Optimal Instrumentation of Data-flow in Concurrent Software

Samaneh Navabpour
Dept. of Elec. and Comp. Engineering
University of Waterloo
200 University Avenue West
Waterloo N2L 3G1, Canada
Email: snavabpo@ece.uwaterloo.ca

Borzoo Bonakdarpour
Dept. of Elec. and Comp. Engineering
University of Waterloo
200 University Avenue West
Waterloo N2L 3G1, Canada
Email: borzoo@ece.uwaterloo.ca

Sebastian Fischmeister
Dept. of Elec. and Comp. Engineering
University of Waterloo
200 University Avenue West
Waterloo N2L 3G1, Canada
Email: sfischme@ece.uwaterloo.ca

***Abstract*—In this paper, we propose an automated technique for optimal instrumentation of *multi-threaded* programs for debugging and testing purposes. 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 our experimental results on popular concurrent data structures show significant gains in performance of instrumented programs using our method as compared to ad-hoc over-instrumented programs.**

***Keywords*-Software debugging; Testing; Diagnosis; Logging; Instrumentation; Concurrency; Multi-threading; Data-flow**

I. 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 siginificant 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 emplyed. Examples include interactive debugging, control- and data-flow analysis, log files, memory dumps, and profiling. Incorporating these techniques mostly requires adding additional instructions to the program under scruitny, 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 system may change the behavior of that system. These problems are amplified significantly in the context of concurrent programs. This is due to the fact that instrumenting these programs adds to their inherent non-determinisitc nature, causing different executions with the same data and 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 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 [1], we proposed a notion of *observability* as the ability to test various features of a *sequential program* and observe its outcome to check if it conforms to the software's specification. The traditional methods for achieving observability incorporate ad-hoc instrumentation techniques [2], [3], [4] 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 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 and show that the concept has a different nature as compared to sequential porgrams due to the existance of shared variables and interleaving dependencies. Roughly speaking, observability in a concurrent program enables a debugger to extrat 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 [5] and the method presented in [6] 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 genrate a SAT model which is the input to the SMT-solver Yices [7]. The solution to the SAT model is the set of variables that need to be instrumented.
- We conduct experiments on two popular concurrent data structures: *linked lists* and *red-black trees*. We consider different implmentations of these data structures with respect to different liveness criteria and synchronization tools, such as lock-based, software transactional memory, and non-blocking implemetations. Our experiments show that our method effectively optimizes instrumentation instructions, resulting in siginificant performance improvement (in some cases up to 50 times), as compared to an ad-hoc over-instrumented program.

Organization. The rest of the paper is organized as follows. In Section II, we present the preliminary concepts on control-flow graphs and data-flow dependencies. Section III 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 IV. Section V presents the results of our experiments. Related work is discussed in Section VI. Finally, we make concluding remarks and dicuss future work in Section VII.

II. Preliminares

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

A. Concurrent Control-flow Graphs

Intutively, 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:

```
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;
}
```

Figure 1. Two threads running concurrently.

- 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.
- 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 of 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 $ThreadExit$. Figure 1 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 is shown in Figure 2.

Notation: Let $\pi = n_1, n_2, \ldots, n_k$ be a sequence of nodes of a CCFG G and thr be a thread of G. By $thr(\pi)$, 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) $thr(\pi) = x_1, x_2, \ldots, x_j$ is a total order, and (ii) for all i, where $1 \leq i \leq j - 1$, $(x_i, x_{i+1}) \in A$, and

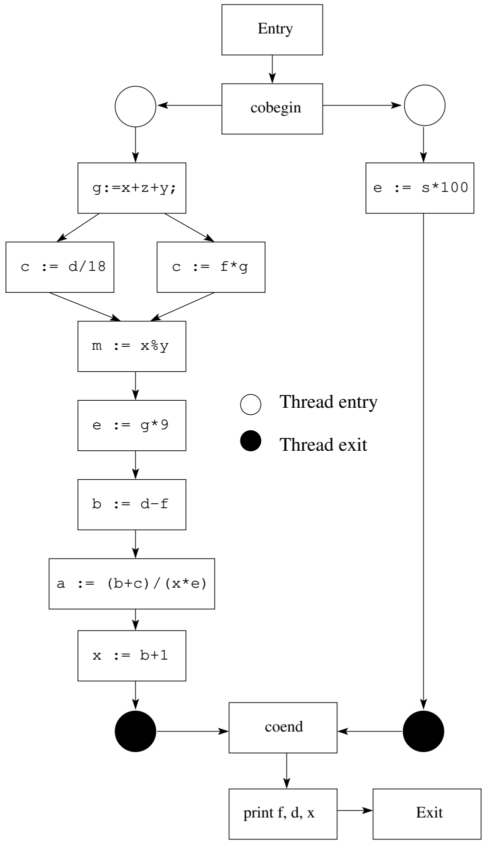


Figure 2. CCFG of the source coded in Figure 1.

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 paths of the CCFG in Figure 2 is $\mathsf{L1}_{\mathtt{thr1}}$, $\mathsf{L2}_{\mathtt{thr1}}$,$\mathsf{L3}_{\mathtt{thr1}}$, $\mathsf{L6}_{\mathtt{thr1}}$, $\mathsf{L1}_{\mathtt{thr2}}$, $\mathsf{L7}_{\mathtt{thr1}}$, $\mathsf{L8}_{\mathtt{thr1}}$, $\mathsf{L9}_{\mathtt{thr}}$, $\mathsf{L10}_{\mathtt{thr1}}$, $\mathsf{L0}$.

B. 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 the errors.

Definition 3: We say that the value of a variable v *depends* on the value of variable v' iff $v = F(v', V)$ [6], where F is an arbitrary function and V is the remaining set of F's arguments called *parameters*. ■

In a source code, any instruction that updates the value of a variable creates a dependency. For instance, in function `thr1`, in Figure 1, 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. Thus, in order to compute data dependencies statically, we consider a conservative set of instructions that can define the value of variable at run time. 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, 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 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 $\langle \mathsf{c}, \mathsf{L3}_{\mathtt{thr1}}, \mathsf{d} \rangle$, we have $\langle \mathsf{c}, \mathsf{L3}_{\mathtt{thr1}} \mid \mathsf{L5}_{\mathtt{thr1}}, \mathsf{d} \rangle$. In other words, `c` may depend on `d` if $\mathsf{L3}_{\mathtt{thr1}}$ or $\mathsf{L5}_{\mathtt{thr1}}$ execute. In this case, parameters of $\mathsf{L3}_{\mathtt{thr1}} \mid \mathsf{L5}_{\mathtt{thr1}}$ is the set $\{\mathsf{d}, \mathsf{g}, \mathsf{f}\}$.

During program execution, the value of a variable depends on a 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 $\mathsf{L9}_{\mathtt{thr1}}$ indirectly depends on:

- variables `d` and `f` used by instruction $\mathsf{L8}_{\mathtt{thr1}}$ which defines `b`,
- variables `d`, `f`, and `g` used by $\mathsf{L3}_{\mathtt{thr1}} \mid \mathsf{L5}_{\mathtt{thr1}}$ which define `c`,
- variable `g` used by instruction $\mathsf{L7}_{\mathtt{thr1}}$ which defines `e`, and
- variables `x`, `z`, and `y` used by $\mathsf{L1}_{\mathtt{thr1}}$ which defines `g`

In order to capture the effect execution paths have on the value of a variable, we introduce the notion of dependency chains.

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

Figure 3. Two threads running sequentially.

Definition 4: Let G be CCGF and v be a variable. A *dependency chain* for v is a sequence $\sigma = \langle v_1, n_1, v_2\rangle\langle v_2, n_2, v_3\rangle \ldots \langle v_{k-1}, n_{k-1}, v_k\rangle$ of dependencies where [6]:

- $v_1 = v$, and
- the sequence $n_1, n_2, \ldots, n_k$ is an execution path of G between basic blocks n_1 and n_k. ■

Clearly, determining data dependencies relies on the structure of the source code. For example, in the source code of Figure 1 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 3, 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 enumerate a subset of all dependency chains that are *maximal*.

Definition 5: Let $\mathcal{S}_v = \{\sigma_1, ..., \sigma_n\}$ 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 σ in $\mathcal{S}_v$, there does not exist a dependency chain σ', such that $\sigma\sigma'$ is in $\mathcal{S}_v$. ■

For example, $\mathcal{S}_\mathtt{a} = \{\langle \mathtt{a}, \mathtt{L9_{thr1}}, \mathtt{b}\rangle, \langle \mathtt{a}, \mathtt{L9_{thr1}}, \mathtt{b}\rangle\langle \mathtt{b}, \mathtt{L8_{thr1}}, \mathtt{d}\rangle, \langle \mathtt{a}, \mathtt{L9_{thr1}}, \mathtt{x}\rangle, \langle \mathtt{a}, \mathtt{L9_{thr1}}, \mathtt{e}\rangle\}$ is *not* a maximal dependency chain of `a`, as the dependency chain $\langle \mathtt{a}, \mathtt{L9_{thr1}}, \mathtt{b}\rangle$ is a prefix of the dependency chain $\langle \mathtt{a}, \mathtt{L9_{thr1}}, \mathtt{b}\rangle\langle \mathtt{b}, \mathtt{L8_{thr1}}, \mathtt{d}\rangle$. To convert $\mathcal{S}_\mathtt{a}$ into a maximal dependency set, we must either remove $\langle \mathtt{a}, \mathtt{L9_{thr1}}, \mathtt{b}\rangle$ or $\langle \mathtt{a}, \mathtt{L9_{thr1}}, \mathtt{b}\rangle\langle \mathtt{b}, \mathtt{L8_{thr1}}, \mathtt{d}\rangle$ from $\mathcal{S}_\mathtt{a}$.

Finally, we introduce the notion of a program *slice* [6]. 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 $\mathtt{L9_{thr1}}$, $\mathtt{L8_{thr1}}$, $\mathtt{L7_{thr1}}$, $\mathtt{L5_{thr1}}$, $\mathtt{L3_{thr1}}$, $\mathtt{L1_{thr1}}$, and $\mathtt{L1_{thr2}}$.

III. Observability in Concurrent Programs

In this section, we establish the notion of *observability* in concurrent systems as a necessary condition for debugging concurrent programs. In the context of program debugging, outputs often provide us with insufficient information to locate the bug(s) that lead to faulty 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 output and input variables, respectively.

Definition 7 (Natural Observability): A value of a variable v is *naturally observable* iff the value is an output or input of the system. ■

For example, in system Figure 1, variables `d`,`f`,`x`, and `y` are *naturally observable* since they are outputs and inputs.

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 subset of $\mathcal{S}$, where each dependency chain in $\mathcal{S}'$ is a prefix of a chain in $\mathcal{S}$. ■

To motivate the idea, note 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. Although, `d` and `f` can only lead to extracting the value of `b` and `c` at lines L8 and L3. The value of `c` is still unknown since we can not predict if `c` will be defined via line `L3` or `L5` at runtime. As a result, we can not guarantee knowing the value of `c` at line L9 without knowing 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 which need to be satisfied to the extract the value of a variable in concurrent programs.

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

1) for each dependency $\langle v, n, v' \rangle$ in a dependency chain in slice $\mathcal{S}$, 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 defined by thr' as well, 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 $\mathcal{S}_v = \{\sigma_1, ..., \sigma_n\}$ 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. 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. In addition to inputs and outputs, we consider variables which are *instrumented* as naturally observable variables, since their value can be explicitly observed for debugging purposes. When program development is divided among multiple development groups, the program may suffer from *under-* or *over-instrumentation* casued by developers due to lack of knowledge about other developements.

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 to be made naturally observable (i.e., instrumented) so that the set of desired variables can be made observable.

Formally, we aim at solving the following optimization problem:

> Given a concurrent program and a set V of variables required to be 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 each v in V.

IV. Approach

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

A. Extracting Program Slices

Given a concurrent program and a set V of variables which are desired to be observed, we first extract program slices of V from the Static Single Assignment (SSA) [9] representation of the program by leveraging the slicing algorithm proposed in [10]. 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 dependency chains of v only using instructions of thr; i.e., we do not expand the chains over different threads. We use a reachability algorithm [11] 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. Now, 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.

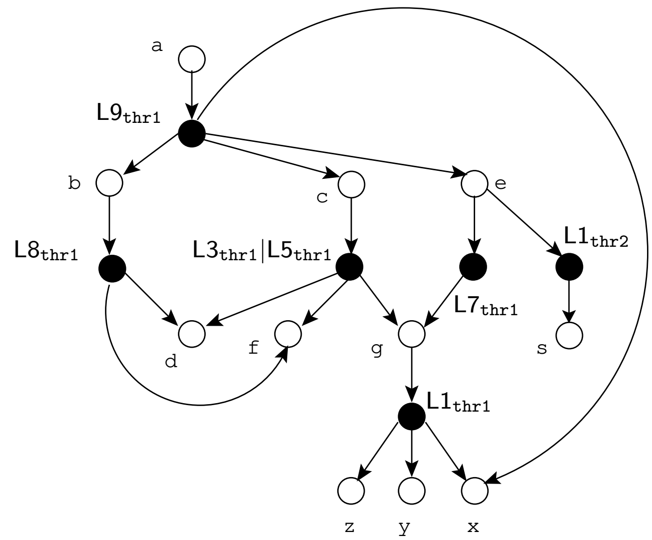


Figure 4. Observability graph of the source coded in Figure 1.

B. 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 more systematic fashion, we build the *observability graph* [1] 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)* $A_\mathcal{G} = \{(u, c) \mid u \in U_\mathcal{G} \wedge c \in C_\mathcal{G} \wedge \text{the value of variable } u \text{ is defined within context } c\} \cup \{(c, u) \mid u \in U_\mathcal{G} \wedge c \in C_\mathcal{G} \wedge \text{variable } u \text{ is used in context } c\}$.

For example, observability graph of variable `a` is presented in Figure 4. For instance, context vertex $\mathsf{L9}_{\mathtt{thr1}}$ shows dependency of `a` to variables `b`, `c`, `e` and `x`. Also, Firgure 4 shows that shared variable `e` affects the value of `a` through instruction $\mathsf{L1}_{\mathtt{thr2}}$ 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 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.

C. SAT-Based Optimization

In [1], 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 [10] (and, hence, an observability graph), and (2) solving the optimization problem. In order to cope with the second exponential blow-up, in [1], we leverage a mapping to integer linear programming. In this section, we introduce a more effective $\Diamond$[1]method by transforming our optimization problem into a propositional satisfiability problem (SAT).

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 is 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 is 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

[1]sn: why are you saying effective?

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.

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

$$x_v \Longleftrightarrow true.$$

Morover, 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 the instrumentation of v's value:

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

where c_v is the context vertex that defines the 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 $z = true$ and has value 0 otherwize:

$$(y_z = 1) \iff z$$

$$(y_z = 0) \iff \neg z.$$

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 $z_v = true$.

$$q_{c_v} \Longleftrightarrow z_v$$

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

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

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

for some positive integer K specified by the user[2].

[2]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.

V. EXPERIMENTS

Our goal in this section is twofold: (1) to demonstrate the efectiveness 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-instrumentd versions.

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

1) First, we implement a new pass over LLVM [5] 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 IV-A and the static single assignment (SSA) mode of the program code. We currently do not handle alias and pointer data structures.
2) Given the extracted program slices, we transform the respective optimization problem into a SAT formula as described in Section IV-C in the input language of our SAT-solver.
3) We solve the generated SAT model using the Yices SMT-solver [7]. The solution presents the set of variables that need to be instrumented in the source code.

A. 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) **Nested-locks.** This group of solutions employ semaphores to enforce linearizable insertion and deletion.
2) **Software transactional memory (STM).** STM-based algorithms 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.
3) **Wait-free.** We use the non-blocking technique presented in [?] to enable linearizable insertion and deletion via the compare-and-swap (CAS) operation.
4) **Obstruction-free.** We use the lock-free technique presented in [?] to linearize the insertion and deletion operations by using virtual locks and reflect the changes into linked-list/red-black-tree atomically by leveraging CAS.

We do not use the non-blocking and obstruction-free red-black-tree as test cases, since they use a large number of pointers when implementing red-black-tree.

The set of desired variables in our experiments includes 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 opertions. 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 insertion and deletion operation, (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 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 execution iterations to study the impact of our optimization on long running programs. Different durations of the instrumentation instruction show the impact of our the 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 utilitiy 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. Table II shows the key statistics for several operating points in our measurements. More specifically, we performed the following consistency checks on the data: (1) measurements must be positive, (2) measurements must be growing with respect to the number of iterations and the amount of I/O delay, (3) ★[3]

B. Results

We present the results in two categories: (1) the reduction in the number of instrumentations achieved by our optimization method in the over-instrumented case studies, and (2) the enhancement in performance of the case studies after reducing the instrumentations using our optimization method.

1) Reduction in Number of Instrumentations: We apply our method to the over-instrumented test cases to optimize the instrumentations residing in their codes. The results are presented in Table I

The results show that our method achieves a 44.57% reduction on average in the test cases. Although the variables required for observation is common among the codes, we see different reductions in the instrumentations since the amount of reduction depends on the structure of the code. For instance, we do not require instrumentation in the code run atomically by the transactional memory, since the changes are local to the thread and do not affect the shared variable. Hence, we only need to instrument the final values committed into the shared variables by the thread (experiments 6 and 4). On the other hand, in the non-blocking and obstruction-free method we require more instrumentation because of the lack of synchronization and more possible interleaving scenarios which must be observed. In nested-locks, since the changes carried out in between the locks directly affect the shared variables, we need to instrument the code in between the locks. Hence, it requires more instrumentation compared to transactional memory, but less compared to nonblocking since it has less interleaving scenarios.

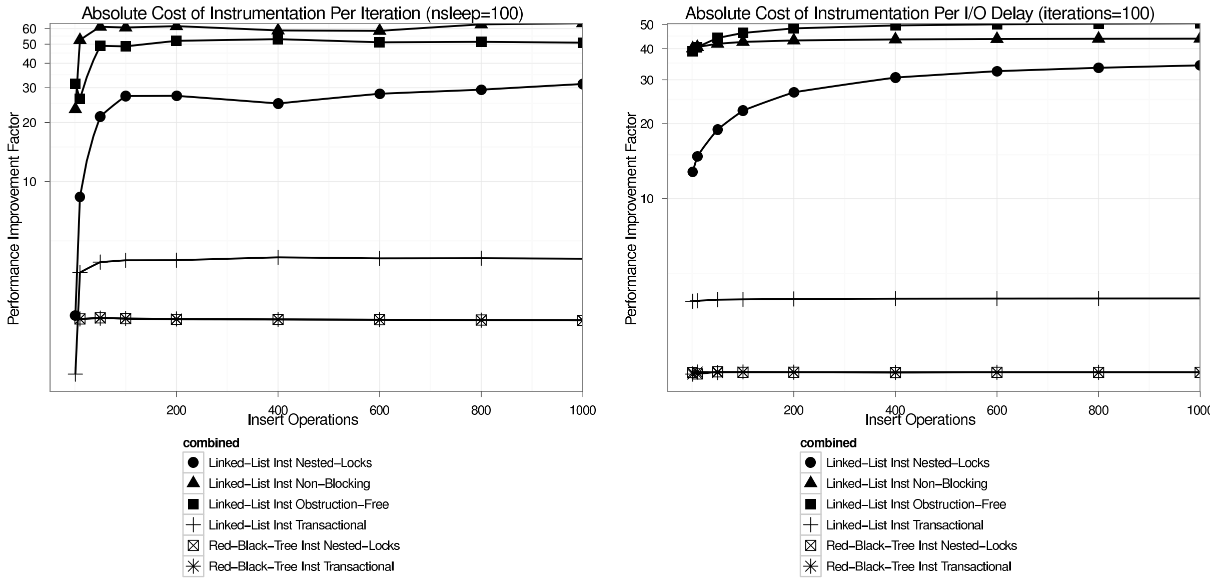
[3] sf: I'll add more here.

	Application	Concurrency	Inst.	Optimized Inst.
1	Linked-List	Nested-Locks	43	20
2	Linked-List	Non-Blocking	49	23
3	Linked-List	Obstruction-Free	42	24
4	Linked-List	Transactional	28	15
5	Red-Black-Tree	Nested-Locks	320	205
6	Red-Black-Tree	Transactional	294	189

Table I
REDUCTION IN INSTRUMENTATION

2) Enhancement in Performance: We compared the performance (i.e., execution time) of the over-instrumented test cases against their performance when their codes are optimally instrumented by our method. We present the results in Figure V-B2.

In the first set of experiments, we compare the performance of the test cases while varying the duration (time consumption) of the instrumentation instructions. We conducted the experiments for duration values of 1 μs to 1 ms. We collected 1692 data samples from these experiments where the results are presented in Figure 5(b). Results in 5(b) show that the performance of the test cases with optimal instrumentation is better compared to when they are over-instrumented. The results show that the performance of the red-black-tree test cases only 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 enable the observation of the set of desired variables in the red-black-tree test cases. Hence, the effect of the duration of each instrumentation on the performance is multiplied with respect to the loop counts. On the other hand, the results show a 40% performance improvement in non-blocking and obstruction-free linked-list test cases, since the majority of the instrumentations introduced by our method reside outside of loop structures. Thus, the amount of improvement differs from one test case to another since the locations where our method introduces instrumentations depends on the structure of the code. In addition, the results



(a) (b)

Figure 5. Performance Evaluation of Instrumentation Optimization

show that the improvement in performance in each test case is insensitive to different durations for instrumentation instructions.

In the second set of experiments, we compare the performance of the test cases while varying the number of insertions in the linked-list/red-black-tree. We conducted experiment for one insertion to 1000 insertions. We collected 966 data samples from these experiments where the results are presented in Figure 5(a). Results in 5(a) show the performance of the test cases with optimal instrumentation is better compared to when they were over-instrumented. The results show that we achieve a small improvement in the performance of red-black-tree test cases because of the same reason presented for the previous experiment. In addition, the results show that the improvement in performance in each test case is insensitive to the number of insertions, although, the amount of improvement differs from one test case to another since the improvement depends on the code structure.

A summery of our experiments is presented in Table II. ◊[4]

VI. Related Work

The work on designing effective runtime logs are closest to our work. LogEnhancer [12] enhances the log messages in a source code by automatically adding causally-relate information to the messages. It uses static analyses 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 casualty-related data, and (2) we focus on concurrent programs while LogEnhancer works in sequential programs.

The work in the context of testing concurrent programs focus on two aspects: (1) finding the interleaving scenarios which cause crashes. CHESS [13] is a stateless model checker which finds erroneous interleaving scenarios by injecting preemptions into the program. CTrigger [14] finds the interleaving scenarios which violate correct access patterns of shared variables. In [15], a technique is proposed which extracts the interleaving scenario which has led to a Heisenbug in a multicore architecture. (2) achieving deterministic replay of previously seen interleaving scenarios. These work carry out logging either at software level [16], [17], [18] which impose considerable overhead, or at hardware level [19], [20] which impose less overhead but their applicability is limited. Our work compliments the work done in this area, since we can extract the information required to find the corruption caused by an erroneous interleaving, and we can check if the log messages are sufficient for successful replay and path reconstruction.

Our work is not limited to concurrent programs. SherLog [21] is a diagnoses tool which uses the log messages in the crash log to automatically extract the execution paths and conditions which caused the crash in sequential programs. Our work can improve the affect of Sherlog by assuring sufficient data in the log messages.

In addition to runtime logs, core dumps are an important source of information for debugging. PSE [22] and ESD [23] carry out offline bug diagnoses using the core dump of a crashed program, while [15] aims at diagnosing Heisen-

[4] sn: Sebastian please Add description of table here

	Application	Concurrency	Count	Mean	Median	SEM	CI-95	Min	Max
1	Linked-List	Nested-Locks	5	11.95	13.91	1.44	2.00	8.43	14.94
2	Linked-List	Non-Blocking	5	4547.98	5502.49	598.98	830.13	3079.39	5539.07
3	Linked-List	Obstruction-Free	5	2628.86	2676.52	40.68	56.38	2528.89	2707.92
4	Linked-List	Transactional	5	251.59	258.05	4.26	5.91	241.07	258.90
5	Red-Black-Tree	Nested-Locks	20	3.95	3.95	0.00	0.00	3.95	3.95
6	Red-Black-Tree	Transactional	10	4.46	4.46	0.00	0.00	4.46	4.46

Table II
DETAILED NUMBERS FOR SLEEP=100 NAME=INST ITER=200

bugs by comparing core dumps from different crashes of a program.

An alternative to logging is profiling. In [24], they use sampling to collect profiling data while [25] collects profiling data from the hardware level. In both cases, the information they collect are not causality-related, therefore some of the data collected are not relevant to a crash. Our method can guide them in collecting data related to a specific bug.

VII. 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 exctractable. Since the complexity of our optimization problem is exponential, we encoded the problem as a Boolean satisfiability problem (SAT) to leverage powerful SAT-solvers to tackle our problem.

In our tool chain, we used LLVM and a slicing algoritm 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 genrate 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 instrumented programs using our method as compared to ad-hoc over-instrumented programs.

For future work, we are considering two main research directions: (1) tecniques for solving the optimization problem more effiecently, and (2) extending the concept of observability to other domains. For the former, we are currently working on using lightweight model checking and our own heuristics to compute slices in concurrent programs more precisely by taking inter-thread scynchronizations into acccount. These techniques 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.

REFERENCES

[1] S. Navabpour, B. Bonakdarpour, and S. Fischmeister, "Software debugging and testing using the abstract diagnosis theory," in *Languages, compilers, and tools for embedded systems (LCTES)*, 2011, pp. 111–120.

[2] H. Thane and H. Hansson, "Towards Systematic Testing of Distributed Real-Time Systems," in *Proceedings of the 20th IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 1999, pp. 360–369.

[3] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson, "Replay Debugging of Real-Time Systems Using Time Machines," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 8.

[4] U. Schmid, "Monitoring of Distributed Real-Time Systems," *Real-Time Systems*, vol. 7, no. 1, pp. 33–56, 1994.

[5] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis and transformation," in *CGO*, 2004, pp. 75–.

[6] P. Ammann and J. Offutt, *Introduction to Software Testing*. New York, USA: Cambridge University Press, 2008.

[7] "Yices: An SMT Solver," http://yices.csl.sri.com.

[8] J. Lee, S. P. Midkiff, and D. A. Padua, "A constant propagation algorithm for explicitly parallel programs," *International Journal of Parallel Programming*, vol. 26, no. 5, pp. 563–589, 1998.

[9] 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*, vol. 13, no. 4, pp. 451–490, 1991.

[10] M. G. Nanda and S. Ramesh, "Slicing concurrent programs," in *ISSTA*, 2000, pp. 180–190.

[11] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," in *SDE*, 1984, pp. 177–184.

[12] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving Software Diagnosability via Log Enhancement," in *ASPLOS*, 2011, pp. 3–14.

[13] M. Musuvathi and S. Qadeer, "Iterative Context Bounding for Systematic Testing of Multithreaded Programs," in *PLDI*, 2007, pp. 446–455.

[14] S. Park, S. Lu, and Y. Zhou, "Ctrigger: Exposing Atomicity Violation Bugs from Their Hiding Place," in *ASPLOS*, 2009, pp. 25–36.

[15] D. Weeratunge, X. Zhang, and S. Jagannathan, "Analyzing Multicore Dumps to Facilitate Concurrency Bug Reproduction," in *ASPLOS*, 2010, pp. 155–166.

[16] R. H. B. Netzer and M. H. Weaver, "Optimal Tracing and Incremental Reexecution for Debugging Long-Running Programs," in *PLDI*, 1994, pp. 313–325.

[17] D. Z. Pan and M. A. Linton, "Supporting Reverse Execution for Parallel Programs," in *SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, 1988, pp. 124–129.

[18] M. Ronsse, K. D. Bosschere, M. Christiaens, J. C. D. Kergonneaux, and D. Kranzlmuller, "Record/Replay for Nondeterministic Program Executions," *Communication of the ACM*, vol. 46, no. 9, pp. 62–67, 2003.

[19] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas, "Capo: A Software-Hardware Interface for Practical Deterministic Multiprocessor Replay," in *ASPLOS*, 2009, pp. 73–84.

[20] 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*, 2004, pp. 29–44.

[21] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: Error Diagnosis by Connecting Clues from Run-Time Logs," in *ASPLOS*, 2010, pp. 143–154.

[22] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang, "PSE: Explaining Program Failures via Postmortem Static Analysis," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 6, pp. 63–72, 2004.

[23] C. Zamfir and G. Candea, "Execution Synthesis: A Technique for Automated Software Debugging," in *EuroSys*, 2010, pp. 321–334.

[24] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug Isolation via Remote Program Sampling," in *PLDI*, 2003, pp. 141–154.

[25] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Biggons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, "Flexible Hardware Acceleration for Instruction-Grain Program Monitoring," in *ISCA*, 2008, pp. 377–388.