In Proc. of the 15th International Conference On Principles Of Distributed Systems (OPODIS),
Toulouse, France, 2011.
Revised version.

Optimal Instrumentation of Data-flow in Concurrent Data Structures

Samaneh Navabpour[1], Borzoo Bonakdarpour[2], and Sebastian Fischmeister[1]

[1] Department of Electrical and Computer Engineering
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada, N2L 3G1
Email: {snavabpo,sfischme}@uwaterloo.ca

[2] School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada, N2L 3G1
Email: borzoo@cs.uwaterloo.ca

Abstract. In this paper, we propose an automated technique for optimal instrumentation of *multi-threaded* programs for debugging and testing of concurrent data structures. We define a notion of *observability* that enables debuggers to trace back and locate errors through data-flow instrumentation. Observability in a concurrent program enables a debugger to extract the value of a set of *desired variables* through instrumenting another (possibly smaller) set of variables. We formulate an optimization problem that aims at minimizing the size of the latter set. In order to cope with the exponential complexity of the problem, we present a SAT-based solution. Our approach is fully implemented and experimental results on popular concurrent data structures (e.g., linked lists and red-black trees) show significant performance improvement in optimally-instrumented programs using our method as compared to adhoc over-instrumented programs.

Keywords: Debugging, testing, multi-thread, concurrent programs, instrumentation, optimization.

1 Introduction

Debugging is a systematic process of finding and reducing the number of defects in a computer program. Program debugging is a continual de facto step in the software development process and often requires significant human and computing resources. The debugging process ranges over a variety of techniques such as traditional or breakpoint-style debuggers, event monitoring systems, and static analysis for which different aspects and tools are employed. Examples include interactive debugging, control- and data-flow analysis, log files, memory dumps, and profiling. Incorporating these techniques mostly requires adding extra instructions to the program under scrutiny, called *instrumentation*.

The main problems associated with instrumenting (and, hence, debugging) programs are increased complexity, the *probe effect*, and non-repeatability. The probe effect refers to the problem that any attempt to observe the behavior of a system may change its behavior. Furthermore, such problems are amplified significantly in the context of concurrent programs. This is due to the fact that instrumenting these programs complicates their inherent non-deterministic nature, causing different executions for the same data and more unpredictable context switches.

Moreover, although there have been significant advances in the multi-core technology, it is currently unclear to what extent software products can be multi-threaded to take advantage of these new chips. Thus, in the presence of challenges in developing, testing, and maintaining scalable multi-threaded programs, having access to effective debugging tools for concurrent programs is highly beneficial. This benefit is even more crucial in the context of concurrent embedded safety-critical applications, where deviation of a mutated program from its specification may result in catastrophic consequences.

In [12], we proposed a notion of *observability* as the ability to test various features of a *sequential program* by observing the program's outcome to check if it conforms to the software's specification. The traditional methods for achieving observability incorporate ad-hoc instrumentation techniques [20, 21, 18] that cause the observed outcome of the software to be produced by a mutated program which can violate its correctness. Our approach to contain such probe effects in [12] is to introduce minimal instrumentation to sequential programs under debugging.

With this motivation, in this paper, we extend the concept of observability to the context of *concurrent programs*. Our contributions in this paper are as follows:

- We formally define the notion of observability for concurrent programs. This notion has a different nature as compared to sequential programs due to the existence of shared variables and interleaving scenarios. Roughly speaking, observability in a concurrent program enables a debugger to extract the value of a set of *desired variables* through instrumenting another set of variables. We call the latter the set of *naturally observable* variables.
- We formulate an optimization problem to tackle under and over-instrumentation defects. In other words, given a multi-threaded program and a set of desired variables, our goal is to identify the minimum set of naturally observable variables (that will be instrumented for debugging) through which one can extract the value of all desired variables.
- Since the complexity of our optimization problem is exponential, we encode the problem as a propositional satisfiability problem to leverage powerful SAT-solvers to solve our problem.
- Our method is fully implemented in a tool chain. We use LLVM [7] and the method presented in [11] to extract program data-flow dependencies. This is

achieved by implementing a new pass over LLVM that takes the source code and a set of desired variables as input and generates the full set of data-flow dependencies as output. Using the extracted dependencies, we automatically generate a SAT model which is the input to the SMT-solver Yices [1]. The solution to the SAT model is the set of variables that need to be instrumented (the naturally observable variables).
- We conduct experiments on two popular concurrent data structures: *linked lists* and *red-black trees*. We consider different implementations of these data structures with respect to different liveness criteria and synchronization primitives, such as lock-based, software transactional memory (STM), lock-free, and obstruction-free implementations. Our experiments show that our method effectively optimizes instrumentation instructions, resulting in significant performance improvement (in some cases up to 50 times), as compared to ad-hoc over-instrumented programs.

Organization. The rest of the paper is organized as follows. In Section 2, we present the preliminary concepts on control-flow graphs and data-flow dependencies. Section 3 is dedicated to define our notion of observability in concurrent programs and the statement of our optimization problem to reduce instrumentation. We describe our approach to solve the optimization problem in Section 4. Section 5 presents the results of our experiments. Related work is discussed in Section 6. Finally, we make concluding remarks and discuss future work in Section 7.

2 Preliminaries

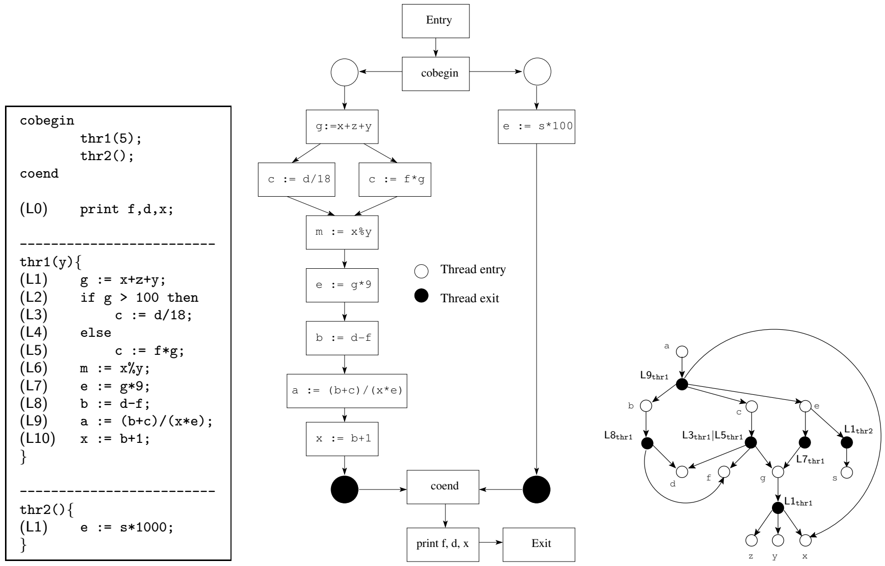
In this section, we present the preliminary concepts. In particular, in order to capture data-flow dependencies caused by interleaving executions in concurrent programs, we first define the notion of *Concurrent Control Flow Graphs* (CCFG) [8] in Subsection 2.1. Then, in Subsection 2.2, we elaborate on the notion of data dependencies as a means to trace the data-flow.

2.1 Concurrent Control-flow Graphs

Intuitively, a *concurrent control-flow graph* is a control-flow graph which incorporates constructs to model concurrency. We use a cobegin/coend construct to express concurrent execution of threads. The cobegin/coend construct contains two or more blocks of code, which may in turn contain other cobegin/coend constructs.

Definition 1. *A* Concurrent Control Flow Graph *(CCFG) is a directed graph* $G = \langle N, A, n^0 \rangle$ *such that:*

- *N is the set of nodes in G. Each node is a* basic block. *Without loss of generality, we assume that each basic block contains only one instruction.*



```
cobegin
        thr1(5);
        thr2();
coend

(L0)    print f,d,x;

-------------------------
thr1(y){
(L1)    g := x+z+y;
(L2)    if g > 100 then
(L3)         c := d/18;
(L4)    else
(L5)         c := f*g;
(L6)    m := x%y;
(L7)    e := g*9;
(L8)    b := d-f;
(L9)    a := (b+c)/(x*e);
(L10)   x := b+1;
}

-------------------------
thr2(){
(L1)    e := s*1000;
}
```

(a) Two threads running concurrently.

(b) CCFG

(c) Observability graph.

Fig. 1. A C program and its concurrent control-flow graph.

- *n^0 is the initial node with indegree 0, which represents the initial basic block of G.*
- *A is a set of arcs (n, m), where $n, m \in N$. An arc (n, m) exists in A, iff the execution of basic block n in a thread thr encoded in G immediately leads to the execution of basic block m in thread thr.* □

For the sake of clarity, we distinguish different types of basic blocks in a CCFG. These types include *Entry*, *Exit*, *Cobegin*, *Coend*, *Compute*, *ThreadEntry*, and *ThreadExit*. Arcs that involve any type of basic block except for *Compute* are trivially added to a CCFG. Arcs which involve *Compute* basic blocks are specified in Definition 1. Figure 1(a) shows an example, where the program consists of two threads `thr1` and `thr2` running concurrently. We consider variables with the same name in different threads as shared variables. For instance, variable `e` is a shared variable. The CCFG of the program in Figure 1(a) is shown in Figure 1(b).

Notation: Let $\pi = n_1, n_2, \ldots, n_k$ be a sequence of nodes of a CCFG G and *thr* be a thread encoded in G. By *thr*(π), we denote the sequence of nodes where the nodes of all threads except for *thr* are removed from π.

Definition 2. *Let $G = \langle N, A, n^0 \rangle$ be a CCFG. An* execution path *π of G between two nodes m and m' in N is a sequence $n_1, n_2, \ldots, n_k$, such that:*

1. $n_1 = m$ and $n_k = m'$,

2. *for all threads thr of G, we have: (i) thr(π) = $x_1, x_2, \ldots, x_j$ is a total order, and (ii) for all i, where $1 \leq i \leq j-1$, $(x_i, x_{i+1}) \in A$, and*
3. *the causal relation between nodes in π is a partial order.* □

Intuitively, in Definition 2, Condition 2 requires that the order of basic blocks of the same thread in π must follow the isolated sequential execution of that thread. Moreover, Condition 3 expresses that the order of basic blocks of a set of concurrent threads in π must be in an ordered interleaving fashion.

Notation: To distinguish instructions of different threads, we denote an instruction of a thread thr by $\mathsf{line\_number}_{\mathtt{thr}}$. When the thread name is irrelevant or clear from the context, we omit it.

For example, an execution path of the CCFG in Figure 1(b) is $\mathsf{L1}_{\mathtt{thr1}}$, $\mathsf{L2}_{\mathtt{thr1}}$, $\mathsf{L3}_{\mathtt{thr1}}$, $\mathsf{L6}_{\mathtt{thr1}}$, $\mathsf{L1}_{\mathtt{thr2}}$, $\mathsf{L7}_{\mathtt{thr1}}$, $\mathsf{L8}_{\mathtt{thr1}}$, $\mathsf{L9}_{\mathtt{thr1}}$, $\mathsf{L10}_{\mathtt{thr1}}$, L0.

2.2 Data-flow Dependencies in Concurrent Programs

Given the value of a set of variables, one normally has to trace back the program's data-flow to reach the source of errors.

Definition 3. *We say that the value of a variable v* depends *on the value of variable v' iff $v = F(v', V)$, where F is an arbitrary function and V is the remaining set of F's arguments called* parameters *[2].* □

In a source code, any instruction that updates the value of a variable creates a dependency. For instance, in function thr1, in Figure 1(a), instruction L9 creates a dependency between variable a and variables b, x, e, and c. We represent a dependency by a tuple $\langle v, n, v' \rangle$, where n is an instruction (i.e., a node in the corresponding CCFG) and v is a variable whose value can be extracted from variable v' via instruction n. In this case, we say that instruction n *defines* the value of variable v. In our example, we have $\langle \mathsf{a}, \mathsf{L9}_{\mathtt{thr1}}, \mathsf{b} \rangle$ and $\langle \mathsf{a}, \mathsf{L9}_{\mathtt{thr1}}, \mathsf{c} \rangle$.

Some data dependencies are resolved based upon runtime circumstances, e.g., in conditional statements. For instance, in thr1, the value of variable c at line L9 is defined by either L3 or L5. As a result, it is incorrect to have the dependency $\langle \mathsf{c}, \mathsf{L3}_{\mathtt{thr1}}, \mathsf{d} \rangle$, as we cannot determine whether this dependency indeed holds at run time. Thus, in order to compute data dependencies statically, we consider a conservative set of instructions that can define the value of a variable at run time. Hence, we require a representation that conveys that c depends on variable d or on variables f and g.

In order to resolve this issue, we combine dependencies that define the same variable into one dependency as follows. When a variable v is defined via instructions n_1 and n_2, where $\langle v, n_1, v_1 \rangle$ and $\langle v, n_2, v_2 \rangle$ are in two separate and mutually exclusive conditional branches, we combine instructions n_1 and n_2 into one instruction $n' = n_1 \mid n_2$. Hence, we replace dependencies $\langle v, n_1, v_1 \rangle$ and $\langle v, n_2, v_2 \rangle$ with $\langle v, n', v_1 \rangle$ and $\langle v, n', v_2 \rangle$. This implies that v depends on v_1 when

```
cobegin
       thr2();
coend
cobegin
       thr1(5);
coend
```

Fig. 2. Two threads running sequentially.

one of the two instructions n_1 or n_2 execute. The same concept applies for the dependency between v and v_2. In our example, for dependency $\langle \mathtt{c}, \mathtt{L3_{thr1}}, \mathtt{d} \rangle$, we have $\langle \mathtt{c}, \mathtt{L3_{thr1}} \mid \mathtt{L5_{thr1}}, \mathtt{d} \rangle$. In other words, `c` may depend on `d` if $\mathtt{L3_{thr1}}$ or $\mathtt{L5_{thr1}}$ execute. In this case, parameters of $\mathtt{L3_{thr1}} \mid \mathtt{L5_{thr1}}$ is the set $\{\mathtt{d}, \mathtt{g}, \mathtt{f}\}$.

During program execution, the value of a variable depends on variables used/defined by a sequence of instructions leading to the instruction that defines the variable. For instance, the value of variable `a` defined by $\mathtt{L9_{thr1}}$ indirectly depends on: (1) variables `d` and `f` used by instruction $\mathtt{L8_{thr1}}$ which defines `b`, (2) variables `d`, `f`, and `g` used by $\mathtt{L3_{thr1}} \mid \mathtt{L5_{thr1}}$ which define `c`, (3) variable `g` used by instruction $\mathtt{L7_{thr1}}$ which defines `e`, and (4) variables `x`, `z`, and `y` used by $\mathtt{L1_{thr1}}$ which defines `g`.
In order to capture the effect of execution paths on the value of a variable, we introduce the notion of dependency chains.

Definition 4. *Let G be a CCGF and v be a variable. A* dependency chain *for v is a sequence $\sigma = \langle v_1, n_1, v_2 \rangle \langle v_2, n_2, v_3 \rangle \dots \langle v_{k-1}, n_{k-1}, v_k \rangle$ of dependencies where:*

- *$v_1 = v$, and*
- *the sequence $n_k, n_{k-1}, \dots, n_2, n_1$ is an execution path of G between basic blocks n_1 and n_k [2].* □

Clearly, determining data dependencies relies on the structure of the source code. For example, in the source code of Figure 1(a) dependency chains $\sigma_1 = \langle \mathtt{a}, \mathtt{L9_{thr1}}, \mathtt{e} \rangle \langle \mathtt{e}, \mathtt{L1_{thr2}}, \mathtt{s} \rangle$ and $\sigma_2 = \langle \mathtt{a}, \mathtt{L9_{thr1}}, \mathtt{e} \rangle \langle \mathtt{e}, \mathtt{L7_{thr1}}, \mathtt{g} \rangle$ are both possible. This is caused by the fact that threads `thr1` and `thr2` run concurrently. However, if we change the structure as shown in Figure 2, then dependency σ_1 is invalid while σ_2 is still valid. Dependency σ_1 is invalid because in all execution paths, $\mathtt{L7_{thr1}}$ executes after $\mathtt{L1_{thr2}}$, hence the value of `a` never depends on the value of `e` defined by $\mathtt{L1_{thr2}}$.

Typically, one does not need to enumerate all dependency chains of a variable in order to extract the value of that variable. In other words, we only need to identify a subset of all dependency chains that is *maximal*.

Definition 5. *Let $\mathcal{S}_v$ be a set of dependency chains for a variable v. We say that $\mathcal{S}_v$ is a* maximal dependency set *for v iff for all dependency chains $\sigma \in \mathcal{S}_v$, there does not exist a dependency chain σ', where $\sigma\sigma' \in \mathcal{S}_v$.* □

For example, $\mathcal{S}_{\mathtt{a}} = \{\langle \mathtt{a}, \mathtt{L9}_{\mathtt{thr1}}, \mathtt{b}\rangle, \langle \mathtt{a}, \mathtt{L9}_{\mathtt{thr1}}, \mathtt{b}\rangle\langle \mathtt{b}, \mathtt{L8}_{\mathtt{thr1}}, \mathtt{d}\rangle, \langle \mathtt{a}, \mathtt{L9}_{\mathtt{thr1}}, \mathtt{x}\rangle, \langle \mathtt{a}, \mathtt{L9}_{\mathtt{thr1}}, \mathtt{e}\rangle\}$ is *not* a maximal dependency chain of a, as the dependency chain $\langle \mathtt{a}, \mathtt{L9}_{\mathtt{thr1}}, \mathtt{b}\rangle$ is a prefix of the dependency chain $\langle \mathtt{a}, \mathtt{L9}_{\mathtt{thr1}}, \mathtt{b}\rangle\langle \mathtt{b}, \mathtt{L8}_{\mathtt{thr1}}, \mathtt{d}\rangle$. To convert $\mathcal{S}_{\mathtt{a}}$ into a maximal dependency set, we must either remove $\langle \mathtt{a}, \mathtt{L9}_{\mathtt{thr1}}, \mathtt{b}\rangle$ or $\langle \mathtt{a}, \mathtt{L9}_{\mathtt{thr1}}, \mathtt{b}\rangle\langle \mathtt{b}, \mathtt{L8}_{\mathtt{thr1}}, \mathtt{d}\rangle$ from $\mathcal{S}_{\mathtt{a}}$.

Finally, we introduce the notion of a program *slice* [11]. Intuitively, a program slice for a variable v is a maximal dependency chain set that covers all dependency chains that start with v.

Definition 6. *Let $\mathcal{S}_v$ be a maximal set of dependency chains for a variable v. We say that $\mathcal{S}_v$ is the* program slice *for v iff there does not exist a dependency chain σ for v, such that $\sigma\sigma'$ is not in $\mathcal{S}_v$ for some σ'.* □

For example, the program slice for variable a includes all dependency chains for a built from instructions $\mathtt{L9}_{\mathtt{thr1}}$, $\mathtt{L8}_{\mathtt{thr1}}$, $\mathtt{L7}_{\mathtt{thr1}}$, $\mathtt{L5}_{\mathtt{thr1}}$, $\mathtt{L3}_{\mathtt{thr1}}$, $\mathtt{L1}_{\mathtt{thr1}}$, and $\mathtt{L1}_{\mathtt{thr2}}$.

3 Observability in Concurrent Programs

In this section, we establish the notion of *observability* in concurrent systems as a means for debugging concurrent programs. In the context of program debugging, outputs often provide us with insufficient information to locate bugs that lead to erroneous behaviour. This is due to the fact that output values depend on internal variables, execution paths, and interleavings that are potentially not unique for each buggy scenario. Normally, we can only observe input and output variables.

Definition 7. *A value of a variable v is* naturally observable *iff the value is an output or input of the system.* □

For example, in Figure 1(a), variables d,f,x, and y are naturally observable since they are outputs and input, respectively. Note that *instrumented* variables are considered as program outputs and, hence, naturally observable variables as well.

We now use the notion of program slices in Definition 6 to define what it means for a variable to be observable. Intuitively, a variable is observable if there exists a *sub-slice* of the variable, where each dependency chain in the sub-slice ends with a variable that is naturally observable.

Definition 8. *A* sub-slice *$\mathcal{S}'$ of a slice $\mathcal{S}$ is a set of dependency chains, where each chain in $\mathcal{S}'$ is a prefix of a chain in $\mathcal{S}$.* □

To motivate the idea of observability, notice that given the value of naturally observable variables d, f, x, and y in our running example, we cannot extract the value of a using a's program slice. To extract the values of a, we require the values of b, c, e, and x. Variable a does not have a dependency with the value of x printed on line L0, since a uses the value of x before it is redefined at line

`L10` and printed on `L0`. On the other hand, `a` has dependencies with `d` and `f` via variables `b` and `c` at lines `L8`, `L5`, and `L3`. Moreover, `d` and `f` can only lead to extracting the value of `b` and `c` at lines `L8` and `L3`. Hence, the value of `c` is still unknown at `L9` since we can not predict if `c` will be defined via line `L3` or `L5` at runtime. As a result, we cannot guarantee determining the value of `c` at line `L9` without having the value of `g`. In addition, based on `thr1`'s code, it is clear that `d`, `f`, and `y` can not be used to extract values of `e` and `x` at line `L9`. Hence, the values of `x`, `e`, and `c` are still required for extracting the value of `a`. Therefore, no sub-slice of `a` provides enough information to extract `a`'s value.

We now formally define the constraints that need to be satisfied to extract the value of a variable in concurrent programs.

Definition 9. *A sub-slice $\mathcal{S}$ is* complete *iff*

1. *for each dependency $\langle v, n, v' \rangle$ in a dependency chain of $\mathcal{S}$, there exists a dependency $\langle v, n, v'' \rangle$ in at least one dependency chain of $\mathcal{S}$, for each variable v'' in n's parameter set.*
2. *for every dependency prefix $\langle v, n_{thr}, sv \rangle \langle sv, m_{thr}, v' \rangle$ in $\mathcal{S}$, if there exists another thread thr' running concurrently with thr that contains an instruction of the form:*

   $$(\mathsf{L}) \quad sv := F(v'');$$

   i.e., sv is a shared variable also defined by thr', then there must exist a dependency chain $\sigma' \in \mathcal{S}$ that contains $\langle sv, \mathsf{L}_{\mathtt{thr'}}, v'' \rangle$. □

For example, the sub-slice $\mathcal{S}_{\mathsf{a}} = \{\langle \mathsf{a}, \mathsf{L9}_{\mathtt{thr1}}, \mathsf{b} \rangle, \langle \mathsf{a}, \mathsf{L9}_{\mathtt{thr1}}, \mathtt{x} \rangle, \langle \mathsf{a}, \mathsf{L9}_{\mathtt{thr1}}, \mathsf{e} \rangle\}$ is not complete, since it violates both Conditions 1 and 2 of Definition 9. To satisfy Condition 1, we add dependency $\langle \mathsf{a}, \mathsf{L9}_{\mathtt{thr1}}, \mathsf{c} \rangle$ to $\mathcal{S}_{\mathsf{a}}$ and to satisfy Condition 2, we add dependency $\langle \mathsf{a}, \mathsf{L1}_{\mathtt{thr2}}, \mathsf{e} \rangle$ to $\mathcal{S}_{\mathsf{a}}$, as `e` is defined by `thr2` which runs concurrently with `thr1`.

Definition 10. *A variable v is* observable *iff there exists a complete sub-slice $\mathcal{S}_v$ where:*

- *every dependency chain $\sigma \in \mathcal{S}_v$ ends in a naturally observable variable, and*
- *every shared variable sv in $\mathcal{S}$ is naturally observable.*

We call $\mathcal{S}_v$ an observable sub-slice. □

To clarify the need for Condition 2 in Definition 10, consider Figure 1(a). In order to observe the value of variable `a`, we require the value of variable `e`. Variable `e` is updated by both lines $\mathsf{L7}_{\mathtt{thr1}}$ and $\mathsf{L1}_{\mathtt{thr2}}$. Since `thr1` and `thr2` run concurrently, we can not predict which of the two lines $\mathsf{L7}_{\mathtt{thr1}}$ or $\mathsf{L1}_{\mathtt{thr2}}$ is last to update `e`. Hence, we need to explicitly extract the time at which both lines execute, so one can determine which instruction defines the value of `e` used at line $\mathsf{L9}_{\mathtt{thr1}}$. As a result, we need to explicitly make `e` naturally observable at both lines $\mathsf{L7}_{\mathtt{thr1}}$ and $\mathsf{L1}_{\mathtt{thr2}}$ to be capable of observing the time at which the instructions execute and consequently extract which instruction was the last to define `e`.

Problem Statement. As mentioned earlier, in addition to inputs and outputs, we consider *instrumented* variables as naturally observable variables, as their value can be explicitly observed. When program development is divided among multiple development groups, the program may suffer from *under-* or *over-instrumentation* defects caused by developers due to lack of knowledge about other developments.

Our goal is to optimize data-flow instrumentation in concurrent programs to tackle over- and under-instrumentation defects. Informally, given a set of *desired variables* required for debugging, our goal is to find the minimum set of variable in the program that should be made naturally observable (i.e., instrumented), so that the set of desired variables become observable. Formally, we aim at solving the following optimization problem:

> Given a concurrent program and a set V of desired variables to be made observable, decide whether there exists a set of variables V', where $|V'| \leq k$ for some positive integer k, such that by making variables in V' naturally observable, there exists an observable sub-slice $\mathcal{S}_v$ for all $v \in V$.

4 Approach

In this section, we propose our approach to solve the optimization problem, introduced in Section 3. Our method consists of three steps: (1) extracting program slices of variables required to be observed (i.e., desired variables), (2) building a graph representation of slices, and (3) transforming the optimization problem using the graph built in Step 2 into a satisfiability decision problem. These steps are discussed in Subsections 4.1, 4.2, and 4.3, respectively.

4.1 Extracting Program Slices

Given a concurrent program and a set V of desired variables, we first extract the program slices of V from the Static Single Assignment (SSA) [4] representation of the program by leveraging the slicing algorithm proposed in [11]. Our slicing approach takes the following steps for all $v \in V$:

1. We find the threads, say *thr*, that execute instructions defining v. Then, we extract the dependency chains of v by only using instructions of *thr*; i.e., we do not expand the chains over different threads. We use a reachability algorithm [14] to extract these chains.
2. For every chain σ found in Step 1, we extract the last variable v' of σ. We check if v' is defined by an instruction of a thread, say *thr*$'$, which does not run concurrently with *thr*. If so, we find dependency chains of v' in *thr*$'$ using the method in Step 1. Subsequently, we append the newly extracted chains to σ and create a new set of chains which we add to the set of dependency chains of v. We repeat this step until no new chains are created.

3. For every chain σ identified in Steps 1 and 2, we extract the instructions, say L, in σ which use a shared variable *sv*. Next, we extract the threads, say *thr*, that execute instruction L. Then, we check if *sv* is defined by instructions executed by a different thread, say *thr*′, that runs concurrently with *thr*. If so, we find the instructions, say L', that define *sv* in *thr*′. We subsequently check if data dependency is possible from the *sv* used in L to *sv* defined in L'. To this end, we perform a lightweight static analysis to take synchronization issues into account. For instance, if L and L' are both protected with the same lock (e.g., a mutex or transaction), then data dependency between L and L' can be eliminated. If a dependency is possible, we apply Steps 1 and 2 to extract the corresponding dependency chains for *sv* in *thr*′. Then, we append *sv*'s dependency chains to σ and create a new set of chains which we add to the set of dependency chains of v. We repeat this step until no new chains are created.
4. Finally, we test whether each dependency chain σ found for v is indeed possible by checking if there exists an execution path in the CCFG of the program that creates σ. If not, we discard σ from the set.

4.2 Building Observability Graph

Let v be a desired variable and $\mathcal{S}_v$ be the program slice for $v \in V$. In order to find the minimum number of variables for instrumentation in a systematic fashion, we build the *observability graph* [12] that encodes program slice $\mathcal{S}_v$. Let $\mathcal{V}_{\mathcal{S}_v}$ be the set of all variables involved in $\mathcal{S}_v$ and $\mathcal{I}_{\mathcal{S}_v}$ be the set of all instructions involved in $\mathcal{S}_v$. We construct the observability graph $\mathcal{G} = \langle V_\mathcal{G}, A_\mathcal{G} \rangle$ as follows.

- *(Vertices)* $V_\mathcal{G} = C_\mathcal{G} \cup U_\mathcal{G}$, where $C_\mathcal{G} = \{c_i \mid i \in \mathcal{I}_{\mathcal{S}_v}\}$ and $U_\mathcal{G} = \{u_v \mid v \in \mathcal{V}_{\mathcal{S}_v}\}$. We call the set $C_\mathcal{G}$, *context vertices* (i.e., one vertex for each instruction in $\mathcal{I}_{\mathcal{S}_v}$) and the set $U_\mathcal{G}$, *variable vertices* (i.e., one vertex for each variable in $\mathcal{V}_{\mathcal{S}_v}$).
- *(Arcs)* $A_\mathcal{G} = \{(u, c) \mid u \in U_\mathcal{G} \ \wedge \ c \in C_\mathcal{G} \wedge$ the value of variable u is defined by context $c\} \ \cup \ \{(c, u) \mid u \in U_\mathcal{G} \ \wedge \ c \in C_\mathcal{G} \wedge$ variable u is used by context $c\}$.

For example, the observability graph[3] of variable `a` is presented in Figure 1(c). For instance, context vertex $\mathtt{L9}_{\mathtt{thr1}}$ shows dependency of `a` to variables (directly) `b`, `c`, `e` and (indirectly) `x`. Also, Figure 1(c) shows that shared variable `e` affects the value of `a` through instruction $\mathtt{L1}_{\mathtt{thr2}}$ as well. We emphasize that although our construction of an observability graph is with respect to one variable, it is trivial to merge several graphs for multiple desired variables.

In the context of an observability graph, notice that a variable vertex v is observable if there exists a context vertex c, such that (1) (v, c) is an arc in the graph, and (2) all variable vertices, say v', are observable, where (c, v') is an arc

[3] For simplicity, this graph is constructed from the original source code and not from its SSA mode.

in the graph. Thus, our objective is to find the minimum number of variable vertices of the graph whose instrumentation makes the root vertex of the graph observable.

4.3 SAT-Based Optimization

In [12], we prove that the optimization problem for observability graphs of sequential programs is NP-complete. Thus, in the context of concurrent programs, the problem involves two exponential blow-ups: one for computing program slices [11] (and, hence, an observability graph), and (2) solving the optimization problem. In order to cope with the second exponential blow-up, in [12], we leverage a mapping to integer linear programming (ILP). In this Subsection, we introduce a more efficient method by transforming our optimization problem into the propositional satisfiability problem (SAT)[4].

Let $\mathcal{G} = \langle V_\mathcal{G}, A_\mathcal{G} \rangle$ be an observability graph and $V' \subseteq V_\mathcal{G}$ represents the set of desired variables. We include the following variables:

- $X = \{x_v \mid v \in V_\mathcal{G}\}$: each variable vertex v is mapped to a Boolean variable x_v, where $x_v = \mathit{true}$ if v is observable and *false* otherwise.
- $Z = \{z_v \mid v \in V_\mathcal{G}\}$: each variable vertex v is mapped to a Boolean variable z_v, where $z_v = \mathit{true}$ if v is instrumented and *false* otherwise.
- $Q = \{q_c \mid c \in (C_\mathcal{G} \cup C_H)\}$: each context vertex c is mapped to a Boolean variable q_c, where $q_c = \mathit{true}$ if all variables used by c are observable and *false* otherwise. In addition, $C_H = \{c_v \mid v \in V_\mathcal{G}\}$ contains context vertices for each variable $v \in V_\mathcal{G}$ representing *hypothetical* instructions that would instrument v; i.e., such instrumentations do not exist in the original code and will only be added to the code if v is chosen to be instrumented. Each context vertex $c \in C_H$ is mapped to a Boolean variable q_c: the value of $q_c = \mathit{true}$ if c is added to the code to instrument the corresponding variable and *false* otherwise.
- $Y = \{y_z \mid z \in Z\}$: for each variable $z \in Z$, we include an integer variable y_z for our optimization objective.

Constraints on variable vertices. Obviously, every desired variable must be observable. Hence, we add the following constraint for each $v \in V'$:

$$x_v \iff \mathit{true}.$$

Moreover, each variable $x_v \in X$ is *true* if and only if the value of the variable v is observable via the context vertex that defines v or by instrumenting v:

$$x_v \iff (q_{c_v} \ \vee \ q_{c'_v}),$$

[4] Our experiments show that the SAT-based method in this paper is considerably faster than the ILP-based method in [12].

where c_v is the context vertex that defines v and c'_v is the instruction that instruments v. Finally, we require that $y_z \in Y$, where $z \in Z$, has value 1 when $z = \mathit{true}$ and has value 0 otherwise:

$$(y_z = 1) \iff z \qquad \text{and} \qquad (y_z = 0) \iff \neg z.$$

Constraints on context vertices. The value of a variable $q_c \in Q$, where $c \in C_{\mathcal{G}}$, must be *true* if and only if all the variables used by c are observable:

$$q_c \Longleftrightarrow \bigwedge_{v \in V_c} x_v,$$

where $V_c = \{u \mid (c,u) \in A_{\mathcal{G}}\}$. On the other hand, if $c_v \in C_H$, then q_c must be *true* if and only if when v is instrumented; i.e., when $z_v = \mathit{true}$.

$$q_{c_v} \Longleftrightarrow z_v$$

In this expression, when $z_v = \mathit{true}$ an instruction is added to the code to instrument v.

Optimization objective. Following the optimization criterion presented in Section 3, we require that the number of variables to be instrumented is not greater than K:

$$\sum_{y \in Y} y \leq K,$$

for some positive integer K specified by the user[5].

5 Experiments

Our goal in this section is twofold: (1) to demonstrate the effectiveness of our method through measuring the number of instrumentations removed from an *over-instrumented* program after applying our method, and (2) to evaluate the impact of our approach by studying the performance (i.e., execution time) of optimally instrumented programs as compared to their over-instrumented versions. We note that in the over-instrumented versions, any instruction that can change the state of concurrent data structures is instrumented for debugging purposes.

Our approach is implemented in a tool chain consisting three phases:

1. First, we implement a new pass over LLVM [7] that takes a program's source code and the set of desired variables as input. The pass extracts program slices using the method described in 4.1 and the static single assignment (SSA) mode of the program code. We currently do not handle alias and pointer data structures.

[5] To find the smallest K one can solve the SAT formula multiple times, each time with a smaller K until the formula is not satisfiable.

2. Given the extracted program slices, we transform the respective optimization problem into a SAT formula in the input language of our SAT-solver using the method described in Section 4.3.
3. We solve the generated SAT model using the Yices SMT-solver [1]. The solution presents the set of variables that need to be instrumented in the source code (set of naturally observable variables).

This section is organized as follows. In Subsection 5.1, we discuss the parameters of our experiments. In Subsection 5.2, we analyze the results of our experiments.

5.1 Experimental Setup

Our two case studies are concurrent implementations of *linked-lists* and *red-black trees.* We use regular arrays to eliminate possible pointer analysis. In both test cases, *insert*, *delete*, and *search* operations can run concurrently, where a thread inserts data elements (i.e., *producer*) and a thread deletes data elements (i.e., a *consumer*). In addition, we consider the following synchronization methods for each case study:

1. **Lock-based.** These algorithms employ blocking data structures (e.g., semaphores and mutexes) to enforce linearizable insertion and deletion. In particular, we use our mutex-based implementation of concurrent linked lists and the algorithm in [5] for concurrent red-black trees (with 656 lines of code and 57 desired variables).
2. **Non-blocking.** This group of solutions ensures that threads competing for a shared resource do not have their execution indefinitely postponed by mutual exclusion. We use the following implementation for our experiments:
   - We use the *lock-free* algorithm in [6] for concurrent linked lists (with 1180 lines of code and 85 desired variables) implemented by the CAS (compare-and-swap) operation. Lock-free algorithms ensure that if the program threads run sufficiently long at least one of the threads makes progress.
   - We use the *obstruction-free* algorithm in [3] for concurrent linked lists (with 797 lines of code and 71 desired variables) implemented by virtual locks and the CAS operation. Obstruction-free algorithms guarantee that at any point, a single thread executed in isolation (i.e., with all obstructing threads suspended) for a bounded number of steps will complete its operation.
   - Algorithms based on *software transactional memory* (STM) hide synchronization issues from the programmer; i.e., the programming language provides the programmer with atomic constructs in which reading and writing shared variables take place. In particular, we use our own STM-based implementation of concurrent linked lists and the algorithm in [5] for concurrent red-black trees (with 730 lines of code and 56 desired variables).

	Application	Concurrency	Count	Mean	Median	SEM	CI-95	Min	Max
1	Linked-list	Nested-locks	5	14.13	14.01	0.22	0.30	13.65	14.94
2	Linked-list	Lock-free [6]	5	5526.37	5529.79	6.53	9.06	5502.49	5539.07
3	Linked-list	Obstruction-free [3]	5	2686.26	2683.78	8.01	11.10	2663.26	2707.92
4	Linked-list	STM	5	257.13	258.05	1.14	1.58	252.72	258.90
5	Red-black tree	Nested-locks [5]	20	3.95	3.95	0.00	0.00	3.95	3.95
6	Red-Black tree	STM [5]	10	4.46	4.46	0.00	0.00	4.46	4.46

Table 1. Detailed numbers for the instrumented version with I/O delay= $100\mu s$ and number of insert operations = 200.

The set of desired variables in our experiments include the data contained in the linked-list/red-black tree at any point of execution and the temporary variables used in search, addition, and deletion operations. Thus, instrumenting all these variables is likely to result in over-instrumentation.

Parameters that affect the execution time of experiments are: (1) number of producer and consumer threads, (2) number of insert and delete operations, (3) type of data elements (e.g., long, short, int), (4) time consumed by each instrumentation instruction, (5) number of shared variables, and (6) structure of the source code (i.e., synchronization method). In our experiments, we keep the number of producer and consumer threads, type of data elements, and number of shared variables as constants. The rest are obviously variables in our experiments. In particular, we incorporate different numbers of insert operations to study the impact of our optimization on long running programs. Different durations of the instrumentation instruction show the impact of our method for different instrumentation technologies. For instance, `printf()` statements normally take $80\mu s$, whereas EEPROM data logs take $1ms$. All experiments in this section are run on workstations ranging from a Core2Duo to a Core I3 quad-core machines with sufficient memory. Each test series is completed on the same machine, hence, the execution-time measurements from one test series are comparable.

We measure the execution time using the well-accepted utility time. Since individual measurements can be inaccurate (due to context switches or I/O operations between the program and time), we repeated each experiment several times and carried out solid statistical analysis. We have collected sufficient data to have representative and robust results. While we cannot provide the key metrics for all individual data points, Table 1 shows the results for the data series with the least number of values—it is the series where several configurations take more than an hour to complete. In Table 1, SEM and CI-95 abbreviate Standard Error of the Mean and 95% Confidence Interval, respectively. We also performed the following consistency checks on the data: measurements must be positive and growing with respect to the number of insert operations and the amount of I/O delay.

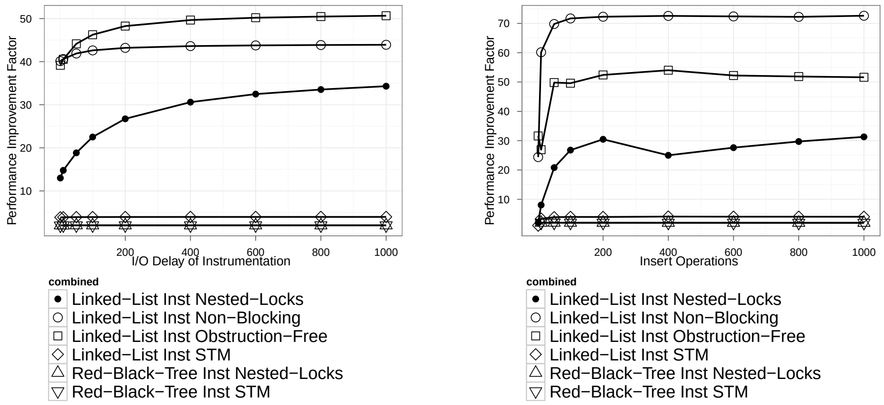
5.2 Results and Analysis

Reduction in number of instrumentations. We apply our method to over-instrumented implementations to optimize the instrumentations of the source code. Table 2 shows that our method achieves a 45% reduction on average across our case studies. Although the set of desired variables is common among different implementations, we observe different reductions in instrumentation, as the amount of reduction depends on the structure of the code. For instance, we do not require instrumentation in the atomic sections of STM-based algorithms, since the changes are local to the threads and do not affect shared variables; i.e., we only need to instrument the values committed into the shared variables by the threads (experiments 4 and 6). On the other hand, in the lock-free and obstruction-free implementations, we require more instrumentation due to lack of synchronization and more possible interleaving scenarios that must be observed. In nested-locks implementations, since the changes carried out in between the locks directly affect shared variables, we need to instrument the code in between the locks. Hence, it requires more instrumentation as compared to STM-based algorithms, but less as compared to non-blocking algorithms, as they have less interleaving scenarios.

Enhancement in performance. We now compare the performance of over-instrumented test cases against the performance of their optimally instrumented versions in terms of execution time. In the first set of experiments (see Figure 3(a)), we compare the performance of the case studies, where the I/O delay (time consumption) of instrumentation instructions varies from $1\mu s$ to $1ms$, while the number of insert operations is constant (= 100). We have collected 1692 data samples from these experiments for statistical soundness (discussed later). Obviously, Figure 3(a) shows that the performance of optimally instrumented implementations is significantly better than the over-instrumented versions. For example, the performance of both red-black tree implementations improve with a factor of two. The reason behind this small improvement is that our method is forced to place 48% of the required instrumentation in loop structures to make desired variables observable. Hence, the effect of the I/O delay of each instrumentation on the performance is multiplied by the loop counts. On the other hand, Figure 3(a) shows a 40 times performance improvement in lock-free

Application	Concurrency	Original Inst.	Optimized Inst.
1 Linked-list	Nested-locks	43	20
2 Linked-list	Lock-free [6]	49	23
3 Linked-list	Obstruction-free [3]	42	24
4 Linked-list	STM	28	15
5 Red-black tree	Nested-locks [5]	320	205
6 Red-black tree	STM [5]	294	189

Table 2. Reduction in Instrumentation



(a) Performance improvement vs. instrumentation instruction I/O delay with constant number of insert operations (= 100)

(b) Performance improvement vs. the number of insert operations with constant instrumentation I/O delay (= $100\mu s$).

Fig. 3. Performance evaluation of instrumentation optimization.

and obstruction-free implementations, as the majority of the instrumentations introduced by our method reside outside loop structures. In general, the improvement factor differs from one case study to another, since instrumentation locations tightly depend upon the structure of the source code. In addition, the results show that the improvement factor in performance is insensitive to different durations for instrumentation instructions.

In the second set of experiments, we compare the execution time of the case studies, where the number of concurrent insertions in the linked-list/red-black tree varies from 1 to 1000 (see Figure 3(b)). We collected 966 data samples from these experiments for statistical soundness. Obviously, Figure 3(a) shows that the performance of the optimally instrumented implementations is better than over-instrumented versions. The results show that we achieve a small improvement in the performance of red-black tree implementations due to the same reason discussed in the previous experiment. In addition, the results show that the improvement in performance in each case study is insensitive to the number of insertions, although the improvement factor differs from one case study to another (as the improvement factor depends upon the source code structure).

6 Related Work

The work on designing effective runtime logs are the closest to our work. LogEnhancer [23] enhances log messages in a source code by automatically adding causally-related information to messages. It uses static analysis to find data that must be presented in the messages to identify the source of a failure. Our work differs from LogEnhancer in two aspects: (1) we are only concerned with the efficient extraction of causally-related data, and (2) we focus on concurrent programs while LogEnhancer works on sequential programs.

The work in the context of testing concurrent programs focuses on two aspects. The first aspect focuses on finding interleaving scenarios that cause crashes. CHESS [10] is a stateless model checker that finds erroneous interleaving scenarios by injecting preemptions into programs. CTrigger [16] finds interleaving scenarios that violate correct access patterns of shared variables. In [22], a technique is proposed for extracting interleaving scenarios that lead to a Heisenbug in a multi-core architecture. The second aspect concentrates on achieving deterministic replay of previously seen interleaving scenarios. These approaches carry out logging either at software level [13, 15, 17] that imposes considerable overhead or at hardware level [9, 19] that imposes less overhead but their applicability is limited. Our work compliments the work in this area, since our approach extracts the information required to find corruptions caused by erroneous interleavings. Moreover, our method checks whether log messages are sufficient for successful replay and path re-construction.

7 Conclusion and Future Work

In this paper, we introduced an automated technique to optimize instrumentation of *multi-threaded* programs to achieve software *observability*. Intuitively, observability in a concurrent program enables a debugger to extract the value of a set of *desired variables* through instrumenting another (possibly smaller) set of variables, called *naturally observable*. Thus, our optimization method identifies the minimum set of naturally observable variables whose instrumentation makes the value of desired variables extractable. Since our optimization problem is NP-complete, we encoded the problem as a propositional satisfiability problem (SAT) to leverage powerful SAT-solvers to tackle our problem.

In our tool chain, we used LLVM and a slicing algorithm to extract program data-flow dependencies. This is achieved by implementing a new pass over LLVM that takes the source code and a set of desired variables as input and generates the full set of data-flow dependencies as output through which we generate a SAT formula. The solution to the SAT problem is the set of variables that need to be instrumented. Our experimental results on concurrent linked lists and red-black trees using different concurrency techniques show significant gains (up to 50 times) in performance of optimally-instrumented programs using our method as compared to ad-hoc over-instrumented programs.

For future work, we are considering two main research directions: (1) techniques for solving the optimization problem more efficiently, and (2) extending the concept of observability to other domains. For the former, we are currently working on using lightweight model checking as well as developing our own heuristics to compute slices in concurrent programs more efficiently while taking inter-thread synchronizations into account. These techniques potentially result in obtaining less complex SAT models. For the latter, we are considering methods that address pointer data structures as well. Another direction is to devise probabilistic methods that make desired variables observable with certain probabilities.

8 Acknowledgement

This research was supported in part by NSERC DG 357121-2008, ORF RE03-045, ORE RE04-036, ORF-RE04-039, ISOP IS09-06-037, APCPJ 386797-09, and CFI 20314 with CMC.

References

1. Yices: An SMT Solver. `http://yices.csl.sri.com`.
2. P. Ammann and J. Offutt. *Introduction to Software Testing.* Cambridge University Press, New York, USA, 2008.
3. H. Attiya and E. Hillel. Built-In Coloring for Highly-Concurrent Doubly-Linked Lists . *Distributed Computing*, 4167:31–45, 2006.
4. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.
5. K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), 2007.
6. T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, pages 300–314, 2001.
7. C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–, 2004.
8. J. Lee, S. P. Midkiff, and D. A. Padua. A constant propagation algorithm for explicitly parallel programs. *International Journal of Parallel Programming*, 26(5):563–589, 1998.
9. P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay. In *ASPLOS*, pages 73–84, 2009.
10. M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *PLDI*, pages 446–455, 2007.
11. M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *ISSTA*, pages 180–190, 2000.
12. S. Navabpour, B. Bonakdarpour, and S. Fischmeister. Software debugging and testing using the abstract diagnosis theory. In *LCTES*, pages 111–120, 2011.
13. R. H. B. Netzer and M. H. Weaver. Optimal Tracing and Incremental Reexecution for Debugging Long-Running Programs. In *PLDI*, pages 313–325, 1994.
14. K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *SDE*, pages 177–184, 1984.
15. D. Z. Pan and M. A. Linton. Supporting Reverse Execution for Parallel Programs. In *SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 124–129, 1988.
16. S. Park, S. Lu, and Y. Zhou. Ctrigger: Exposing Atomicity Violation Bugs from Their Hiding Place. In *ASPLOS*, pages 25–36, 2009.

17. M. Ronsse, K. D. Bosschere, M. Christiaens, J. C. D. Kergonneaux, and D. Kranzlmuller. Record/Replay for Nondeterministic Program Executions. *Communication of the ACM*, 46(9):62–67, 2003.
18. Ulrich Schmid. Monitoring of Distributed Real-Time Systems. *Real-Time Systems*, 7(1):33–56, 1994.
19. S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A Lightweight Extension For Rollback and Deterministic Replay for Software Debugging. In *USENIX*, pages 29–44, 2004.
20. Henrik Thane and Hans Hansson. Towards Systematic Testing of Distributed Real-Time Systems. In *RTSS*, pages 360–369, 1999.
21. Henrik Thane, Daniel Sundmark, Joel Huselius, and Anders Pettersson. Replay Debugging of Real-Time Systems Using Time Machines. In *IPDPS*, page 8, 2003.
22. D. Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. Analyzing Multicore Dumps to Facilitate Concurrency Bug Reproduction. In *ASPLOS*, pages 155–166, 2010.
23. D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving Software Diagnosability via Log Enhancement. In *ASPLOS*, pages 3–14, 2011.