

# Speculative Lock Insertion

## 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 [Shavit 1997] and automatic parallelization [Bacon 1994], 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 on the process of updating it and an unfortunate interleaving causes 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 characterisation 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 characterise and approximate the parts of the crashing thread's behaviour which are most relevant to 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 or ODR[Altekar 2009], both of which record only 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[Zamfir 2010] is an extreme form of this approach, and logs nothing at all but instead attempts to recreate the path to 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[Musuvathi 2008] could be used to detect unknown bugs, which could then be passed to SLI to be automatically characterized and fixed.<sup>1</sup>

### 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 and to reify it. We start by using a dynamic analysis on the DRS log to determine the first instruction where the program has definitely gone wrong, and to produce a direct explanation of the crash considering only that instruction. This might, for instance, indicate that the program crashed because the instruction at address 40037e dereferenced register `rdi` which contained an invalid pointer. This is referred to as the proximal cause of the crash. We then expand upon this cause, finding earlier causes by translating it backwards through the captured log using a scheme which combines both static and dynamic analyses, producing a series of state machines which approximate steadily growing suffixes of the crashing thread's execution and which can be treated either as "explanations" of the crash or as logical operations performed by the thread towards the end of its life.

We use a running example, shown in figure 1, to illustrate some of the details of our approach. The example shows a buggy instance of the common structure privatization pattern in an x86-like assembly language. Thread 2 initializes (`V`) and publishes (`W`) a structure, does some unrelated work, and then attempts to privatize (`X`) and de-initialize (`Y`) the structure. Meanwhile, thread 1 loads the pointer published by thread 2 (`A`), checks whether it is valid (`B`, `C` and `D`), and, if it is not, loads a fallback version (`E`), before attempting to use it (`F` through `J`). This will lead to a crash if instructions `X` and `Y` occur between instructions `A` and `G`, as in that case `rdx` will contain a bad pointer at instruction `J`, which will cause

an immediate crash. Figure 2 then shows some of the state machine generated by our algorithm.

#### 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 the most 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[Nethercote 2007], 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 single-state state machine which captures the reason for the crash using only information which is available at the instruction on which the error is detected; 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 faulty instruction is executed it is usually too late to attempt to fix it. 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 in the execution. We do this inductively, moving back one instruction at a time through the DRS log. There are three main classes of relevant instructions: register-register arithmetic instructions, memory accesses, and branches; 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, by induction, 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

<sup>1</sup>I want to use the phrase closed-loop here

```

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.

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.<sup>2</sup>

<sup>3</sup>

### 3.2.2 Branch instructions

Branch instructions also require 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.<sup>4</sup>

One simple approach would be to assume that taking such branches is sufficient to avoid the crash. Unfortunately, this is often not true: there are often irrelevant and distracting branches in the region of code to be examined, and they must be eliminated. Consider, for instance, this example:

```

1  p = global_ptr;
2  assert(p->foo);
3  if (p->need_churn)
4      churn(p->bar->bazz);

```

Assume that the program was observed to crash on line 4 because some other thread was updating the structure and set `need_churn` before setting `bar`. Presumably, the assertion on line 2 was true in this instance, and so naively assuming that any changes to the control flow are sufficient to avoid a repeat of the crash would cause SLI to attempt to fix it by making the assertion always evaluate to false. This is unlikely to lead to an acceptable solution.

We determine the effects of unexecuted code using a simple static analysis. The first stage of this analysis is to build an approximation of the program’s control flow graph starting from the branch instruction and stopping when we encounter an instruction for which we already have a state ma-

<sup>2</sup>The reason this works is something to do with the underlying category of state machines being linear with respect to register-register instructions expressed as homomorphisms, but a) that’s not really the kind of thing you say in a systems paper, and b) I don’t understand it well enough to explain it correctly, anyway.

<sup>3</sup>Our implementation uses libVEX to decode x86 instructions into a sequence of micro-operations which can be used as input to this process.

<sup>4</sup>SSP

chine<sup>5</sup>. These state machines can then be propagated backwards through the graph exactly as if the instructions had been executed in the dynamic trace, allowing us to assign a state machine to the root of the graph and hence to the original branch instruction.

### 3.2.3 Memory accesses

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

- The pointer aliasing problem<sup>6</sup>: 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 synchronisation 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. Loads are represented by special load expressions, tagged with the instruction which issued them, while stores are listed on the edges of the state machines.

Because the memory accessing instructions remain explicit in the state machines, 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<sup>7</sup>.

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, and so treating

<sup>5</sup>We also impose a limit on the number of instructions examined, but this was not reached in any of our experiments.

<sup>6</sup>need cite

<sup>7</sup>need cite

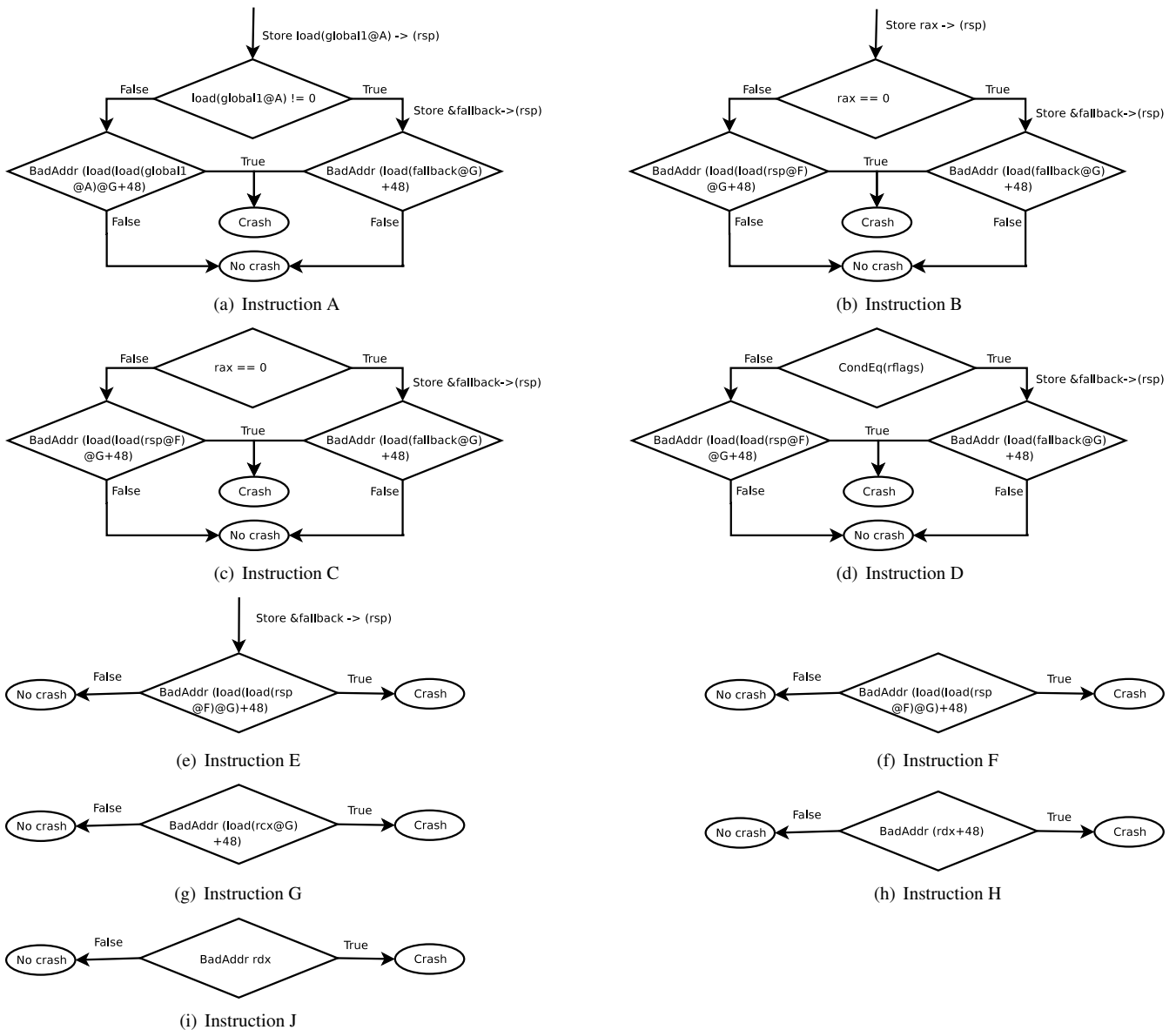


Figure 2. State machines produced for the example program

every stack access as being potentially involved in a race significantly complicates the analysis for little gain in power. We therefore 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.

### 3.2.4 Example

In the example, the next instruction which must be assigned a state machine is D: `je F`, as that preceded instruction F

in the captured trace. It is followed by instruction F, which already has a state machine, and E, which does not, and E is followed by 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<sup>8</sup>. We then proceed to derive a state machine for E, which is a simple store to a stack location. Assuming that this store was ever executed in the captured execution, the stack resolution heuristic will predict that the load at F will load the stored value. The load will therefore be eliminated, and so the state machine for instruction E will be as shown in figure 2(e). It is now possible to combine the machines for E and F to produce one for D. This is illustrated

<sup>8</sup>Diagram?

in figure 2(d). Backtracking further to instruction A will then produce the state machine illustrated in figure 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. This example also illustrates that the stack resolution heuristic is not equivalent to assuming a completely static data flow graph, as it is interleaved with control flow discovery and hence respects simple control-flow dependencies. Separating the two processes into independent phases would lose this property, and result in far less accurate characterizations.<sup>9</sup>

### 3.2.5 Indirect branches

Indirect branches, and any other branches which compute their target dynamically, pose an additional challenge here, as it is often difficult to statically predict the target of the branch. The most common case is return instructions, and we handle these by inlining called functions, eliminating most return instructions completely. The remainder correspond to functions which had started but not completed at the time of the crash, and we assume that if any of those return then the bug has been avoided.<sup>10</sup>

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.

### 3.2.6 Loops

Loops also complicate this simple algorithm. 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. Identifying relevant structures

Once the abstract read operation has been suitably characterised, 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 specialisations 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 specialisation 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.

This step often results in the construction of duplicate state machines, and these are eliminated in the obvious way<sup>11</sup>.

Similar is defined here to mean that the instruction pointer and call stack match. Other definitions would also be possible; all that SLI requires for correctness is that the point for which the machine was derived is similar to itself. 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<sup>12</sup>. We have not investigated this trade off in any detail.

<sup>9</sup> ref phase order problem?

<sup>10</sup> This should be much earlier!

<sup>11</sup> Someone complained that the previous draft didn't specify the definition of duplicate, but I really can't be bothered to spell it out.

<sup>12</sup> Holly run on sentence, Batman.

Dynamically allocated memory complicates this simple procedure. The specialisations, as described, in effect assume that memory locations which are shared between threads have a form of type-stability in the region of interest. We mitigate this by limiting the range over which the specialised state machines are applicable. Newly-created specialised 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.<sup>13</sup>

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 synchronisation or control flow properties of the program which would restrict the situations in which it could be used<sup>14</sup>. If this assumption is false then SLI will produce an overly-conservative fix, synchronising 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 synchronisation 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 generally correspond to update operations against which the read operation must be synchronised.

Determining the possibly-conflicting stores is straightforward. We already have, from the previous stage, a list of the memory locations which each state machine loads, and it is then simple to use the DRS log to discover all instruc-

tions which store to one of those locations. As before, we use calls to the memory allocator to scope memory locations and avoid synchronising against accesses to other structures which simply happen to have been assigned the same virtual addresses.

To determine the regions of the log when a state machine would 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 synchronisation 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 synchronised 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 synchronise against any remaining stores.

## 6. Producing a fix

We now have, for each specialised 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.
- 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 state machines, and synthesise from them an appropriate fix for the observed bug. For SLI, the fix will consist of a newly-created global (recursive) 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.

We now simplify the problem by expanding the dynamic ranges such that their start and end points are in the same dynamic function invocation. Since the start and end are executed by the same thread, there must be some frame which is common to both call stacks, and SLI selects the function corresponding to the last such frame to insert synchronisation into. If the start and end points are not already in the desired frame then this will require the region to be expanded until they are. This is straightforward. To move the start, SLI simply finds the last function call made by the desired frame before current start point and designates that function call to be the new start point. Likewise, the end point is moved to be immediately after the last function call made by the desired frame before the current end point. The set of loads

<sup>13</sup> This isn't right: after specialisation, we could still refer to pointers in the heap, and we won't track validity of the referenced objects. Probably doesn't matter in practice.

<sup>14</sup> This assumption also implies that we can identify all of the program's memory allocation APIs.

can likewise be moved to be within a single function by assigning loads which are issued by called functions to the call instruction.

These synchronisation regions are now converted into code patches. Each code patch consists of a fragment of the original program, modified such that patches always run atomically with respect to other patches. There will be at most one such patch for every synchronisation region derived above.

The first stage of building a patch is to determine which instructions should be in it. The procedure for doing so is similar but not quite identical for each of the types of region. For simple store instructions, it is easy: the patch contains just the store instruction in question. For unsafe regions, we start by building a control flow graph starting at the start of the region and terminating at its end or at the end of the enclosing function and then remove any instructions which cannot reach the end point; any instructions which remain in the control flow graph will be represented in the patch. The approach for the load set is similar: we build a control flow graph starting from every load in the load set and terminating at the end of the function, and then remove any instruction which cannot reach a load in the load set.

The patch can now be constructed by duplicating all of the instructions which are contained in it, being careful to update any instruction pointer-relative addresses in an appropriate way. All of the instructions in the set of instructions for which the patch was derived are modified to acquire the lock and then branch to the patch, and wherever a branch in the patch targets an instruction not in the patch it is modified to drop the lock and then branch back to the original code. In this way it is possible to ensure that no two patches can ever execute in parallel, and that lock acquisitions are always precisely matched to lock releases (ignoring abnormal control transfers such as `longjmp`)<sup>15</sup>. Once every instruction set has been converted to a patch in this way, 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.

Sometimes code patches can overlap. A moderate amount of care is required to ensure that the resulting composite fix behaves correctly, but this is not conceptually difficult provided that the lock is recursively acquirable; see below for details. SLI applies some very simple optimisations to avoid some obviously unfortunate edge cases (for instance, if one patch is a subset of another then it can be safely eliminated), but these are neither necessary for correctness nor particularly interesting, and we do not discuss them in detail here.

Modifying acquiring instructions to branch to the new code 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 also a risk that the proposed patch 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 branch 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.

## 6.1 Selecting a fix

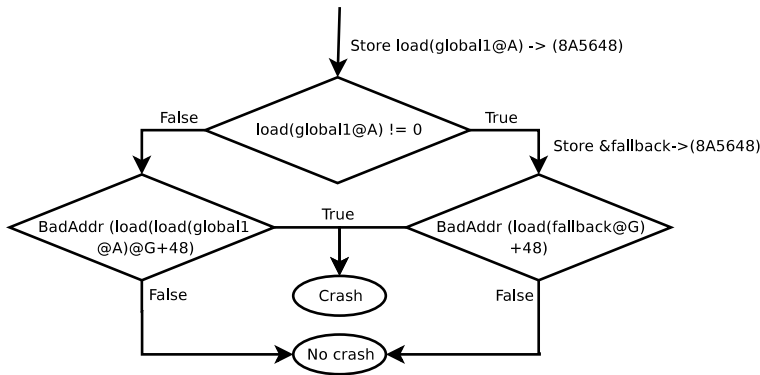
This process might produce multiple possible fixes if multiple state machines are available, and it is then necessary to select an appropriate one to instantiate into a binary patch. Some can be discarded by very simple heuristics (for instance, fixes in which every atomic section is a single memory access can be immediately eliminated), but there will in general be multiple possible fixes 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.

## 6.2 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 specialisation will then simply amount to substituting in constant values for that register. Assume that only one place in the log is considered to be similar to the point for which the machine was derived, and in that in that case `rsp` was `8A5648`. In that case, the only specialisation of the machine will be as shown in figure 3. The only locations accessed when evaluating this machine will be `global1`, `fallback`, and `variable1`. None of these are dynamically allocated, and so the machine is valid throughout the log. The possibly-conflicting stores list

<sup>15</sup>Very non-obvious why this is true.



**Figure 3.** Results of specialising figure 2(a)

will then consist of every dynamic instance of instructions V, W, X, and Y.

We now move on to identifying the unsafe regions in the log. This requires us to evaluate the state machine before and after every possibly-conflicting store. In this case, this 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 instruction sets to be protected are simply:

- All of the loads issued by thread 1 which are mentioned in the state machine. In this case, there are only two; A and G.
- A singleton set containing V.
- A singleton set containing W.
- A singleton set containing X.
- A singleton set containing Y.

We must now construct the code patches for these sets. In this case, none of the sets' control flow graphs will overlap, and so there will be one patch for each set. For the singleton sets, the patch will contain a lock operation, the instruction in the set, an unlock operation and a branch back to the original code. For the larger set, the patch will contain the lock and unlock operations, the instructions from A to G, and a branch back to instruction H in the original code. Note several important properties of this patch:

- When the branch instruction at D is duplicated, it is rewritten such that both the taken and untaken exits point into the new patch, rather than into the original code.
- The instruction which crashed, J, is not included in the 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.
- If the program enters the code shown in figure 1's thread 1 anywhere other than instruction A it will not be affected

by the patch. In particular, there is no danger of the lock being released without first being acquired. However, the program will not be protected against any possible races.

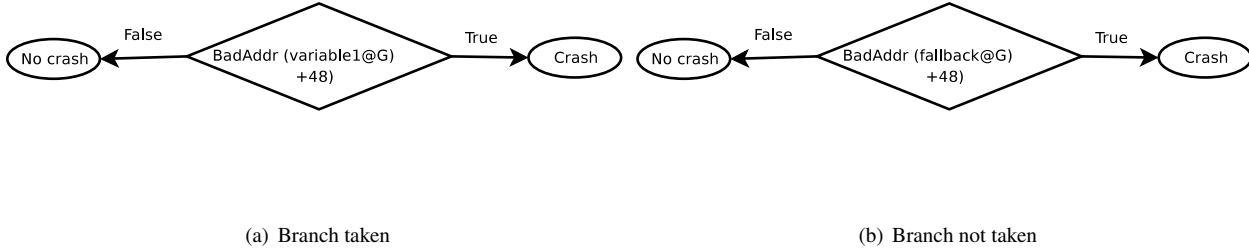
- The fix is correct: it completely eliminates the observed bug.

This is not the only fix which will be suggested, however. 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 specialisations, shown in figure 4, one corresponding to the branch-taken case and one to the branch-not-taken one.

The captured execution shows no writes to fallback, and so the load in figure 4(b) is replaced with a constant and the entire machine is subject to constant folding, producing the trivial machine no-crash. It is therefore discarded. The machine in figure 4(a), however, produces another suggested fix. In this case, the state machine evaluates to crash whenever variable1 is zero i.e. whenever thread 2 is between instructions Y and V. SLI therefore produces alternative sets of instructions to protect:

- The load set will contain just the load at G.
- There will be an unsafe region from Y to V.
- There will be singleton store sets V and Y. Our prototype will recognise that these are redundant because they are covered by the unsafe region and eliminate them before proceeding any further.

The load set will be instantiated into a trivial single-instruction patch, as the store sets were in the previous example. We also construct a patch for the code in figure 1's thread 2 which duplicates instructions Y, Z, and V, plus all of the instructions between Y and Z, and which drops the lock and branches back to instruction W at the end. These patches



**Figure 4.** Results of specialising figure 2(f)

would also fix the bug. Furthermore, assuming that there are only a small number of instructions between Y and Z, they will be preferred by the prioritisation heuristic, because they require fewer critical sections.

It could be argued, however, 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 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).<sup>16</sup>

It is instructive to consider what would have happened if the redundant singleton sections had not been eliminated. In that case, SLI would have constructed patches for each of the singleton sets, as shown in figure 5, before attempting to produce a patch for the unsafe region. The new patch will therefore include duplicates of these previous patches

This requires some explanation. The new critical region starts at instruction Y, and so we rewrite Y to be an acquire-and-branch instruction to a new patch. Y starts off as `acquire_and_jump Y'1`, and so the first instruction of the patch is a similar `acquire_and_jump` instruction modified to branch to a duplicate of Y'1. Y'1 is now duplicated to Y'1', and so are the following instructions. We continue

```
V: acquire_and_jump V'1
W: mov &variable1 -> (global1)
...
X: mov $0 -> (global1)
Y: acquire_and_jump Y'1
U: ...
Z: jmp V
V'1: mov &struct1 -> (variable1)
V'2: release()
V'3: jmp W
Y'1: mov $0 -> (variable1)
Y'2: release()
Y'3: jmp U
```

**Figure 5.** Partial patch based on the state machine in figure 4(a).

duplicating code following the control flow graph until we reach instruction V. This is the final instruction in the range to be protected, and so the new patch should issue that instruction, release the lock, and branch back to the original code. In this case, the instruction to be issued is itself an `acquire_and_jump`, and so SLI optimises slightly by rewriting it to a simple `jmp V'1`.

This illustrates that the binary patching algorithm will perform correctly even in the presence of overlapping critical sections, although it may in some cases produce non-optimal synchronisation patterns.

## 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.

<sup>16</sup> This could do with being a bit earlier.

Name of test	Nature	Number of fixes	Size of logfile	Number of state machines	Total number of state machine states
toctou	Synthetic TOCTOU	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	Loading memory image	Replay engine	Analysis
toctou	0.51 ± 0.01	0.18 ± 0.01	0.19 ± 0.00	0.15 ± 0.00
twovar	0.75 ± 0.01	0.18 ± 0.01	0.42 ± 0.00	0.14 ± 0.00
publish	0.48 ± 0.03	0.18 ± 0.02	0.17 ± 0.00	0.13 ± 0.00
privatize	2.34 ± 0.78	0.18 ± 0.02	1.04 ± 0.01	1.12 ± 0.75
glibc	2.46 ± 0.06	0.19 ± 0.01	1.20 ± 0.05	1.07 ± 0.02
thunderbird	356 ± 1	1.76 ± 0.02	354 ± 1	0.58 ± 0.01

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

```

V: acquire_and_jmp V'1
W: mov &variable1 -> (global1)
...
X: mov $0 -> (global1)
Y: acquire_and_jmp Y''
U: ...
Z: jmp V
V'1: mov &struct1 -> (variable1)
V'2: release()
V'3: jmp W
Y'1: mov $0 -> (variable1)
Y'2: release()
Y'3: jmp U
Y'' : acquire_and_jmp Y'1'
Y'1': mov $0 -> (variable1)
Y'2': release()
Y'3': jmp U''
U'': ...
Z'': jmp V''1
V''1: jmp V'1

```

**Figure 6.** Complete patch based on the state machine in figure 4(a), extending figure 5.

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

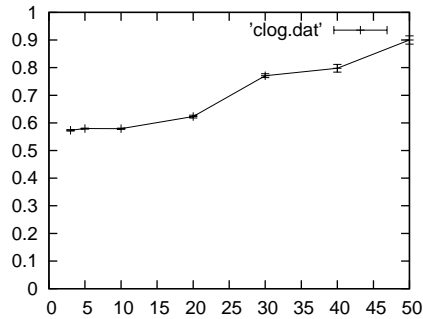
replay engine and the remainder in loading the larger memory image, with the time spent in the analysis phases largely unchanged. This is encouraging. One obvious criticism of this kind of approach is that it might suffer an equivalent of the model checking state explosion problem and hence collapse 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. This 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 eight 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.<sup>17</sup>

### 7.1 Effects of backtracking further

One important parameter to the system is how far to backtrack through the crashed thread, and hence how many state machines to generate, before attempting to derive a fix. For implementation reasons, our prototype will always backtrack to a branch instruction, and for the above tests we limited this backtracking to ten branches. Figure 7 shows the effect of changing this parameter on the time taken by the

<sup>17</sup>Not sure this is all that interesting, or that I've phrased it very well...



**Figure 7.** Time taken by the analysis phases of the thunderbird bug, in seconds, versus the level of backtracking applied to the crashed thread, in dynamic branches. Mean and standard deviation of five runs.

non-replay components of the thunderbird test (the replay components were unchanged). It can be seen that the time taken increases moderately as the distance which we backtrack increases, from 0.57 seconds when backtracking three branches to 0.90 seconds when backtracking fifty. None of our tests depend on more than three levels of backtracking to produce a correct fix (and the thunderbird test requires just one), and so this suggests that SLI should be able to backtrack sufficiently to solve most bugs without requiring excessive resources.<sup>18</sup>

## 7.2 Detailed discussion of test bugs

We now discuss our test bugs in more detail.

`toctou` is a simple two-thread time-of-check, time-of-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:

```
11: mov (global1) -> %rax
12: cmp %rax, (global2)
    jne __assert_fail
```

The state machine for 11 is

```
if load(global1@11) != load(global2@12)
  then crash
  else no-crash
```

<sup>18</sup>Meh.

This produces a single candidate fix, consisting of two critical sections: one protecting instructions 11 and 12 in the second thread, and four single-instruction sections each of which protects a single store in the other thread. This correctly eliminates the bug. Note that, as this bug involves two variables, it would not be eliminated by Kivati [Chew 2010].

`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 pointer, 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, unpublished, 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 unpublishing 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 [Campbell 2006], 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 7.2, 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>19</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[Mery 2007], 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>19</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

20

A number of previous systems have tackled similar problems. Most recently, Kivati[Chew 2010] 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. There are several important differences between their approach and ours:

- SLI is only activated once a bug has been observed, whereas Kivati runs at all times. This means that it is more likely to "fix" perfectly benign races. It also means that the fixes cannot easily be applied without also requiring the Kivati runtime, whereas, once generated, SLI fixes can stand alone without any of the rest of the SLI infrastructure. In other words, while SLI's overheads are higher (due to the use of a DRS), they are only paid once, until the bug is reproduced, whereas Kivati's are paid until a programmer can generate a long-term fix, which may be significantly later.
- Kivati requires access to the program's source code during the initial static analysis phase, whereas SLI requires only the binary.
- SLI can be applied to a wider class of bugs than Kivati, such as the `twovar` example described above.

21

Another approach, taken by systems such as Isolator [Ramalingam 2009] and ToLeRace[Ratanaworabhan 2009], 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. As with Kivati, they do not target specific bugs.<sup>22</sup>

Atom-Aid[Lucia 2009] 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.

<sup>20</sup> This is a bit of a bestiary. Could do with a bit more analysis.

<sup>21</sup> Need an advantage of Kivati here, really.

<sup>22</sup> ...

```

_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 8.** Source code for the glibc test case.

Dimmunix[Julia 2008] solves the closely-related problem of preventing deadlocks in code which uses mutex-like synchronisation. It relies, as SLI does, on having observed the bug at least once, and then synthesises additional synchronisation 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 this requires the use of somewhat different techniques. Gagara[Wang 2008] 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[Lin 2007] and Exterminator[Novark 2007]. 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[Rinard 2004], which waits until the protected program makes an invalid memory reference and then attempts to fix it from the resulting exception handler. DieHard[Berger 2006] 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[Qin 2005] 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. This is arguably safe, although not according to the definition used in this paper, and can cover a wide variety of bugs with little overhead.

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 [Pratikakis 2006] or [Engler 2003], or via model checking, such as [Elmas 2006]. 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. Bugs which can be exhibited are likely to be more important than those which have never been seen, and so this is often a significant limitation.<sup>23</sup>

## 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

<sup>23</sup> ...

global optimum exists; better characterizing the trade-offs involved is likely to suggest useful improvements.

There may also be scope for significantly optimising the DRS used. SLI, at present, collects an enormous amount of information which it then discards, and collecting less information could lead to worthwhile performance 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 approach would be to combine SLI with an invariant inference scheme such as Daikon[Ernst 2007] or DIDUCE[Hangal 2002], 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[Jula 2008], 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.<sup>24</sup>

## References

- [Altekar 2009] G. Altekar and I. Stoica. ODR: Output-Deterministic Replay for Multicore Debugging. In *SOSP*, 2009.
- [Bacon 1994] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345, 1994.
- [Berger 2006] ED Berger and BG Zorn. DieHard: probabilistic memory safety for unsafe languages. In *PLDI*, 2006.
- [Campbell 2006] Neil Campbell. Race condition during unwind code after thread cancellation. Glibc bug number 2644, 2006. URL [http://sourceware.org/bugzilla/show\\_bug.cgi?id=2644](http://sourceware.org/bugzilla/show_bug.cgi?id=2644).
- [Chew 2010] Lee Chew and David Lie. Kivati : Fast Detection and Prevention of Atomicity Violations. In *EuroSys 2010*, pages 307–319, 2010.
- [Elmas 2006] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Precise race detection and efficient model checking using locksets. Technical Report Microsoft Research Technical Report MSR-TR-2005-118, Microsoft Research, 2006.
- [Engler 2003] D Engler and K Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. *ACM SIGOPS Operating Systems Review*, 2003.
- [Ernst 2007] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007. ISSN 01676423.
- [Hangal 2002] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. *International Conference on Software Engineering*, 2002.
- [Jula 2008] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.
- [Lin 2007] 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.
- [Lucia 2009] Brandon Lucia, Joseph Devietti, Luis Ceze, and Karin Strauss. Atom-Aid: Detecting and Surviving Atomicity Violations. *IEEE Micro*, 29(1):73–83, January 2009.
- [Mery 2007] Wayne Mery. crash [@ nsproxyreleaseevent::run()]. Mozilla bug number 39125, 2007. URL [https://bugzilla.mozilla.org/show\\_bug.cgi?id=391259](https://bugzilla.mozilla.org/show_bug.cgi?id=391259).
- [Musuvathi 2008] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.
- [Nethercote 2007] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):100, 2007.
- [Novark 2007] G Novark, ED Berger, and BG Zorn. Exterminator: Automatically correcting memory errors with high probability. *PLDI*, 2007.
- [Pratikakis 2006] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. LOCKSMITH : Context-Sensitive Correlation Analysis for Race Detection. In *PLDI*, 2006.
- [Qin 2005] F Qin, J Tucek, Y Zhou, and J Sundaresan. Rx: Treating bugs as allergies: a safe method to survive software failures. In *SOSP*, 2005.
- [Ramalingam 2009] G. Ramalingam, S.K. Rajamani, V.P. Ranganath, K. Vaswani, and S. Rajamani. Isolator: Dynamically Ensuring Isolation in Concurrent Programs. In *Asplos'09*. ACM, 2009.

<sup>24</sup> Wibble wibble wibble

- [Ratanaworabhan 2009] Paruj Ratanaworabhan, Martin Burtscher, Darko Kirovski, Benjamin Zorn, Rahul Nagpal, and Karthik Pat-tabiraman. Detecting and tolerating asymmetric races. *PPoPP*, 2009.
- [Rinard 2004] M. Rinard, C. Cadar, D. Dumitran, D.M. Roy, T. Leu, and W.S. Beebee Jr. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, 2004.
- [Shavit 1997] N Shavit and D Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.
- [Wang 2008] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multi-threaded programs. In *8th Usenix symposium on Operating System Design and Implementation*, pages 281–294, 2008. URL [http://www.usenix.org/events/osdi08/tech/full\\_papers/wang/wang.html/](http://www.usenix.org/events/osdi08/tech/full_papers/wang/wang.html/).
- [Zamfir 2010] C. Zamfir and G. Candea. Execution Synthesis: A Technique for Automated Software Debugging. In *EuroSys*, 2010.