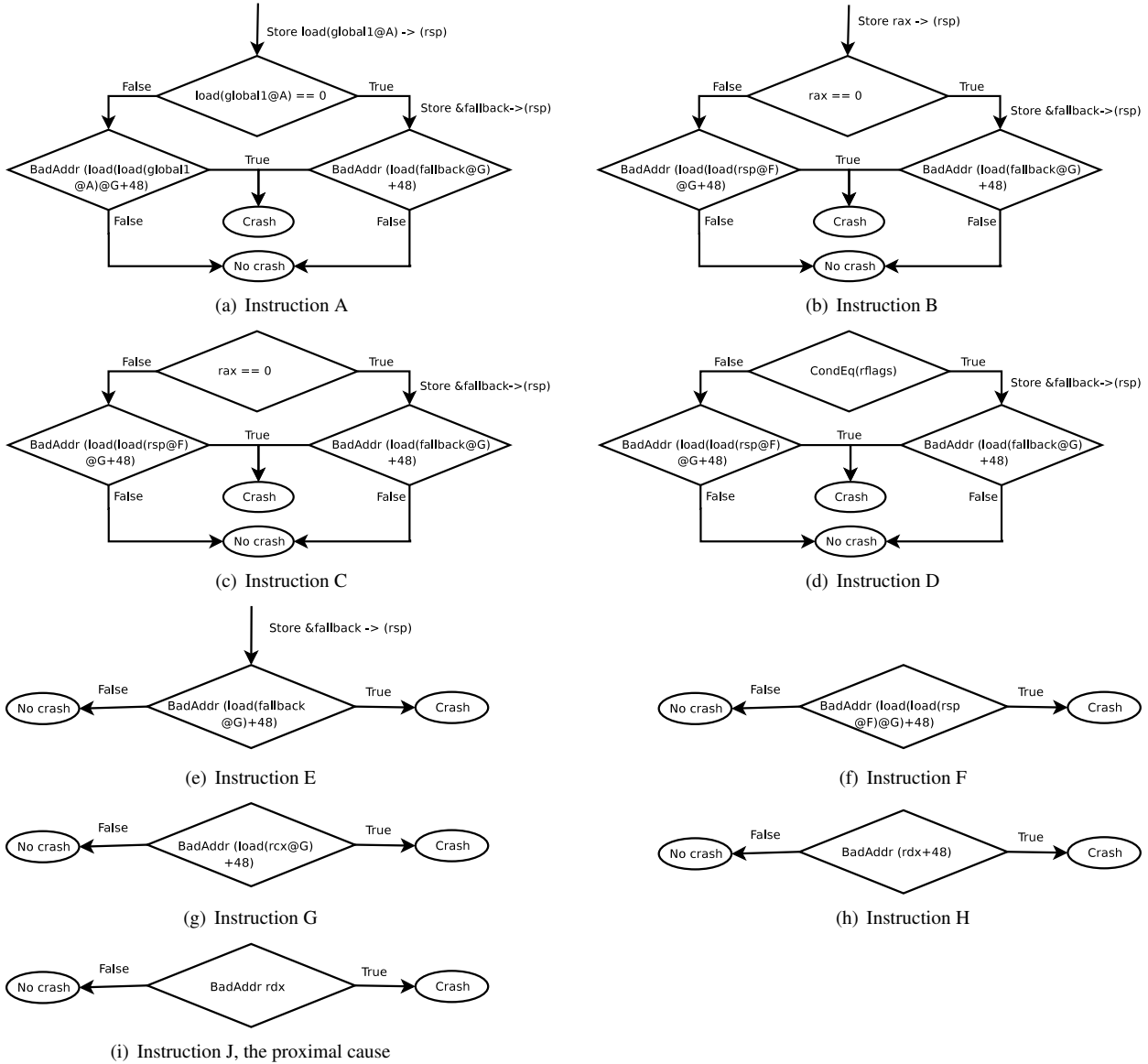
Memory accesses are more difficult to handle, for three main reasons:

- The pointer aliasing problem: given a load and a store, it is not always clear whether they access the same memory location.
- Non-determinism: due to the actions of other threads, it cannot be assumed that loading the same location twice will produce the same result.
- Temporality: the exact order in which loads and stores are issued is often important in the kinds of synchronization bugs which SLI targets, but the transformation-based approach used for arithmetic instructions does not preserve it.

We avoid these problems by leaving memory accesses explicit in the state machines in most cases, and so we do not need to be able to determine whether two pointers alias. This is fortunate, as doing so is difficult, even in a more conventional model checking environment where higher-level information and programmer annotations are available.

Consider again the example. We have already illustrated how to generate state machines for instructions J and H. The next instruction for which a machine must be derived is G, as that is immediately before H. This is simply the machine for H with `rdx` replaced with `load(rcx@G)`, because instruction G replaces `rdx` with a load of the memory location pointed at by `rcx`. This is shown in Figure 2(g). Likewise, instruction F replaces `rcx` with the contents of the memory location pointed at by `rsp`, and so the state machine for instruction F is as shown in Figure 2(f).

Accesses to the local stack are an important exception. Cross-thread accesses to the stack are extremely rare, while intra-thread accesses are extremely common,



**Figure 2.** State machines produced for the example program in Figure 1. Internal states are indicated by diamonds, terminal states by ovals, and edges by lines.

and so treating every stack access as being potentially involved in a race complicates the analysis for little gain in power. We use a simple heuristic, based on the contents of the DRS log, to attempt to resolve these accesses. When we encounter a store instruction, we check through the DRS log to determine whether it stored to the stack, and, if it did, which loads then loaded the stored value. If any of those loads occur in the state machine after the store instruction then we assume that they will always load the stored value, and so eliminate them.

Returning to the example, the state machine shown in Figure 2(b) is derived from that in Figure 2(c) by backtracking across instruction B, which stores `rax` to

`rsp`. First, the store is added to the start of the state machine, and then the stack resolution heuristic is used to link it to the load of `rsp` in the left-hand state. This allows the load to be replaced with a simple register reference, producing the machine shown.

### 3.2.3 Branch instructions

Branch instructions require more care, as they may involve parts of the program which were not used in the captured execution and for which no state machines will be available. Handling this would be simple if we could assume that any change to the program's control flow would be sufficient to avoid the bug, but this is not the

case as many branches are irrelevant to the observed bug. Consider, for instance, this example:

```
1  p = global_ptr;
2  if (!p->foo)
3      abort();
4  if (p->need_churn)
5      churn(p->bar->bazz);
```

Assume that the program was observed to crash on line 5 because some other thread was updating the structure and set `need_churn` before setting `bar`. `foo` must have been true on line 2, as otherwise the program would have aborted on line 3, and so if one were to assume that any changes to the program’s control flow would be sufficient to avoid the bug then one would conclude that ensuring that this code only ran when `foo` is false would fix the bug. This would force the program to always abort, which is unlikely to be a useful solution.

We determine the effects of unexecuted code using a simple static analysis. We first build a fragment of the program’s static control flow graph starting from the branch instruction and stopping when we encounter an instruction for which a state machine has already been derived (or an indirect branch which we cannot predict; see §3.2.4)<sup>1</sup>. We then eliminate any loops in the graph using a scheme described in §3.2.5, and so produce a directed acyclic graph of instructions rooted at the branch instruction and whose leaves are state machines which have already been derived. These state machines can then be propagated backwards through the graph using exactly the same inductive algorithm as we use to move backwards through the dynamic trace, allowing state machines to be derived for the internal nodes of the graph and ultimately for the original state machine.

In the example, instruction F could have been preceded by either D or E. Assume that in the observed crash it was preceded by D (or, equivalently, that the branch at D was taken). D is a branch instruction, and, statically, can be followed by either E or F. E is always followed by F, and a state machine has already been derived for F. The CFG therefore contains just three nodes, representing D, E, and F, with edges from D to E and F and one from E to F. We now search the CFG for any node which does not have a state machine but whose successors do; only E satisfies these constraints, and so we will derive a state machine for it first. E is a store to a stack location, and so the stack resolution heuristic is triggered. Assuming that the instruction was ever executed in the captured execution, it will predict that the load at F will load the stored value and the load will be eliminated. The resulting machine is shown in Figure 2(e). It is now possible to combine the machines for E and F to produce

<sup>1</sup>We also impose a limit on the number of instructions examined, but we have never reached this limit in practice.

one for D. This is illustrated in Figure 2(d) (the left-hand branch is the machine for instruction F and the right-hand one that for instruction E). The state machines for instructions C, B and A can now be derived using the mechanisms already illustrated, producing the machines shown in figures 2(c), 2(b) and 2(a).

Note that identical state machines would have been derived if the program had not taken the branch immediately before crashing (which might happen if the `fallback` structure was itself invalid); the only difference is that E’s state machine will be derived from the captured execution rather than a hypothetical execution generated by static analysis. This is useful: by eliminating uninteresting aspects of the observed behavior, SLI is able to generalize from one crash to closely related ones, and hence fix them at the same time, without eliminating an excessive number of safe schedules.

### 3.2.4 Indirect branches

Indirect branches, and any other branches which compute their target dynamically, pose an additional challenge here, as it is generally difficult to statically predict the target of the branch. Fortunately, the most common case is return instructions, and we can handle these by inlining called functions.

Other indirect branch instructions are more challenging. We solve this problem by using the captured trace as a simple oracle: if the branch instruction exists in the dynamic trace, we assume that the instruction will always branch to the same place (if it occurs several times then we use the last target). If the instruction did not occur, we place a special analysis-failed node in the graph. Once the CFG is complete, we eliminate all branches to these failed nodes; if that causes some node to have no known successors then that node is also marked as analysis-failed, and the process iterates until all failed nodes are removed. Branches taken in the captured execution will always be preserved at this stage, as the oracle will always be able to provide at least some prediction of their target. As such, this step can be seen as pruning the set of paths considered to be only those which are sufficiently similar to the captured execution.

### 3.2.5 Loops

Loops also complicate things. We avoid the problem by breaking them, removing a subset of the graph’s edges so as to eliminate the loop. We choose edges to remove based on two heuristics. First, we try not to remove any edges which are present in the captured execution. Second, we try not to partition the graph, so as many instructions as possible — and hence as much of the program’s behavior as possible — are represented in the final state machine.

Breaking loops has the useful side-effect of disambiguating the labels used to refer to load instructions: in a loop-free control flow graph, each instruction is executed at most once, and so dynamic instructions can be referred to unambiguously by their static location in the CFG. In the case of the x86 architecture, all common instructions which perform multiple loads can be converted into loops, which can then be broken in the usual way, and so this is sufficient to ensure that any static instruction issues at most one dynamic load and hence that load labels refer to at most one load.

#### 4. Locating relevant structures

Once the abstract read operation has been suitably characterized, the next step is to discover memory structures to which it can be applied. Rather than attempting to represent these (potentially complicated) structures explicitly, we instead represent them as specializations of the already-derived state machines. In other words, we take each state machine and convert it into a set of closely-related state machines, each of which will check one particular dynamic instance of the structure to be examined.

We rely on the DRS log in order to achieve this. Our approach is simply to identify all points in the log which are “similar” to the point for which the state machine was originally derived and to create a specialization for each by substituting in appropriate constant values for all references to thread-local state such as registers, and stack locations. This has the useful side-effect of “detaching” the state machine from a particular point in the program’s execution and allowing it to be applied to other memory configurations.

Similar is defined here to mean that the instruction pointer and call stack match. A more flexible definition would allow more dynamic instances of the structure to be discovered, and so more potentially-conflicting updates would be discovered and a more complete fix could be generated, but would increase the risk of unrelated concrete operations being falsely identified as instances of the dynamic operation, and hence memory locations being falsely labelled as dynamic instances of the structure to be synchronized, and hence the generation of overly-pessimistic fixes. We have not investigated this trade off in any detail.

Dynamically allocated memory complicates this simple procedure. The specializations, as described, in effect assume that memory locations which are shared between threads have a form of type-stability in the region of interest, which may not always be true. We mitigate this by limiting the range over which the specialized state machines are applicable. Newly-created specialized state machines are evaluated at the point in the log at which they are constructed, and a record kept of

the memory locations which they access. If any of these locations were obtained through a known dynamic allocator then the machine is restricted so that it is only used between the point at which that memory was last allocated and the point at which it is released (and if multiple locations are dynamically allocated then their ranges of validity are intersected). This effectively weakens the type-stability assumption, from assuming that a particular memory location always has a particular type to assuming that it can only be re-typed after being released and re-allocated.

We still assume that the read operation can be applied at any point where the memory structure has the correct type, and ignore any existing synchronization or control flow properties of the program which would restrict the situations in which it could be used<sup>2</sup>. If this assumption is false then SLI will produce an overly-conservative fix, synchronizing the read operation against write operations with which it could never actually race, fixing the bug but with unnecessarily high performance overhead. This could be ameliorated using programmer-supplied semantic annotations, but we have not needed to do so yet.

#### 5. Discovering possible racing write operations

Once the read operation has been identified and a suitable selection of dynamic instances of the racing structure discovered, SLI can proceed to investigate the crashing operation’s interactions with other threads, and thence to determine what additional synchronization would be necessary to prevent the observed crash. Our approach has two stages. First, we determine which stores could possibly conflict with loads of memory locations used by the read operation; this allows us to make the read operation behave as-if atomically. Second, we determine if there are any regions in the DRS log where executing the read operation atomically would still crash; these correspond to update operations against which the read operation must be synchronized.

Determining the possibly-conflicting stores is straightforward. We already have, from the previous analysis, a list of the memory locations which each state machine loads, and it is then simple to use the DRS log to discover all instructions which store to one of those locations. As before, we use calls to the memory allocator to temporally scope memory locations and avoid synchronizing against accesses to other structures which simply happen to have been assigned the same virtual addresses.

<sup>2</sup> This assumption also implies that we can identify all of the program’s memory allocation APIs.

To determine the regions of the log when a state machine predicts a crash, we replay the log over the range of validity of the state machine and evaluate the state machine before and after every possibly-conflicting store instruction. In many cases, the machine will always evaluate to `no-crash`, which indicates that the read operation is always safe when executed atomically and so no further synchronization needs to be added. If the operation is not always safe to execute atomically then there will be some `crash` regions in the log. If the starting and ending stores of the region are issued by the same thread then this corresponds to a simple update of the memory structure, and all that is needed is for the relevant range of instructions in the storing thread to be synchronized to not run in parallel with the read operation. Otherwise, the update operation spans several threads, and SLI is unable to generate a fix. In that case, we simply ignore the crashing region and synchronize against any remaining stores.

## 6. Producing a fix

We now have, for each specialized state machine:

- The set of loads which it issues which might be involved in a race;
- A list of ranges of the dynamic execution where issuing the read operation might be unsafe; and
- A list of all of the static store instructions which might race with the read operation.

The task is to combine these, across all of the specializations of each state machines, and synthesize from them an appropriate fix for the observed bug. For SLI, the fix will consist of a newly-created global lock plus a number of lock acquire and release operations. We must use these to ensure that the read operation does not occur in parallel with any of the store instructions, and does not occur during any of the unsafe ranges.

Note that we group all of the specializations of a given state machine together and consider them at once, but treat specializations derived from different state machines independently. The different unspecialized state machines represent different conceptual read operation, albeit closely related ones, while the specializations of a given state machine represent applying that operation to different in-memory structures. It is therefore likely that a race which is possible according to one specialization has a close equivalent which is possible with the other specializations of the same machine, but much less likely that it has an equivalent in specializations of a different state machine, and so combining the results is more appropriate in the former case than the latter.

### 6.1 Deciding which static instructions to protect

Comparing ranges of the program's dynamic execution to instructions or sets of instructions in its static image is not entirely well-defined. We must therefore either convert the static instruction references to points in the dynamic execution or dynamic ranges of execution to some suitable static representation; we choose to convert the dynamic ranges to fragments of the program's static control flow graph.

This is most straightforward if the dynamic range is contained entirely within a single function invocation, but even this case is not entirely trivial. The most obvious approach is to simply acquire the lock on the instruction which corresponds to the start of the range and drop it again on the instruction which corresponds to its end, but this is unsafe because there might be some other branch in between the two instructions which would allow the lock to be acquired but never released. It would be possible to design a simple analysis to detect such branches and to arrange to drop the lock if one is ever taken, but this would cause any irrelevant branches (as discussed in §3.2.3) to terminate the locked region, causing the fixes to be much less effective.

We therefore use a more subtle approach. We start from a control flow graph which contains just the instructions contained within the dynamic range. We expand this graph to include every instruction which is reachable from one of these instructions until we reach a dynamic branch (which is usually the return instruction at the end of the function), and then trim the graph back by removing any instruction which cannot reach one of the instructions which was in the original dynamic range. The remaining instructions are precisely those which are reachable from the instructions in the dynamic range and which are able to return to it, and it is these instructions which we protect. The instruction corresponding to the start of the dynamic range is labelled as the CFG's entry point; this will be significant later.

In many cases the region to be protected will include multiple function invocations. Functions which are contained entirely within the region are straightforward, and can be largely ignored: the mechanism already described will ensure that we acquire the lock precisely once before calling them, and release it precisely once after they return, which is generally sufficient<sup>3</sup>. Function invocations which are in progress at the start or end of the range, but not both, are more challenging. We handle them by expanding the dynamic range at the start and end by the minimum amount such that every function which is called after the start of the dynamic range has

<sup>3</sup> Abnormal control flow transfers such as `longjmp` break this pattern; we check whether the DRS log contains any such transfer out of the called function, and abort if it does, but do not otherwise handle them.

returned before it ends. This converts the more difficult case back into the simpler one, at the expense of using larger synchronization regions and hence potentially reducing concurrency by more than necessary. This technique is equivalent to nominating the last stack frame which is active throughout the range as being responsible for the bug and then treating any functions called from that frame as unmodifiable, opaque operations.

The set of loads is also expanded into a fragment of control flow graph in much the same way. We first map the instructions to be protected into a common stack frame, then build a control flow graph starting from those instructions, and then trim the CFG to remove any instructions which cannot reach an instruction which is to be protected. Finally, we nominate some instructions as entry points of the graph such that no entry point can reach another one and every instruction is reachable from at least one entry point. The store instructions are also converted into trivial single-instruction control flow graphs.

## 6.2 Building the patches

Each CFG is now converted into one or more patches. The core of each patch is a fragment of machine code which duplicates the parts of the original program which are covered by the CFG, but modified to release the lock at appropriate points, and a small number of trampolines which acquire the lock and branch to the entry point instructions in the machine code fragment. The instructions in the original program corresponding to these entry points are then replaced with branches to these trampolines. No other instruction in the original program is modified; this ensures that the lock is never released without having been first acquired, even if some code path not represented in the captured execution branches to an instruction in the middle of the critical section.

After all of the patches have been constructed, the resulting code can be compiled into an ELF shared library which can be loaded into the target program using `LD_PRELOAD` (or an equivalent mechanism) and which will apply the generated fix. Patches can sometimes overlap, and a moderate amount of care is required to ensure that the resulting composite fix behaves correctly, but this is not conceptually difficult; see §6.4 for an example.

Modifying entry point instructions is sometimes difficult, as the instruction to be patched might be too small to encode a branch instruction, and enlarging an instruction is dangerous unless one is able to prove that the following instruction is never the target of a branch. SLI avoids this issue by using the processor’s debugging facilities to set breakpoints on the acquiring instructions, and then transferring to the new code from the relevant exception handler. On x86 architectures, the breakpoint

instruction is a single byte, and so always fits in the instruction to be patched.

There is a risk that the proposed fix will introduce a deadlock. We avoid this issue with a simple timeout scheme: if a thread takes more than a second to acquire an SLI-introduced lock, it will time out, and immediately jump back to the unpatched code. This is sufficient to ensure forward progress, but renders the purported fix ineffective and could lead to very poor performance. Fortunately, it has not been a problem for us so far.

## 6.3 Selecting a fix

This process might produce multiple possible fixes if multiple state machines are available, and it is then necessary to select the best one. Some can be discarded by very simple heuristics (for instance, fixes in which every atomic section is a single memory access can be immediately eliminated), but there will in general be several possibilities to choose from. We use a very simple cost heuristic to do so: the cost of a fix is given by  $U \cdot n_u + C_s \cdot n_s + \sum_i s_i$  where  $n_u$  is the number of crashing regions which we discarded because they started and ended on different threads,  $n_s$  is the total number of critical sections,  $s_i$  is the number of accesses in the  $i$ th critical section, and  $C_s$  and  $U$  are constants reflecting the cost of introducing a new empty critical section and of only partially fixing the bug. SLI then selects the candidate fix with the lowest cost. Our prototype sets  $U = 1000$  and  $C_s = 10$ , strongly preferring fixes for which all unsafe states can be eliminated and weakly preferring fixes with a smaller number of critical sections. This simple heuristic has worked well in our experience to date (§7), as there are generally only a small number of possible fixes, all of which are correct and none of which would obviously lead to pathological performance.

## 6.4 Example

Our earlier example derived the state machine shown in Figure 2(a) for the program shown in Figure 1. The only register referenced in the machine is `rsp`, and so specialization will then simply amount to substituting in constant values for that register, which will have no interesting effects. The only locations accessed when evaluating this machine will be `global1`, `fallback`, and `variable1`. None of these are dynamically allocated, so the machine is valid throughout the log. The possibly-conflicting stores will be those which modify one of those locations; in this case, `V`, `W`, `X`, and `Y`.

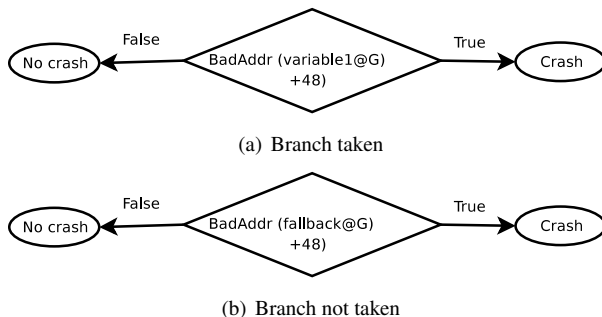
We must now evaluate the state machine before and after every possibly-conflicting store, in order to discover any unsafe dynamic regions in the log. In this case, it will always evaluate to `no-crash`, which is correct: if the reader were executed atomically, it would never

crash, regardless of what state the writer was in. There are therefore no unsafe regions, and the instructions to be protected are simply the load set and the individual possibly-conflicting store instructions.

These are now converted into patches. The store instructions are trivial, and so we only explain the procedure for the load set. The control flow graph will contain every instruction from A to G in Figure 1 with A nominated as the entry point. The patch’s machine code fragment will therefore consist of duplicates of those instructions followed by a lock release operation and a branch back to instruction H, and instruction A will be modified to acquire the lock and jump to the start of the patch. Note two important properties of this patch:

- The instruction which crashed, J, is not included in any critical region. By the time instruction J executes, the race has already happened, and so it is too late to try to prevent it and protecting that instruction would not be helpful.
- The fix is correct: it completely eliminates the observed bug.

This is not the only fix which will be suggested, however, as state machine generation will also have produced machines for other instructions in the crashing thread. Consider, in particular, the state machine shown in Figure 2(f). As this accesses both registers and the stack, it can be specialized. Assume now that the DRS log contains at least one instance of the read operation in which the branch at D is taken and one in which it is not. In that case, there will be two specializations, shown in Figure 3, one corresponding to the branch-taken case and one to the branch-not-taken one.



**Figure 3.** Results of specializing Figure 2(f)

These machines produce another suggested fix. The machine in Figure 3(a) will predict a crash whenever `variable1` is zero, i.e. whenever thread 2 is between instructions Y and V, and so there will be at least one unsafe dynamic range; assume for the sake of exposition that there is precisely one such range. We must now convert that unsafe range into a patch. The CFG will contain all of the instructions from Y to V, crossing the

```
V: mov &struct1 -> (variable1)
W: mov &variable1 -> (global1)
...
X: mov $0 -> (global1)
Y: acquire_and_jump Y'
U: ...
Z: jmp V
Y': mov $0 -> (variable1)
U': ...
Z': jmp V'
V': mov &struct1 -> (variable1)
a: release()
    jmp W
```

**Figure 4.** Partial patch based on the state machine in figure 3(a).

back edge of the loop. A patch will be created which duplicates these instructions before dropping the lock and jumping back to instruction W, and instruction Y will be modified to acquire the lock and branch to this patch, as shown in Figure 4. The only load instruction mentioned in the state machines is G, and that will be instantiated into a trivial single-instruction patch as the store instructions were in the previous example.

The potentially conflicting stores will be Y and V in this case. These are more interesting, as they overlap with the existing critical section. In particular, there are now two versions of V in the program (the original version and the one in the previous patch), and both must be patched, while Y, as the entry point of the previous patch, has been effectively moved, and the new version is the one which must be patched. The resulting patch set will be as shown in Figure 5. These patches will also fix the bug, and, assuming that there are only a small number of instructions between Y and Z, they will be preferred by the prioritization heuristic, because they require fewer critical sections.

It could be argued that this smaller fix is inferior to the larger alternative, despite completely eliminating the crash and potentially imposing lower overhead, as it is less “sympathetic” to the existing structure of the program. The author of thread 2 had presumably intended to privatize the structure at instruction X, and so might reasonably have defined a “correct” fix to be one which ensures correct privatization, a goal which is achieved by the first fix but not by the second. SLI, by contrast, has no notion of intended behavior, or indeed any form of good software engineering practice, and so selects its fix based on the more mundane concerns of avoiding the crash and minimizing the impact on performance. This is both a strength and a weakness: a strength in that it increases the likelihood that a low-overhead fix will be

```

V:  jmp V''
W:  mov &variable1 -> (global1)
...
X:  mov $0 -> (global1)
Y:  acquire_and_jmp Y'
U:  ...
Z:  jmp V
Y': acquire_and_jmp Y''
U': ...
Z': jmp V'
V': jmp V'''
a:  release()
b:  jmp W
Y'': mov $0 -> (variable1)
     release()
     jmp U'
V'': mov &struct1 -> (variable1)
     release()
     jmp W
V''':mov &struct1 -> (variable1)
     release()
     jmp a

```

**Figure 5.** Complete patch based on the state machine in Figure 3(a), extending Figure 4.

found, and a weakness in that the fixes cannot be translated back to source-level patches and applied unthinkingly by the application programmer.

Producing a long-term, maintainable fix for a bug is, in general, a different problem to producing a short-term one which simply eliminates its symptoms, and the techniques used by SLI are much more applicable to the latter. This will be true of any approach which does not rely on programmer annotations (ignoring trivial systems which just pattern-match the buggy code against a library of common bugs and their standard fixes).

## 7. Evaluation

As shown in tables 1 and 2, our prototype implementation is able to fix a reasonable selection of artificial bugs within a few seconds, given only the program binary and a log which shows the bug reproducing, and is able to fix at least one real-world bug in a little over five minutes. The data also shows that the prototype avoids state machine explosion, generating only a small number of distinct state machines each of which has only a small number of states (a little under four on average, and in no case more than twenty-three). All experiments were conducted on an Intel Q6600 with 8GiB of RAM running 64-bit Linux 2.6.28.

One important observation here is that while the total time taken by the thunderbird bug was much greater

than for any of the other bugs, the bulk of this time was spent in the replay engine and the most of the remainder in loading the larger memory image, with the time spent in the analysis phases largely unchanged. This is encouraging. One obvious criticism of this kind of approach is that it might suffer an equivalent of the model checking state explosion problem and hence collapse when faced with bugs in realistically-sized programs, and if that were happening then it would show up as an increase in the time spent in the analysis phases. That has clearly not happened in this case.

We also investigated how the analysis depends on the exact way in which a particular bug is reproduced, by running our tool on five independent reproductions of the `glibc` bug. Every reproduction produced the same set of state machines and suggested fixes. The time taken by the analysis process varied significantly, however (from two to thirteen seconds), mostly because the time taken to reproduce the bug varied and hence produced differently sized logs to be parsed. Similar results were obtained for the other bugs.

### 7.1 Detailed discussion of test bugs

`toctou` is a simple two-thread time-of-check, time-of-use race. In this test, one thread loops incrementing a counter, while another thread repeatedly issues pairs of loads of the counter and asserts that the loads returned the same value. Our prototype generates a single suggested fix consisting of two critical sections, one protecting the write-back of the incremented counter in the first thread and the other protecting the two loads in the second thread. This is a correct and minimal fix.

`twovar` is a two-variable atomicity violation. In this test, there are two global variables, and one thread loops setting both to 5 and then setting both to 7 while another thread loops loading both and asserting them to be equal. SLI again produces a single correct fix in this case. The core of the second thread consists of two instructions:

```

l1: mov (global1) -> %rax
l2: cmp %rax, (global2)
    jne __assert_fail

```

The state machine for l1 is

```

if load(global1@l1) != load(global2@l2)
  then crash
  else no-crash

```

This produces a single candidate fix, consisting of three critical sections: one protecting instructions l1 and l2 in the second thread; one protecting the interval of the first thread where `global1` is 5 and `global2` 7; and the other protecting the interval of the first thread where `global1` is 7 and `global2` is 5. This correctly elimi-

Name of test	Nature	Number of fixes	Size of logfile	Number of state machines	Total number of state machine states
toctou	Synthetic TOCTOU race	1	28MiB	9	19
twovar	Synthetic two-variable atomicity violation	1	31MiB	13	31
publish	Synthetic broken publish pattern	2	31MiB	8	24
privatize	Synthetic broken privatize pattern	2	43MiB	9	25
glibc	Kernel of a genuine atomicity violation	8	48MiB	19	139
thunderbird	Genuine TOCTOU	1	758MiB	7	15

**Table 1.** Summary of the results for each test case.

Name of test	Total time taken	Loading initial memory image	Replaying logfile	All analysis phases
toctou	$0.51 \pm 0.01$	$0.18 \pm 0.01$	$0.19 \pm 0.00$	$0.15 \pm 0.00$
twovar	$0.75 \pm 0.01$	$0.18 \pm 0.01$	$0.42 \pm 0.00$	$0.14 \pm 0.00$
publish	$0.48 \pm 0.03$	$0.18 \pm 0.02$	$0.17 \pm 0.00$	$0.13 \pm 0.00$
privatize	$2.34 \pm 0.78$	$0.18 \pm 0.02$	$1.04 \pm 0.01$	$1.12 \pm 0.75$
glibc	$2.46 \pm 0.06$	$0.19 \pm 0.01$	$1.20 \pm 0.05$	$1.07 \pm 0.02$
thunderbird	$356 \pm 1$	$1.76 \pm 0.02$	$354 \pm 1$	$0.58 \pm 0.01$

**Table 2.** Time taken for the various phases of operation, in seconds. Mean and standard deviation of analyzing a single reproduction five times.

nates the bug. Note that, as it involves two variables, it would not be eliminated by Kivati[5].

`publish` is a buggy implementation of the `publish` pattern. In this pattern, a structure is initialized by one thread and then published by writing its address into a global pointer. Other threads then occasionally read this global pointer and, if it contains a non-NULL value, use the referenced object. This is safe if correctly implemented, but in this test the programmer published the structure before finishing constructing it, which leads to the other thread eventually crashing. The test case consists of two threads, one of which repeatedly publishes and un-publishes a structure and the other of which repeatedly tests whether it has been published and, if it has, attempts to use it in a way which leads to an immediate crash if initialization is not complete. Note that in this case the bug is in the publishing thread, but the crash is observed in the consuming one.

Our tool produced two suggested fixes in this case. One of these was expected, consisting of two critical sections, one protecting the consuming thread from the point at which it loaded the pointer to the point at which it used its contents and the other protecting the producing thread from the point at which it published the structure to the point at which it finished initializing it. This is a correct fix.

The other suggestion was somewhat surprising. Our test harness repeatedly published, initialized, unpub-

lished, and then deinitialized the same structure. SLI was able to look through this pattern, and determined that it was sufficient to prevent the consuming thread from validating the published structure at any point between the deinitialization in one iteration and the initialization in the subsequent one. It therefore suggested a fix which protected the load which the consuming thread used to perform validation in one critical section and the range of the publishing thread from the deinitialization to the subsequent initialization in another, completely ignoring the accesses related to publishing and un-publishing the structure. While somewhat surprising, this is also a correct fix, completely preventing the observed bug, and, as it produces slightly smaller critical sections, might preserve a greater degree of concurrency than the more obvious one. In this case, SLI has produced a fix which is actually better than might be expected of a purely manual process, precisely because it operates at a very low level without any reference to the program’s intended semantics.

`privatize` is the converse of `publish`: a thread is attempting to make a structure private, and does it incorrectly. It is similar to the example program in Figure 1, which has already been extensively discussed.

`glibc` is a kernel of `glibc` bug 2644 [4], which affected versions of `glibc` up to 2.5 and could lead to a crash if multiple threads were shut down at the same time. A simplified version of the code involved is shown

in Figure 7.1, where `forcedunwind` and `done_init` are global variables. Note that the bug here depends on the compiler’s optimizer, and is not apparent at the source-code level<sup>4</sup>. SLI operates entirely at the machine-code level, and so this does not present any additional complexity.

SLI produced six suggested fixes when run on a log generated by running this test. The first of these had five critical sections: one covering the load on line 1 to the load of `done_init` on line 3, and one each for each of the stores on lines 4, 6, 13, and 14. The other suggestions were supersets of this suggestion, extending it to include various accesses in `pthread_barrier_wait`. This illustrates an important weakness of the approach. Because SLI does not know anything about any OS-provided functionality, it cannot take advantage of any existing synchronization present in the program (in this case, the `pthread_barrier_waits` make the critical sections protecting statements 13 and 14 redundant). It also means that the analysis must explore these standard functions, and can sometimes attempt to “fix” the benign races inherent in synchronization operations, which is unlikely to be productive.

`thunderbird` is Mozilla bug number 391259[13], a simple time-of-check, time-of-use race in the IMAP client component of Thunderbird, a popular open-source e-mail client. We modified Thunderbird to include some additional debugging messages and used a custom scheduler in order to make the bug reproduce more readily; the test is otherwise identical to the behavior which a user might have encountered. The relevant parts of the program are as follows:

```
void nsImapProtocol::CloseStreams() {
    if (m_transport)
        m_transport = nullptr;
}
PRBool nsImapProtocol::ProcessCurrentURL() {
    if (m_transport)
        m_transport->SetTimeout(
            TIMEOUT_READ_WRITE, PR_UINT32_MAX);
}
```

If `m_transport` is set to `nullptr` by `CloseStreams()` in between the two accesses in `ProcessCurrentURL` then the program will crash. This is essentially the same bug as `toctou`, but embedded in a much larger program. As such, the final result is similar: a single suggested fix, with two critical sections, one containing the two accesses in `ProcessCurrentURL` and one containing the assignment in `CloseStreams`. This fixes the bug.

<sup>4</sup>Unfortunately, only the 32-bit x86 version of `gcc` optimizes the function like this, and our implementation of SLI assumes a 64-bit x86 program, and this prevented us from testing with the real bug.

## 8. Related work

A number of previous systems have tackled similar problems. Most recently, Kivati[5] attempts to fix single-variable atomicity violations automatically by combining a static analysis pass with some runtime support. The result is able to prevent many common kinds of race-like bugs with low overhead. The key difference between their approach and ours is that Kivati does not produce targeted fixes for specific, observed, bugs, but instead eliminates a wide selection of schedules which are likely to be erroneous. This means that they are able to fix some bugs which have never been seen, whereas SLI requires at least one observation of it, but also means that the fixes generated by Kivati cannot be easily separated from the Kivati runtime. This in turn means that the overheads of Kivati, while low, must be paid for as long as the bug needs to be fixed, whereas SLI’s overheads, while somewhat higher due to the use of a DRS, only need to be paid until a fix is generated. Kivati is also limited in that it can only protect against single-variable atomicity violations; it would not, for instance, have been able to fix the `twovar` example described above.

Another approach, taken by systems such as Isolator [20] and Tolerace[21], restricts the problem domain to asymmetric races, where one thread is correctly following a locking discipline while some other thread is not, and seeks to ensure that the correct thread continues to be correct despite the misbehavior of the incorrect one. This might, for instance, be useful if the correct thread is controlled by an application while the incorrect one is controlled by a library which the application writer is unable to modify.

Atom-Aid[12] also attempts to mitigate race bugs, in this case by using hardware transactional memory to bundle sequences of memory accesses into transactions according to some heuristics. This reduces the number of permissible schedules and hence the scope for memory ordering related bugs. Provided the necessary hardware is available, this is simple and reasonably efficient, and should eliminate a reasonable selection of non-trivial bugs. The main downside of the approach is that it requires non-standard (and presently non-existent) hardware, which makes it less practically useful than it otherwise would be.

Dimmunix[10] solves the closely-related problem of preventing deadlocks in code which uses mutex-like synchronization. It relies, as SLI does, on having observed the bug at least once, and then synthesizes additional synchronization which prevents it from re-occurring. The most important difference is that whereas Dimmunix need only be concerned with lock operations, which are relatively rare, SLI must consider memory accesses, which are extremely common, and

```

_Unwind_ForcedUnwind() {
    if (forcedunwind == NULL)
        pthread_cancel_init();
    forcedunwind();
}
pthread_cancel_init() {
    if (done_init) return;
    forcedunwind =
        _forcedunwind_impl;
    done_init = 1;
}
(a) Before optimizations

_Unwind_ForcedUnwind() {
1:  l = forcedunwind;
2:  if (l == NULL &&
3:      done_init) {
4:      forcedunwind = l =
5:          _forcedunwind_impl;
6:      done_init = 1;
7:  }
8:  l();
}
(b) After optimizations

while (1) {
10:  pthread_barrier_wait();
11:  _Unwind_ForcedUnwind();
12:  pthread_barrier_wait();
13:  done_init = 0;
14:  forcedunwind = NULL;
}
(c) Test harness

```

**Figure 6.** Source code for the glibc test case.

this requires the use of somewhat different techniques. Gadara[24] also attempts to solve deadlock bugs using a completely different offline analysis to detect bugs before they happen, but requires access to the program’s source code, which may not always be available.

There have also been a number of attempts to automatically fix heap management bugs, such as buffer overflows and use-after-free errors, including AutoPaG[11] and Exterminator[16]. These systems both take an example of a buffer overflow bug (assumed to be deterministic) and use various analyses to determine the root cause of the bug, eventually using this to produce a potential fix. In that, they are remarkably similar to the system currently under discussion; the main difference being the type of bug targeted.

All of these systems attempt to fix bugs or otherwise prevent them from happening. An alternative strategy is to make errors less serious when they do happen. The most famous example of such a strategy is probably failure obliviousness[22], which waits until the protected program makes an invalid memory reference and then attempts to fix it from the resulting exception handler. DieHard[3] is conceptually similar, but works preemptively rather than from a fault handler, by guessing where memory errors are likely to occur and modifying the program’s memory map to make those errors as harmless as possible. In this way programs are able to continue executing in spite of the presence of errors which would otherwise cripple them. Failure obliviousness cannot, however, completely remove any errors, and so can be seen as complementary to the other schemes discussed here.

RX[19] takes a third strategy. Here, rather than attempting to fix the bug, an attempt is made to determine which subset of a program’s functionality is bug-free, and then to restrict the program’s inputs to only exercise that functionality. The result is that inputs which might have triggered the bug continue to produce incorrect output, but the damage is at least contained rather

than propagating throughout the program and potentially leading to a crash.

All of these approaches are primarily dynamic in nature. There have also been many systems which attempt to detect races using static analysis, such as [18] or [7], or via model checking, such as [6]. These techniques have the advantage that the bug to be fixed does not first have to be exhibited (and, indeed, they are often used to discover bugs in code which has never been run) and have precisely no runtime overhead, so are attractive wherever they are applicable. However, many programs have sufficiently complicated structures that a sound static analysis is impractical, and they cannot as easily compensate for this by taking advantage of any exhibition which might be available or by being directed towards fixing a specific bug.

## 9. Future work

There are a number of potential extensions of this work, beyond the obvious ones of broadening the evaluation and improving performance. At a high level, SLI must balance the use of static analysis, which considers many possible executions and produces general fixes, and dynamic analysis, which considers only the observed execution and produces much more targeted fixes. We do not claim to have found the optimal combination of these two approaches, or even that a global optimum exists; better characterizing the trade-offs involved is likely to suggest useful improvements.

There is a similar balance to be struck between attempting to be completely generic and using semantic knowledge, both of the program and of its libraries. At present, we make very little use of this information, and so our implementation is generic across a wide range of applications but struggles with more complicated bugs. Incorporating more semantic information, or providing a generic way for programmers to introduce their own semantic models, might improve our ability to produce useful, performant fixes. One particularly intriguing ap-

proach would be to combine SLI with an invariant inference scheme such as Daikon[8] or DIDUCE[9], which would allow us to obtain such semantic information without compromising SLI's current ability to run on almost arbitrary unmodified binaries. We intend to investigate this idea more fully in the future.

The current timeout-based mechanism for avoiding deadlocks is also a weakness, and can lead to poor performance or incompletely fixed bugs. This could be ameliorated to some extent by using further static analysis to detect which patches are likely to lead to deadlocks and to penalize them in the fix selection phase. Alternatively, SLI could be integrated with a deadlock immunity system such as Dimmunix[10], which would detect and fix deadlocks when they happen. The system would then hopefully converge on a state which is both deadlock and race free.

## 10. Conclusions

We have presented SLI, a system for automatically fixing specific synchronization bugs in shared-memory programs using only their binaries, with minimal user intervention. We have demonstrated that it can be used to fix real-world bugs in at least some cases, and discussed the compromises and trade-offs which are necessary in order to produce a practically useful implementation. While these techniques do have a number of limitations and drawbacks, we feel that they provide a useful basis for ongoing work to extend the set of situations in which they are applicable.

## References

- [1] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *SOSP*, 2009.
- [2] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345, 1994.
- [3] E. Berger and B. Zorn. DieHard: probabilistic memory safety for unsafe languages. In *PLDI*, 2006.
- [4] N. Campbell. Race condition during unwind code after thread cancellation. Glibc bug number 2644, 2006.
- [5] L. Chew and D. Lie. Kivati : Fast Detection and Prevention of Atomicity Violations. In *EuroSys 2010*, pages 307–319, 2010.
- [6] T. Elmas, S. Qadeer, and S. Tasiran. Precise race detection and efficient model checking using locksets. Technical Report MSR-TR-2005-118, Microsoft Research, 2006.
- [7] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. *ACM SIGOPS Operating Systems Review*, 2003.
- [8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.
- [9] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. *International Conference on Software Engineering*, 2002.
- [10] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.
- [11] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie. AutoPaG: towards automated software patch generation with source code root cause identification and repair. In *ASIACCS07*, Singapore, 2007. ACM.
- [12] B. Lucia, J. Devietti, L. Ceze, and K. Strauss. Atom-Aid: Detecting and Surviving Atomicity Violations. *IEEE Micro*, 29(1):73–83, Jan. 2009.
- [13] W. Mery. crash [@ nsproxyreleaseevent::run()]. Mozilla bug number 39125, 2007.
- [14] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.
- [15] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):100, 2007.
- [16] G. Novark, E. Berger, and B. Zorn. Exterminator: Automatically correcting memory errors with high probability. *PLDI*, 2007.
- [17] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. Lee, and S. Lu. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *SOSP*, 2009.
- [18] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI*, 2006.
- [19] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies: a safe method to survive software failures. In *SOSP*, 2005.
- [20] G. Ramalingam, S. Rajamani, V. Ranganath, K. Vaswani, and S. Rajamani. Isolator: Dynamically Ensuring Isolation in Concurrent Programs. In *ASPLOS'09*. ACM, 2009.
- [21] P. Ratanaworabhan, M. Burtscher, D. Kirovski, B. Zorn, R. Nagpal, and K. Pattabiraman. Detecting and tolerating asymmetric races. *PPoPP*, 2009.
- [22] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and W. Beebe Jr. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, 2004.
- [23] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.
- [24] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *OSDI*, pages 281–294, 2008.
- [25] C. Zamfir and G. Candea. Execution Synthesis: A Technique for Automated Software Debugging. In *EuroSys*, 2010.