Practical Lock/Unlock Pairing for Concurrent Programs

Hyoun Kyu Cho
University of Michigan
netforce@umich.edu

Yin Wang
Hewlett-Packard Labs
yin.wang@hp.com

Hongwei Liao
University of Michigan
hwliao@umich.edu

Terence Kelly
Hewlett-Packard Labs
terence.p.kelly@hp.com

Stephane Lafortune
University of Michigan
stephane@umich.edu

Scott Mahlke
University of Michigan
mahlke@umich.edu

Abstract

In the multicore era, developers face increasing pressure to parallelize their programs. However, building correct and efficient concurrent programs is substantially more difficult than building sequential ones. To address the multicore challenge, numerous tools have been developed to assist multithreaded programmers, including static and dynamic bug detectors, automated bug fixers, and optimization tools. Many of these tools rely on or benefit from the precise identification of critical sections, i.e., sections where the thread of execution holds at least one lock. For languages where critical sections are not lexically scoped, e.g., C/C++, static analysis often fails to pair up lock and unlock calls correctly.

In this paper, we propose a practical lock/unlock pairing mechanism that combines static analysis with dynamic instrumentation to identify critical sections in POSIX multithreaded C/C++ programs. Our method first applies a conservative inter-procedural path-sensitive dataflow analysis to pair up all lock and unlock calls. When the static analysis fails, our method makes assumptions about the pairing using common heuristics. These assumptions are checked at runtime using lightweight instrumentation. Our experiments show that only one out of 891 lock/unlock pairs violates our assumptions at runtime and the instrumentation imposes negligible overhead of 3.34% at most, for large open-source server programs. Overall, our mechanism can pair up 98.2% of all locks including 7.1% of them paired speculatively.

CGO '13 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages – Program analysis

General Terms Algorithms, Languages, Verification

Keywords Program analysis, Critical section, Concurrent programs, Static analysis, Dynamic monitoring

1. Introduction

Most performance-aware programmers are experiencing ever growing pressure to parallelize their programs because uniprocessor performance has flattened out and multicore processors promise more cores in each successive hardware generation. Parallel programming, however, remains a daunting task for a variety of reasons. First of all, reasoning about concurrent events and synchronization is inherently challenging for human programmers, who think sequentially. In addition, concurrency bugs such as deadlocks, data races, and atomicity violations require global knowledge of the program. Finally, the nondeterministic execution of parallel programs makes these bugs hard to detect, reproduce, and fix.

There has been much effort to relieve the burden of parallel programming. For example, many automatic concurrency bug detection tools have been developed [9, 19, 28]. Automated bug fixing tools are also available [15, 31]. In addition, researchers are exploring ways to adapt classical compiler optimization techniques for sequential programs to parallel programs [17, 30].

```
if (x)
   lock(L)
...
if (x)
   unlock(L)
```

Figure 1. Infeasible path example

The aforementioned techniques often rely on or benefit from precise lock usage information, which is very difficult to obtain for languages without lexically scoped critical sections. For instance, static bug detection tools using *lockset analysis* [28] must map each statement to the set of locks held by the thread of execution. Infeasible paths such as the one illustrated in Figure 1 are a major source of false positives [9]. Automated bug fix tools often add locks to avoid deadlock [31] or restore atomicity [15], which may introduce new deadlocks if the usage of existing locks is unknown. Finally, it is well known that many compiler optimization techniques cannot be directly applied to concurrent programs [30]. Currently compilers only optimize code sections that do not involve any lock operation, which can be quite conservative [17]. Correct identification of critical sections allows better optimization for concurrent programs.

Numerous static analysis approaches have been developed and many can be adapted to infer critical sections. In general, model checking based tools are precise but do not scale to practical programs [14], while scalable tools using specially designed algorithms are often imprecise [4, 6, 7]. For example, Saturn [7] is a scalable static analysis engine that is both sound and complete with respect to the user-provided analysis script. Writing a script that is sound and complete with respect to the target program, however, is as difficult as writing an analysis engine itself. The lock analysis script bundled with Saturn is neither sound nor complete, most notably because it lacks global alias analysis.

In this paper, we propose a practical lock/unlock pairing mechanism that combines dataflow analysis with dynamic instrumentation. Our interprocedural path-sensitive dataflow analysis is a variant of existing tools [4, 6]. It conservatively identifies lock acquisition and release pairs. When the analysis is uncertain, we use heuristics such as those based on structure types and location proximity to determine the pair. Finally, we instrument the target program with light-weight checking instructions to monitor whether the pairing is correct at run-time. When a violation occurs, feedback information is provided to revise the pairing.

This paper makes several contributions. We present a static lock/unlock pairing analysis algorithm which yields accurate results in most cases. We develop a lightweight dynamic checking mechanism to ensure our analysis is correct. We demonstrate the effectiveness of our lock/unlock pairing mechanism including both static analysis and dynamic checking with real-world multithreaded programs such as OpenLDAP, Apache, and MySQL.

The remainder of the paper is organized as follows. We first present the challenges with motivating examples in Section 2. Next, Section 3 describes how our static lock/unlock pairing analysis works, and Section 4 discusses how to extend the analysis for inter-procedural cases. We explain the dynamic checking mechanism in Section 5. Section 6 presents the experimental results and Section 7 outlines related work. Finally, we summarize the contributions and conclude in Section 8.

2. Background and Motivation

While our approach can help any tool that needs accurate static information about critical sections, we show its effectiveness in the context of deadlock avoidance. In this section, we briefly provide some background on dynamic deadlock avoidance in Gadara [31] and explain what kind of challenges exist for lock/unlock pairing.

Benchmarks	Number of Annotations
OpenLDAP	90
MySQL	71
Apache	19

Table 1. Number of annotations needed to model

2.1 Gadara

Gadara is a tool that enables multithreaded programs to avoid circular-mutex-wait deadlocks at runtime. The basic idea is to intelligently postpone lock acquisition attempts when necessary to ensure that deadlock cannot occur. It proceeds in the following phases. First, Gadara constructs a Petri net model from program source code using compiler techniques. Based on structural analysis of the model, Gadara synthesizes feedback control logic for each structural construct in the model that contributes to a potential deadlock. Finally, it instruments the control logic into the target program.

Given that the model is correct, Gadara automatically synthesizes maximally permissive controllers that delay lock acquisitions only if the program model indicates that deadlock might later result if the lock were granted immediately. Due to lack of accurate information about critical sections, however, Gadara's program analysis or control synthesis could fail. In such cases, Gadara requires programmers to provide annotations. These annotations specify which unlocks match and that no mutex can be held at certain program points. Table 1 shows the number of annotations needed to model the benchmarks for Gadara. Providing annotations can be tedious, difficult, and error-prone even for programmers familiar with both the target program and Gadara.

2.2 Challenges for Lock/Unlock Pairing

This subsection discusses the major challenges that a static lock/unlock pairing analysis should address in order to model programs accurately in the context of deadlock avoidance. We illustrate the challenges via simplified code examples.

2.2.1 Infeasible Path

One of the challenges for static lock/unlock pairing is the ambiguity caused by infeasible paths. The traditional way in which compilers abstract programs' control path is to represent programs with control flow graphs (CFGs). Each vertex in a CFG represents a basic block and each edge corresponds to a branch from one basic block to another. Thus, a sequence of basic blocks connected with edges in a CFG represents a control path. Not all sequences, however, are actually possible control paths in program execution; some are *infeasible paths*.

```
1 :   if (flag)   lock(node->mutex);
2 :   while (condition) {
3 :       ...
4 :       if (flag)   unlock(node->mutex);
5 :       ...
6 :       node = node->next;
7 :       ...
8 :       if (flag)   lock(node->mutex);
9 :       ...
10:   }
11:   if (flag) unlock(node->mutex);
                     (a)
```

```
1 :   void callee(task *ptr) {
2 :       unlock(ptr->mutex);
3 :       ...
4 :       lock(ptr->mutex);
5 :   }
6 :
7 :   void caller(task *ptr) {
8 :       lock(ptr->mutex);
9 :       ...
10:       callee(ptr);
11:       ...
12:       unlock(ptr->mutex);
13:   }
                     (b)
```

```
1 :   lock(parent->mutex);
2 :   ...
3 :   if (condition1) {
4 :       lock(child->mutex);
5 :       ...
6 :       unlock(child->mutex);
7 :   }
8 :   ...
9 :   unlock(parent->mutex);
                     (c)
```

Figure 2. Challenges for lock/unlock pairing

Infeasible paths are a challenge for lock/unlock pairing because there can be cases where a lock is paired up with unlocks for all feasible paths but not paired up for some infeasible paths. This can be seen in the example of Figure 2 (a). If we assume the value of variable `flag` is not changed throughout the snippet, all the branches corresponding to the `if` statements should follow the same direction. However, a naive static analysis would consider all possible combinations of branch directions, if it cannot correlate branch conditions. Another example of infeasible path is infinite loops, since we can assume a finite number of iterations for all reasonable executions. In the same example, if the analysis considers the infinite loop, the lock of line 8 might not be paired up.

This challenge is especially problematic for Gadara, whose models must have the semiflow property [32]. Intuitively, the semiflow property means that a mutex acquired by a thread should be released later in the model. It is not always satisfied, however, if we directly translate the CFG to a Gadara model due to the infeasible path problem as described above. The semiflow property is one of the most important characteristics of Gadara models, and it is the main reason why accurate lock/unlock pairing is important in the context of deadlock avoidance.

2.2.2 Spanning Function Boundaries

Another challenge arises from the fact that locks and unlocks need not reside in the same function. For widely used concurrent programs, it is not rare for locks and unlocks to span multiple levels of call chains. If a lock/unlock pairing analysis operates only within function boundaries it is not possible to pair up such cases.

Figure 2 (b) illustrates such a case. An intra-procedural lock/unlock pairing analysis would conclude that the lock in function `caller()` is paired up inside the function and the lock in function `callee()` is not paired up. However, actually the lock in line 8 is paired up with the unlock in line 2 and the lock in line 4 is paired up with the unlock in line 12, in this calling context.

It gets even more complicated since there can be many calling contexts. Gadara models function calls by substituting into the call site a copy of the callee's Petri net model [31], thus the model of one function can be analyzed differently depending on the calling contexts. Therefore, in order to handle this kind of cases, the lock/unlock pairing analysis should be inter-procedural and context-sensitive.

2.2.3 Pointers

Imperfect pointer analysis imposes another challenge. Mutex variables are usually passed to lock/unlock functions via pointers. Since only locks and unlocks on the same mutex variable pair up, it is important to figure out which lock pointers point to the same location and which pointers do not. As widely known, however, even state-of-the-art pointer analyses cannot provide perfect information. For some cases, they can only conservatively tell that two pointers *may* alias.

Figure 2 (c) illustrates how pointers can cause problems for lock/unlock pairing analysis. Assume that the pointer analysis concludes that the pointer `parent` and `child` may alias, which is the normal case for heap variables. In this case, although it is reasonable for a human programmer to pair up the lock in line 1 and the unlock in line 9, it is not trivial for a static analysis to reach the same conclusion.

In order to model concurrent programs accurately, the lock/unlock pairing analysis should work well under such circumstances with imperfect information about pointers. Furthermore, Gadara conservatively approximates mutex pointers based on types [31]. More specifically, it models the mutexes accessed by pointers, which are enclosed in the same type of structure, as one resource place. The lock/unlock pairing analysis can relieve the impact of this kind of approximation.

3. Static Lock/Unlock Pairing Analysis

In Sections 3 and 4, we discuss how our static analysis pairs up locks and unlocks to cope with the challenges described

```
1 :  int handle_task(task *job) {
2 :      if(job->has_mutex)
3 :          lock(job->mutex);
4 :      if(job->is_special) {
5 :          // Handle special case
6 :          if(job->has_mutex) {
7 :              unlock(job->mutex);
8 :              return result;
9 :          }
10:      }
11:      //Handle normal cases
12:      if(job->has_mutex)
13:          unlock(job->mutex);
14:      return result;
15:  }
```

Figure 3. Simple example of lock/unlock pairing.

in the previous section. Section 3 first covers the detailed steps for intra-procedural cases and we show how to extend it for inter-procedural cases in Section 4.

Our static lock/unlock pairing analysis is carried out in four steps. First, the analyzer extracts an enhanced control flow graph (CFG) from source code and prunes it. This CFG is augmented with information about function calls and branch conditions. It prunes the CFG for computational efficiency, leaving only relevant branches. Second, it maps each lock to a set of corresponding unlocks through dataflow analysis traversing the CFG in a depth first manner while managing lock stack data structures. Third, it calculates Boolean expressions that express the conditions under which each lock and unlock is executed. Finally, using a SAT solver [8], it examines whether all locks are paired up with unlocks on every feasible path. Section 3.1 shows how the analysis works with a simple example, then the rest of the section describes each of these steps in detail.

3.1 Simple Example of Analysis Flow

This section presents the conceptual flow of our lock/unlock pairing analysis with a simple example in Figure 3. In this example, the mutex acquired by the lock in line 3 is always released before the function `handle_task()` returns by either the unlock in line 7 or the unlock in line 13. However, directly translating the CFG of this example to Petri net violates the semiflow property due to infeasible paths as described in Section 2.2. Our analysis rules out the infeasible paths and gives accurate lock/unlock pairing results through the following process.

After pruning the CFG, the corresponding unlock set mapping decides that both the unlock in line 7 and the unlock in line 13 can release the mutex acquired by the lock in line 3. Then, the path condition calculation step determines the Boolean expressions that represent path conditions for each lock and unlock. In this example, path condition $(job \rightarrow has\_mutex \neq 0)$ must be true for the lock to be executed. Similarly, the unlock in line 7 has $(job \rightarrow is\_special \neq 0) \wedge (job \rightarrow has\_mutex \neq 0)$, and the unlock in line 13 has $(job \rightarrow is\_special = 0) \wedge (job \rightarrow has\_mutex \neq 0)$ as their path conditions. The analysis then translates them

```
// Apply analysis to all functions
void traverse_function(fn)
    // Traverse CFG to calculate GEN and KILL sets
    traverse_bb(entry);
    for(each lock discovered)
        corresponding_unlocks[lock]
                                  = GEN[lock] - KILL[lock];
end

// Compute GEN and KILL sets for all locks traversing
// CFG in a depth first manner while managing lock
// stack data structure
void traverse_bb(bb)
    for(each instruction s in bb in order)
        if(s is a lock)
            push s to lock stack;
        else if(s is an unlock)
            top = top element of lock stack;
            add s in GEN[top];
            for(each element e in lock stack, e!=top)
                add s in KILL[e];
            pop from lock stack;
    for(each successor child of bb)
        traverse_bb(child);
end
```

Figure 4. Finding unlock set corresponding to lock

into Boolean expressions by assigning a Boolean variable to each branch condition. In order to encode the branch correlations into the expressions, this assignment process assigns the same Boolean variable to branch conditions that must have the same value. As a result, the Boolean expression of the lock is (x_1), and the unlock in line 7 and the unlock in line 13 receive $(x_2 \wedge x_1)$ and $(\neg x_2 \wedge x_1)$, respectively.

The final step of the analysis is to check whether the lock and the corresponding unlocks pair up for all feasible paths. This can be done by determining whether the statement "if the path condition for the lock is true, then the disjunction of the path conditions for corresponding unlocks is true" is always true or not. In this example, the statement is interpreted as the Boolean expression $(\neg x_1) \vee (x_2 \wedge x_1) \vee (\neg x_2 \wedge x_1)$. In order to verify if it is always true, we apply a SAT solver on the negation of the Boolean expression. If it is unsatisfiable, then the statement is always true, and the lock and the corresponding unlocks are paired. Otherwise, they are not paired up. In this example, the negation of the Boolean expression is unsatisfiable and the lock is paired up with the unlock in line 7 and line 13.

3.2 Mapping Lock to Set of Corresponding Unlocks

Before applying the infeasible path analysis, this step groups corresponding locks and unlocks. More specifically, it maps a set of corresponding unlocks to each lock. We say an unlock corresponds to a lock if it can release the mutex acquired by the lock on any path. Since the mutex acquired by a lock can be released at different program points, we map a set of corresponding unlocks and not just a single unlock. This step is necessary because the same mutex can be acquired and released multiple times.

This analysis algorithm traverses the CFG in a depth first manner while managing a stack of locks for each mutex. The core analysis algorithm is given in Figure 4. Although we only show it for one mutex in this version, the actual analysis simultaneously works on all mutexes.

The underlying idea is to add an unlock to the corresponding unlock set of a lock, if there is a path in which the unlock follows the lock but there is no path in which there is another lock of the same mutex between the lock and unlock. The top element of the lock stack is the most recent lock that the traversal encountered, so it adds the unlock to the `GEN` set of the top element. Other elements in the lock stack are the locks that the traversal met before the last lock along the traversal, so it adds the unlock to the `KILL` sets of them. Ultimately, the corresponding unlocks of each lock are the unlocks that are in the `GEN` set but not in the `KILL` set.

3.3 Path Condition Calculation

This step calculates the Boolean expressions for path conditions that must be true for each lock and unlock to execute. It first calculates path conditions and translates them by assigning a Boolean variable for each branch condition. We define the path condition of a statement as the Boolean combination, i.e., AND($\wedge$), OR($\vee$), NOT($\neg$), of branch conditions, which must be true for the statement to be executed.

The core path condition calculation algorithm is illustrated in Figure 5. With this algorithm, the path condition of a statement is the path condition from the entry basic block to the basic block that the statement belongs to. The underlying idea of this algorithm is that the path condition from the CFG node `'src'` to `'dest'` is the disjunction (OR) of the conditions along the paths which go through `'child'`, for all children of `'src'`. This idea is reflected in line 18.

This algorithm uses caching and post dominator information for computational efficiency. Since there can be an exponential number of paths to the number of basic blocks, the naive recursive algorithm is not feasible for real programs. In order to avoid repetitive computation for the same path, it uses a path condition cache indexed by `(src,dest)` pair. In addition, it uses post dominator (PDOM) information as a shortcut, to simplify the resulting conditions.

After path condition calculation, the analysis translates the path conditions to Boolean expressions by assigning a Boolean variable to each branch condition. To reveal the branch correlations in the Boolean expressions, our analysis assigns the same Boolean variable to the branch conditions that must have the same value. This is possible by using global value numbering (GVN) and hashing them to map to Boolean variables.

3.4 Checking Lock/Unlock Pairing

Using the analysis results of the previous steps, this step finally verifies whether all locks are paired up with the corresponding unlocks on every feasible path. To achieve this goal, we use an open source SAT solver MiniSAT [8]. For each lock, through the previous steps, we have the set of corresponding unlocks and the relevant Boolean expressions for the path conditions of them and the lock. With these analysis results, to verify the statement "the lock is paired up with the corresponding unlocks on every feasible path" is equivalent to checking the proposition "if the Boolean expression for the lock is true, the disjunction of the corresponding unlocks' Boolean expressions is always true." Let L be the Boolean expression for the lock, and U_1, U_2, ..., U_n be the Boolean expressions for the corresponding unlocks. Then our analysis tries to check if the following expression is always true.

$$L \Rightarrow U_1 \vee U_2 \vee \ldots \vee U_n \tag{1}$$

or equivalently

$$\neg L \vee (U_1 \vee U_2 \vee \ldots \vee U_n) \tag{2}$$

Checking whether a Boolean expression is always true or not can be done with a SAT solver. If the expression is always true, its negation always evaluates false, which in turn implies that the negation of the expression is unsatisfiable. Therefore, we can check whether the lock is paired up with the corresponding unlocks by applying a SAT solver to the negation of (2), which is

$$L \wedge \neg U_1 \wedge \neg U_2 \wedge \ldots \wedge \neg U_n \tag{3}$$

```
1 : // Recursively calculates path condition from
2 : // src to dest
3 : condition calculate_path_cond(CFG, src, dest)
4 :     // Consult path condition cache for efficiency
5 :     if (src, dest) is in cache
6 :         return condition from cache;
7 :     // Always reaches dest if it post-dominate src
8 :     if dest PDOM src
9 :         return TRUE;
10:     // Dead end
11:     if src has no successor
12:         return FALSE;
13:     // The control can reach dest following
14:     // each successor
15:     for(each successor c of src)
16:         cond1 = branch condition of branch (src->c);
17:         cond2 = calculate_path_cond(CFG, c, dest);
18:         condition = condition OR (cond1 AND cond2);
19:     put condition in cache with index (src, dest);
20:     return condition;
21: end
```

Figure 5. Calculating path conditions

3.5 CFG Pruning

One of the hurdles that the static lock/unlock pairing analysis must overcome is computational complexity. In real world server programs, the number of basic blocks in a function easily grows to several hundreds. In addition, the Boolean satisfiability problem is well known to be NP complete. For these reasons, we must carefully minimize the number of clauses in the Boolean expressions, in order to make our analysis scale to real programs. We achieve this by pruning the CFG.

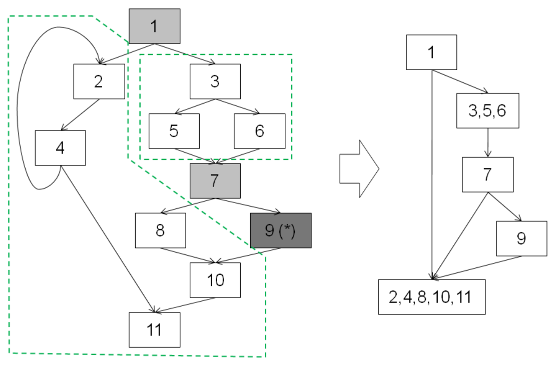

Figure 6. Example of CFG pruning.

Our analysis tool prunes the CFG without losing any relevant information needed for the analysis based on control dependence analysis [10]. Intuitively, CFG node `X` is control dependent on node `Y` if the outgoing edges from `Y` determine whether `X` is executed or not, and control dependencies can be calculated by finding post-dominator frontiers in the CFG. Given this property of control dependence, when we calculate the path condition for a basic block `X`, we must consider the basic blocks on which `X` is control dependent, the basic blocks on which those basic blocks are dependent, and so forth. Therefore, if we calculate the control dependence closure for the basic block, the basic blocks in the closure are the only ones that are relevant for the path condition calculation. We calculate the control dependence closure by iteratively including basic blocks until it converges.

The CFG pruning algorithm works as follows. It starts with the basic blocks of interest as input. Then, it calculates the control dependence closure for them, i.e., the closure relevant basic blocks. Finally, we prune the CFG by maximally merging irrelevant basic blocks that are connected. The CFG pruning algorithm can be easily understood with the example in Figure 6. In this example, basic block 9 is the basic block of interest. It is control dependent on basic block 7; furthermore basic block 7 is control dependent on basic block 1. After calculating the control dependence closure $\{1,7,9\}$, the rest of the basic blocks can be merged if they are connected. The simplified CFG on the right results from pruning. By working on this pruned CFG, the path condition calculation and the resulting Boolean expressions get much simpler.

4. Inter-procedural Analysis

As discussed in Section 2.2.2, many lock/unlock pairs span function boundaries. In order to model concurrent programs for most cases, our lock/unlock pairing analysis must be inter-procedural and context-sensitive. In this section, we describe how to extend the analysis presented in the previous section for inter-procedural cases.

One straightforward way to make the analysis inter-procedural is a top-down approach that performs the analysis on the whole program CFG by conceptually replacing call instruction with the CFG of callee function at every call site, starting from the `main()` function. However, this can cause a computational complexity problem by producing an excessively large CFG to analyze.

Instead of flattening out the CFG for the entire program, we divide the problem into small pieces and perform the analysis on subgraphs in order to limit the analysis time. We first partition the callgraph with a proximity-based heuristic, and analyze the subgraphs in a bottom-up manner. We describe the details of this analysis in the following subsections.

4.1 Proximity-based Callgraph Partitioning

We made two observations while we were trying to manually pair up locks and unlocks across function boundaries. The first observation is that the calling contexts of a lock and paired unlocks differ from a lowest common ancestor in the callgraph with respect to the root node `main()`, in most cases. Suppose that a mutex acquired by a lock with the calling context of $main \Rightarrow f_1 \Rightarrow ... \Rightarrow f_n \Rightarrow f_{l1} \Rightarrow ... \Rightarrow f_{ln}$ is released by an unlock with the calling context of $main \Rightarrow f_1 \Rightarrow ... \Rightarrow f_n \Rightarrow f_{u1} \Rightarrow ... \Rightarrow f_{un}$ on a path, then the other unlocks, if any, that pair up with the lock usually have calling context that shares $main \Rightarrow f_1 \Rightarrow ... \Rightarrow f_n$ and f_n is a lowest common ancestor of them in the callgraph. The second observation is that the depths from locks and unlocks to the lowest common ancestor of the pairing context are relatively small (< 5) for most cases.

Based on the above observations we use a heuristic of proximity-based callgraph partitioning to keep the inter-procedural lock/unlock pairing analysis tractable. The partitioning algorithm works as follows. It starts from functions that have unpaired locks and follows upward the callgraph. It continues until it reaches a node that has the nodes with potentially pairing unlocks as descendants or a predefined depth threshold. Then, it cuts the subgraph from the node as a root. In this way, we can limit the size of Boolean expressions to be small enough to analyze.

4.2 Extending Lock/Unlock Pairing for Inter-procedural Analysis

For inter-procedural lock/unlock pairing, we apply the analysis described in Section 3 on the subgraph partitioned in the previous subsection. The inter-procedural lock/unlock pairing analysis must handle function calls in a different way from the intra-procedural analysis, which just ignores function calls except locks and unlocks. The information about locks and unlocks in callee functions must be considered when the analysis meets a function call. Our analysis takes two different approaches to do so for mapping a lock to the set of corresponding unlocks and for path condition calculation.

Mapping a lock to the set of corresponding unlocks can be modified to be inter-procedural in a relatively straight-

```
1 :  Connection *c = NULL;
2 :  for(; index < tblsize; index++) {
3 :    ...
4 :    if (connections[index].state == C_USED) {
5 :      c = &connections[index];
6 :      lock(&c->c_mutex);
7 :      break;
8 :    }
9 :  }
10:  ...
11:  if (c!=NULL) unlock(&c->c_mutex);
```

Figure 7. Example of uncaught infeasible path.

forward way. It can be considered as conceptually inlining function calls. When it meets a function call it follows the CFG of the callee function. When the function returns it goes back to the caller function's CFG. Other than that, it is identical to the mapping algorithm explained in Section 3.2. It is a simple extension but it is enabled by the proximity-based callgraph partitioning.

On the other hand, path conditions are calculated in a bottom-up manner. In order to calculate the path condition that decides the execution of a lock, it first calculates the lock's path condition in the leaf node function that contains the lock. Then, following the context recognized in the partitioning, it calculates the path condition of the function call in its caller function, and its caller function, and so forth until it reaches the root function of the partition. These conditions get merged with a conjunction operator to finally calculate the context-sensitive path condition for the lock. After it calculates the context-sensitive path conditions for the locks and the unlocks, the remaining steps are identical to the intra-procedural analysis.

5. Dynamic Checking

Our static lock/unlock pairing analysis can be potentially incorrect in some cases due to the assumptions and heuristics it uses. In this section, we discuss these potential sources of incorrect analysis results and explain how our dynamic checking instrumentation can detect them.

One important reason why our analysis might yield potentially incorrect results is pointers as described in Section 2.2.3. Due to the limitations of the default memory dependency analysis, we augment it with generic aggressive refinements. Although the probability is very low, they can result in incorrect analysis results.

The second source of potentially incorrect analysis results is the proximity-based callgraph partitioning heuristic. Although we could not find a real example, it is theoretically possible that one lock has two pairing unlocks whose lowest common ancestors with the lock differ. In that case, our partitioning algorithm can give an incorrect subgraph to analyze and end up with an incorrect analysis result.

Lastly, there are cases where our analysis maps unlocks correctly but cannot guarantee the lock is paired up with unlocks for all feasible paths due to the limitations of our analysis. An example of this case is shown in Figure 7. Our

```
lock_wrapper(mutex, callsite, callstack) {
  lock(mutex);
  LOCK_ID = get_id(callsite, callstack);
  mutex_to_lock_id[mutex] = LOCK_ID;
  ROOT_FID = SEMIFLOW_RESULT[LOCK_ID];
  held_mutex[ROOT_FID].insert(mutex);
}
unlock_wrapper(mutex, callsite, callstack) {
  UNLOCK_ID = get_id(callsite, callstack);
  LOCK_ID = mutex_to_lock_id[mutex];
  mutex_to_lock_id.erase(mutex);
  assert(LOCK_UNLOCK_PAIR[LOCK_ID][UNLOCK_ID]);
  ROOT_FID = SEMIFLOW_RESULT[LOCK_ID];
  held_mutex[ROOT_FID].erase(mutex);
  unlock(mutex);
}
```

Figure 8. Instrumentation wrapper for lock and unlock

analysis can map the unlock in line 11 as the corresponding unlock of the lock in line 6. However, it cannot guarantee the lock is paired up for all feasible paths due to the lack of understanding about program semantics. A human programmer can easily figure out that the variable c is not NULL when the lock in line 6 is executed, thus the lock is paired up with the unlock in line 11 if the value of c is not modified in between. However, it is difficult for a static analysis to understand such program semantics. In this case, our analysis provides the mapping for the modeling as the best effort result. However, this type of best effort analysis result might be incorrect for other cases.

For these reasons, our lock/unlock pairing mechanism needs a way to verify whether all of the analysis results are correct or there exists any violation of the assumptions it made. In order to do that, we need to check two types of conditions. First, the mapping of unlocks to each lock should be checked. If the mutex acquired by a lock is released by an unlock that is not in the corresponding unlock set, it should be detected. Second, the semiflow requirement has to be checked. In other words, whether each lock is paired up with an unlock for all feasible paths or not is to be checked. In the following subsections, we discuss these two types of checking in detail.

5.1 Checking Lock-to-Unlocks Mapping

We instrument all locks and unlocks to check whether the mapping of each lock to corresponding unlocks is correct or not. We first assign a unique ID to each lock and unlock. At runtime, the instrumented code manages a thread-local data structure that keeps the acquiring lock's ID of each mutex. Since the data structure is thread local, it does not need to synchronize with other threads to access the data structure. When an unlock releases the mutex, the instrumented code looks up the acquiring lock's ID of the mutex and checks whether its own ID is in the corresponding unlock set of the lock.

The IDs of locks and unlocks can be simply assigned as a unique number to each calling instruction for intra-procedural cases. However, if they are paired up by the inter-

Benchmarks	LOC	Number of lock	Trivial	DFT	Our Approach				
					Statically Paired	Speculatively Paired	Total Paired	Unpaired	Static Analysis
OpenLDAP	271,546	357	110	267	319	34	353	4	152.7%
MySQL	926,111	499	147	428	463	26	489	10	211.8%
Apache	224,884	19	0	0	17	0	17	2	33.9%
pbzip2	4,011	3	0	1	2	1	3	0	23.4%
pfscan	752	11	8	10	10	1	11	0	50.0%
aget	835	2	2	2	2	0	2	0	43.8%

Table 2. Coverage of static lock/unlock pairing analysis

procedural analysis, we need to manage different IDs for different calling contexts even for the same lock or unlock. This is achieved by managing private call stacks. For the functions that appear in the subgraph analyzed by the inter-procedural analysis, we assign IDs and instrument the entrances and exits to push and pop the ID in the private call stack. This call stack information is concatenated to the IDs of locks and unlocks in order to make it context sensitive.

Figure 8 is the pseudo code for our locks and unlock wrapper functions. It obtains context sensitive IDs of the locks used by the acquisition and release functions at lines 3 and 9, respectively. We verify whether the released lock is in the unlock set corresponding to the acquired lock at line 12.

5.2 Checking Semiflow Property

Another condition that we need to check dynamically is the semiflow property. As described in Section 2.2.1, the semiflow property guarantees that a mutex acquired by a thread will always be released later. With the static analysis we check this property by testing whether locks are paired with unlocks for all feasible paths. If the condition is not satisfied due to incorrect analysis, the dynamic checking should be able to detect it.

We also check this property by instrumenting locks, unlocks, and function exits. For this type of check, the instrumented code maintains the information about held mutexes indexed with the acquiring lock's ID. Again, these IDs are concatenated with call stack information for context-sensitive cases. We instrument the root node functions of the subgraphs partitioned by the proximity-based partitioning to check whether it is holding any lock that should be paired up inside the calling context when it returns. This is done by checking whether the held_mutex[FID] set (kept in line 6 of Figure 8) is empty when the root node function (FID) returns.

6. Experimental Results

We have implemented the lock/unlock pairing mechanism including both the static analysis and checking instrumentation as a pass of the LLVM compiler infrastructure [18]. Our implementation operates on the LLVM intermediate representation and provides both analysis results and instrumented code. For the aggressive refinement of LLVM's memory dependency analysis, we use Gadara's type-based method for mutex pointers and memory profiling for other variables.

All of our experiments were executed on a 2.50GHz Intel Core 2 Quad machine with 8GB of memory running Linux 2.6.32. We evaluate the effectiveness of our lock/unlock pairing with Apache 2.2.11 web server [1], MySQL 5.0.91 database server [24], OpenLDAP 2.4.21 lightweight directory access protocol server [26], pbzip2 1.1.4, pfscan 1.0, and aget 0.4.

6.1 Effectiveness of Static Analysis

Table 2 shows the effectiveness of our static lock/unlock pairing analysis. The third column is the total number of locks and the fourth column is the number of locks trivially paired up in a basic block. We also compare our approach against depth first traversal (DFT) of control flow graph, which is used by previous static lockset-based tools such as RacerX [9]. Statically paired locks mean the number of locks that could be paired up with infeasible path analysis. Speculatively paired locks are the ones that our analysis could successfully map the corresponding unlock sets but could not guarantee pairing for all feasible paths due to the limitation described in Section 5. Thus the sums of the sixth and seventh columns are the numbers of locks that our static analysis could pair up with unlocks. As can be seen in the table, our static analysis works effectively for nearly all of the cases. Overall, trivial pairing fails to handle 70% of locks and DFT fails to handle 20.5% of locks. By contrast, our approach handles all but 1.8% of locks—an eleven-fold improvement compared with DFT.

There are still unpaired locks, although the number of such cases is relatively small. There are three types of causes for these cases. First, there are inherently unpaired locks in the programs. Three unpaired locks of OpenLDAP are from one function, `ldap_new_connection()`, and in this category. When the function is called in certain contexts, these locks are paired up and our analysis can catch those cases. In other contexts, however, they are not paired up and thus our analysis cannot pair them up.

The second category of unpaired locks is due to the type-based memory dependency analysis refinement that we use for mutex pointers. This refinement assumes that two mutex pointers do not alias if the types of wrapper structures enclosing the mutex variables are different. With this assumption our analysis cannot pair up a lock and unlocks if they

```
1 :  class THD {
2 :    struct st_my_thread_var *mysys_var;
3 :    ...
4 :    char* enter_cond(mutex_t* mutex) {
5 :      ...
6 :      mysys_var->current_mutex = mutex;
7 :      ...
8 :    }
9 :    void exit_cond(char* old_msg) {
10:      ...
11:      unlock(mysys_var->current_mutex);
12:      ...
13:    }
14:  };
15:  ...
16:  bool wait_for_relay_log_space(RELAY_LOG_INFO* rli) {
17:    THD *thd = rli->mi->io_thd;
18:    char *save_proc_info;
19:    ...
20:    lock(&rli->log_space_lock);
21:    save_proc_info = thd->enter_cond(&rli->log_space_lock);
22:    ...
23:    thd->exit_cond(save_proc_info);
24:    ...
25:  }
```

Figure 9. Example of unpaired lock due to type mismatch.

have different types. The example in Figure 9 shows how this can cause a problem. In this example, the lock in line 20 and the unlock in line 11 are called on the same mutex, because the call of `enter_cond()` in line 21 saves the mutex in a pointer and passes it to `exit_cond()`. The problem is that the types of wrapping structure for the lock and the unlock are different. The wrapping type is `RELAY_LOG_INFO` for the lock and `st_my_thread_var` for the unlock. The type based memory dependency refinement would consider them not to alias, and consequently our lock/unlock pairing analysis cannot pair them. Among the unpaired locks of MySQL, eight of them are in this category.

The last cause of unpaired unlocks is function pointers. The current implementation of our lock/unlock pairing analysis cannot track the inter-procedural cases in which a function is called via a function pointer, since it uses the call-graph information which only puts edges for direct function calls. One of OpenLDAP's locks, two of MySQL's locks, and two of Apache's locks could not be paired up for this reason.

Static analysis time as a percentage of compilation time is presented in the last column of Table 2. Analyzing MySQL and OpenLDAP takes considerably longer than analyzing other benchmarks because they have more complex control flows and include more lock/unlock function calls. Table 3 presents the number of basic blocks in a function before and after CFG pruning, as described in Section 3.5. Our CFG pruning significantly reduces both average and maximum number of basic blocks, which is essential for the scalability of our static analysis procedure.

Benchmarks	**Before Pruning**		**After Pruning**	
	Average	**Maximum**	**Average**	**Maximum**
OpenLDAP	20.19	818	2.22	80
MySQL	5.88	3513	1.18	112
Apache	12.12	465	1.02	13
pbzip2	6.16	431	1.06	10
pfscan	10.57	48	3.61	33
aget	12.11	35	1.83	16

Table 3. Number of Basic Blocks before/after Pruning

6.2 Runtime Overhead of Dynamic Checking

Figure 10 presents the runtime overheads of the dynamic checking instrumentation. For server programs, it is measured as the comparison of average response time to clients on the same machine. For `pbzip2`, `pfscan`, and `aget`, the execution times are compared. Four parallel clients and worker threads are used for the servers and the other programs, respectively. As can be seen in the graph, our checking instrumentation imposes very small overheads for the programs. The runtime overheads range from 0.5% to 3.4% and the average is 1.6%.

As discussed in Section 6.1, our static analysis yields three types of results: statically paired, speculatively paired, and unpaired. For both statically and speculatively paired locks, our framework instruments the checking mechanism presented in Section 5, whose major overhead comes from the executions of locks and unlocks. Therefore, even if our analysis does not work well so that it yields more speculatively paired locks, the runtime overhead would not be drastically increased. For unpaired locks, the current implementation of our framework falls back to programmer annotations and does not add checking instrumentation. It is possible to add more heuristics to make guesses for unpaired locks, but the dynamic checking overhead would be still roughly proportional to the number of the executions of locks and unlocks even for those cases.

Compared to the native implementation of lock and unlock, our instrumentation slows down a pair of lock and unlock by roughly 18×. Thus, it is possible that our dynamic checking incurs excessive overhead if the target program locks and unlocks too many times without doing much work. However, it is not a common practice to make programs lock and unlock too often, and such programs would already suffer poor performance. Furthermore our current instrumentation implementation is a simple un-optimized use of the C++ STL library, and overheads can be further reduced by optimizing the implementation of instrumented code.

6.3 Assumption Violation

Although the frequency is very low, our static lock/unlock pairing analysis can potentially yield incorrect results due to the assumptions and heuristics it uses as described in Section 5. Once the instrumented dynamic checking detects a problem, the information is fed back to the analyzer and the underlying client system revises the model. While

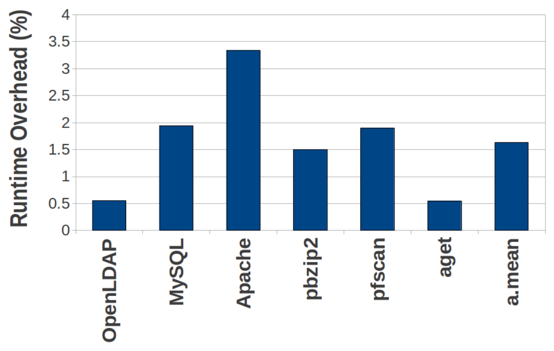

Figure 10. Runtime overheads of dynamic checking.

we perform the experiments on the six programs, only one such case actually occurred for OpenLDAP and the dynamic checking instrumentation detected it.

The code snippet that caused the violation is summarized in Figure 11. The cause of this incorrect analysis result is the type-based memory dependency analysis refinement that we use for mutex pointers. As opposed to the cases where different types for lock and unlock cause a problem, two distinct mutexes having same type is the problem in this case. The programmer's intention is that the lock in line 9 and the unlock in line 11 are called for different mutexes in the same iteration because `ei2` is supposed to point to one of the children of the node pointed by `eip`. Since both pointers have the same wrapper type, however, our mapping algorithm results in mapping the unlock in line 11 to the lock in line 9 and the unlock in line 16 to the lock in line 3. In real execution the mutex acquired by the lock in line 9 can be released by either the unlock in line 11 of the next iteration or the unlock in line 16 after breaking the loop. The lock in line 3 should also be paired up with both unlocks in line 11 and line 16. The instrumented checking code for the lock-to-unlocks mapping check detects this violation and reports the incorrect analysis result.

7. Related Work

In order to better model concurrent programs by pairing up locks and unlocks, we combine static analysis and dynamic checking. Since existing static analysis methods cannot provide a perfect solution to our purpose, we obtain best-effort analysis results with static analysis and check them at runtime to verify whether the results are correct. In this section, we first survey previous work on static analysis and dynamic monitoring techniques, focusing on the application of the lock/unlock pairing problem. Then, we provide possible use case scenarios for our framework.

Static analysis. Existing static techniques applicable to the lock/unlock pairing problem can be largely divided into model checking methods that emphasize precision, and program analysis methods that emphasize scalability.

```
1 :  EntryInfo *eip, *ei2;
2 :  ...
3 :  for (lock(&eip->kids_mutex); eip; ) {
4 :    ...
5 :    // Search children in tree-like in data structure
6 :    ei2 = avl_find(eip->kids, ...);
7 :    ...
8 :    // Lock for next iteration
9 :    lock(&ei2->kids_mutex);
10:    // Unlock current node
11:    unlock(&eip->kids_mutex);
12:    eip = ei2;
13:    ...
14:  }
15:  ...
16:  unlock(&eip->kids_mutex);
```

Figure 11. Incorrectly paired lock due to pointer problem.

Software model checking has a long history. We recommend an excellent survey for the background on this subject [14]. Here we summarize several results relevant to this paper. Classical model checking techniques model systems as *labeled transition systems* and verify properties specified in *temporal logic*. These techniques scale poorly for software verification due to the state explosion problem. Most software model checking tools are execution based and stateless. These tools systematically explore all program paths in hope to find bugs more quickly than stress testing [3, 22].

Abstract model checking scales to real software by mapping program states to an abstract domain [5]. As abstraction may not capture all the information needed to verify a property, when a counter-example is discovered, it is unclear whether it is genuine or spurious due to abstraction. In this case, the abstraction can be refined to filter out spurious examples. Automated program abstraction and refinement are difficult, and the iterative process may not converge. In practice, automated abstract model checking methods are limited to small or special-purpose programs [12, 13].

In the area of static program analysis, many scalable dataflow analysis algorithms have been developed, which can be viewed as model checking with manually defined abstraction [29]. For example, Saturn [7] is a scalable analysis engine that is both sound and complete with respect to the user-provided abstraction, written in its Calypso language. This framework enables the programmer to manually refine and optimize the abstraction for each specific analysis task. Other scalable algorithms use carefully tuned heuristics that can be viewed as predefined abstraction. For example, ESP [6] is a path-sensitive analysis tool that scales to large programs by merging branches that lead to the same analysis state. The analysis is sound but incomplete with respect to this abstraction. Regarding the original program, however, manually defined abstractions are often unsound.

For example, the locking patterns in Figure 1(a) often confuses standard dataflow analysis algorithms integrated in tools designed for higher level applications [9, 31]. Both the locking analysis script bundled in Saturn and the ESP algorithm would identify the branch correlations easily if the code snippet is inside one function. But as both tools use function summaries for scalability, they can fail to infer correctly inter-procedural variations of the pattern if the function summary does not encode enough information. In this case, the analysis result can be sound but incomplete as missing information is often modeled by free variables with arbitrary values. On the other hand, the locking script in Saturn and ESP both ignore global alias, therefore the analysis result is unsound if the branching condition `flag` is modified via a global pointer. Encoding sound and complete global alias information in function summaries is nontrivial [11]. Applications of Saturn often ignore alias analysis too [33].

As of today, we are not aware of any sound and complete program analysis tool that can verify the lock pairing property in large software such as Apache and OpenLDAP. Nevertheless, the analysis techniques in the previous program analysis tools have partially inspired the static analysis part of our work. For instance, we employ the infeasible path analysis similar to the ones used in [4, 6] and we also adopt caching analysis results for computational efficiency as RacerX [9] does.

Dynamic monitoring. Although not directly suitable for our problem, there has been a considerable body of work on monitoring the behavior of programs, especially in the context of profiling and bug detection. The main benefit of dynamic techniques is that they can closely collect information about program execution, which is difficult for static tools to infer.

Such tools as DynamoRIO [2], Pin [20], and Valgrind [25] provide generic instrumentation frameworks for dynamic monitoring. Through the comprehensive API of Pin and DynamoRIO, users can write their own monitoring client fitting their purpose, and Valgrind is widely used to detect memory bugs. In spite of the many optimizations they exploit such as code cache, branch linking, and trace building, however, they can impose a substantial amount of runtime overheads depending on what kind of code should be instrumented.

There are also dynamic monitoring techniques customized for specific purposes. LiteRace [21] and ReEnact [27] track concurrent programs' memory accesses to detect data races. AVIO [19] and AtomTracker [23] aim for atomicity violations. Our framework shares the idea of reducing runtime overheads by customizing the type of tracking information with these tools. As opposed to these tools, however, our framework performs most of its analysis offline and uses dynamic checking only for confirmation.

Use cases. As mentioned in Section 1, our framework can benefit static bug detection tools and automated bug fix tools by providing more accurate information about critical sections. For instance, static bug detection tools using lockset analysis suffer false positives due to infeasible paths. RacerX [9] uses many heuristics and error ranking to mitigate the impact of such false positives. Our framework would help them prune invalid locksets and thus reduce the false positives. On the other hand, automated bug fix tools [15, 16] often add synchronizations to restore atomicity or order constraints, and they may introduce new deadlock if the usage of existing locks is unknown. AFix [15] sets timeout for the new synchronizations to avoid introducing deadlocks. Our framework can help them eliminate the timeouts and the potential chances of missing bugs.

Our framework can also promote compiler optimizations for concurrent programs. Currently compilers only optimize code sections that do not involve any lock operations, limiting the efficiency of the generated code. Joisha et al. [17] suggest extending the scope of optimizations beyond the synchronization-free regions by using procedural concurrency graph (PCG). With accurate lock/unlock pairing, they can further refine PCGs by reflecting the concurrency limited via mutexes. Consequently, this can provide more optimization opportunities.

8. Conclusion

We have proposed a practical lock/unlock pairing mechanism that combines an inter-procedural analysis and dynamic checking for better modeling of critical sections in POSIX multithreaded C/C++ programs. We have demonstrated the effectiveness of our mechanism through experiments on six benchmarks including three large and complex server programs. Compared with depth-first traversal, our method reduces by $11\times$ the number of statically unpaired locks. CFG pruning keeps problem size small so that compile time is low, and dynamic checking compensates for imperfections in our static analysis with modest overhead (at most 3.3%).

Acknowledgments

This work is supported in part by an Open Innovation Award from Hewlett-Packard Laboratories and by the National Science Foundation under grants CCF-0819882 and CNS-0964478. The authors would like to thank our shepherd (James Tuck) and the anonymous reviewers for their time, suggestions, and valuable feedback.

References

[1] apache. The Apache HTTP Server Project, 2012. `http://httpd.apache.org`.

[2] D. L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institude of Technology, Sept. 2004.

[3] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.

[4] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proc. of*

the '07 Conference on Programming Language Design and Implementation, pages 480–491, 2007.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[6] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. of the '02 Conference on Programming Language Design and Implementation*, pages 57–68, 2002.

[7] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *Proc. of the '08 Conference on Programming Language Design and Implementation*, pages 270–280, 2008.

[8] K. Een and N. Sorensson. An extensible sat-solver [ver 1.2], 2003.

[9] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, pages 237–252, 2003.

[10] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[11] B. Hackett. *Type Safety in the Linux Kernel*. PhD thesis, Stanford University, Apr. 2011.

[12] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Conference Record of the 31st Annual ACM Symposium on Principles of Programming Languages*, pages 232–244, 2004.

[13] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar. Efficient sat-based bounded model checking for software verification. *Theor. Comput. Sci.*, 404(3):256–274, 2008.

[14] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.

[15] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.

[16] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, pages 221–246, 2012.

[17] P. G. Joisha, R. S. Schreiber, P. Banerjee, H.-J. Boehm, and D. R. Chakrabarti. A technique for the effective and automatic reuse of classical compiler optimizations on multithreaded code. In *POPL*, 2011.

[18] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[19] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of the '05 Conference on Programming Language Design and Implementation*, pages 190–200, 2005. ISBN 1-59593-056-6.

[21] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: Effective sampling for lightweight data-race detection. In *Proc. of the '09 Conference on Programming Language Design and Implementation*, pages 134–143, 2009. ISBN 978-1-60558-392-1.

[22] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proc. of the '07 Conference on Programming Language Design and Implementation*, pages 446–455, 2007.

[23] A. Muzahid, N. Otsuki, and J. Torrellas. Atomtracker: A comprehensive approach to atomic region inference and violation detection. In *Proc. of the 43rd Annual International Symposium on Microarchitecture*, pages 287–297, 2010.

[24] mysql. MySQL: The World's Most Popular Open Source Database, 2012. `http://www.mysql.com`.

[25] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. of the '07 Conference on Programming Language Design and Implementation*, pages 89–100, 2007. ISBN 978-1-59593-633-2.

[26] openldap. OpenLDAP: Community Developed LDAP Software, 2012. `http://www.openldap.org`.

[27] M. Prvulovic and J. Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 110–121, 2003.

[28] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 15(4):391–411, Nov. 1997.

[29] D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Conference Record of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 38–48, 1998.

[30] J. Sevcík. Safe optimisations for shared-memory concurrent programs. In *PLDI*, 2011.

[31] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 281–294, 2008.

[32] Y. Wang, H. Liao, S. Reveliotis, T. Kelly, S. Mahlke, and S. Lafortune. Gadara nets: Modeling and analyzing lock allocation for deadlock avoidance in multithreaded software. In *Proc. of the 48th IEEE Conference on Decision and Control*, pages 4971–4976, 2009.

[33] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from runtime logs. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 143–154, 2010.