Sampling-based Program Execution Monitoring

Sebastian Fischmeister
Department of Electrical and Computer Engineering
University of Waterloo
sfischme@uwaterloo.ca

Yanmeng Ba
Department of Electrical and Computer Engineering
University of Waterloo
yba@uwaterloo.ca

Abstract

For its high overall cost during product development, program debugging is an important aspect of system development. Debugging is a hard and complex activity, especially in time-sensitive systems which have limited resources and demanding timing constraints.

System tracing is a frequently used technique for debugging embedded systems. A specific use of system tracing is to monitor and debug control-flow problems in programs. However, it is difficult to implement because of the potentially high overhead it might introduce to the system and the changes which can occur to the system behavior due to tracing.

To solve the above problems, in this work, we present a sampling-based approach to execution monitoring which specifically helps developers debug time-sensitive systems such as real-time applications. We build the system model and propose three theorems to determine the sampling period in different scenarios. We also design seven heuristics and an instrumentation framework to extend the sampling period which can reduce the monitoring overhead and achieve an optimal tradeoff between accuracy and overhead introduced by instrumentation. Using this monitoring framework, we can use the information extracted through sampling to reconstruct the system state and execution paths to locate the deviation.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—tracing

General Terms Theory, Algorithms, Experimentation

Keywords sampling, monitoring, tracing, debugging, embedded system

1. Introduction

Debugging is an important procedure in embedded software development, because between 30 to 50 percent of the development cost is spent on testing and debugging (Bouyssounouse and J.Sifakis 2005; Gallaher and Kropp 2002). We define a software defect as "An incorrect step, process, data definition or result." (IEE 1990). While testing is the process of revealing failure by showing the presence of software defects, debugging starts after testing and is the process of locating and removing the defects. Because of the increase of system and software complexity, good debugging methods for embedded software become increasingly important. Software instrumentation is a popular technique for software-based debugging and tracing. System monitoring can effectively detect faults of programs running in the system. However, both instrumentation and monitoring introduce overhead to the execution of the program.

Our approach to bound the cost of monitoring is to sample the program at a fixed rate to collect adequate and necessary information about the execution at runtime. Besides, we develop several algorithms to strategically insert markers to the program to reduce the overhead introduced by the instrumentation. We thus propose a sampling-based monitoring mechanism which allows the developer to extract information from the system under test to determine the execution path of programs with an optimal tradeoff between accuracy and overhead. The execution trace contains the execution path of the last run and helps to find the location of software defects. Our work targets embedded, time-sensitive systems, and provides a framework for reconstructing system state and execution paths. A typical debugging session in our work flow consists of the following steps:

1. **Detect the presence of software defect.** Using traditional testing methods such as unit tests (Elbaum et al. 2009) or white/blackbox tests (Kirani et al. 1994), the developer determines that the tested software deviates from its specification.
2. **Budget resources for monitoring.** A good practice when developing embedded systems is to define a specific budget for debugging overhead. The available budget determines the monitoring precision as it affects the sampling period.
3. **Prepare program for monitoring.** The developer uses our algorithms to instrument the program with markers for sampling-based monitoring.
4. **Collect traces.** The developer runs the instrumented system. A monitor periodically collects data from the markers and other system state and transmits them to the developer workstation.
5. **Reconstruct execution.** Once the instrumented system crashes or the developer stops the execution, the framework permits reconstructing the execution path from the sequence of samples to locate the software defect in the code.

The advantages of our approach are as follows: Compared with continuous monitoring, our sampling-based technique can greatly reduce the overhead introduced. Since the sampling rate is fixed, we can estimate and bound the overhead and impact on the system under test. Also, our approach can easily be combined with data-tracing methods. There are, however, several issues to be solved regarding sampling-based approaches:

First, we need to balance between overhead and correct reconstruction of the control flow. On one hand, we want to gather enough information to be able to reconstruct the execution path;

on the other hand, monitoring should have only modest impact on the program. Second, we need to address the problem of how many markers should be used. Each marker requires memory, so we want to use as few markers are possible. Third, we need to devise an instrumentation algorithm that efficiently uses the available markers. Using markers well can reduce the number of required markers to achieve the target sampling period.

This paper makes several contributions to the issues mentioned above:

- We provide a formal framework that permits quantitative reasoning of many aspects involved in sampling-based mechanisms.
- We propose the optimality from both vertex and the whole control-flow graph perspectives.
- We provide theorems for termination conditions of instrumentation algorithms with an unlimited number of markers and with exactly one marker.
- We validate the general approach by proposing and comparing several algorithms for inserting markers into programs.
- We investigate interference among markers and propose tailored algorithms to compensate for this interference.
- We discuss a number of observations and insights from the development of the algorithms.

Besides debugging, sampling-based execution monitoring can also be used in performance profiling of software systems. Combined with tracing, it can give performance engineers a sufficiently detailed analysis of the system with relatively low overhead (Metz et al. 2005), such as event relationships in time and reconstructing the dynamic behavior of a software system. In addition, sampling-based execution monitoring can be applied to code coverage testing (Shye et al. 2005) which finds the code exercised by a particular set of test input. Moreover, sampling-based execution monitoring can be a feasible and efficient technique for reducing the overhead while collecting profile information (Lee and Zilles 2008).

In our paper, following a discussion about several traditional approaches, we firstly describe the problem targeted and then build our system model (Section 3) which served as the theoretical foundation for the whole paper. Secondly, we propose two theorems which provide the termination conditions of instrumentations in different scenarios and give the corresponding examples to explain them (Section 4). We then proceed and propose a BFS-based algorithm which calculates the sampling period (Section 5). We analyze the interference between instrumentations in Section 6. With the interference and the proper model, we proceed to experiment with different algorithms (Section 7) and interpret the results (Section 8). By drawing conclusions we close the paper in Section 9 and outline future work.

2. Related Works

Debugging techniques have been around since the early programming stages and come in different flavors. In hardware-based debugging, boundary scan testing (also named as JTAG) (Chun and Lim 2005) targets systems with limited resources and space for debugging. This method provides the ability to run the program and halt it at any given time which is undesirable in real-time system because of the timing constraints. With the higher integration of SoC and the increase of software complexity, logic analyzers become expensive for debugging, because of the high frequency and the increased number of ports required by large systems. ARM CoreSight is an On-chip debug technology for streaming data off the chip (Orme 2008). However, it suffers from limited bandwidth on the trace port (e.g., not all program counter values can be streamed off chip) and is only available on a limited number of chips.

Instrumentation is usually used to collect program profile and run-time information for various testing/debugging and analysis applications, such as detecting program invariants, dynamic slicing and alias analysis (Kumar et al. 2005), to monitor and track the program behavior (Biberstein et al. 2005). However, it introduces considerable overhead to the execution of the program. Researchers have proposed several methods to reduce the cost of instrumentation overhead (Kumar et al. 2005; Arnold and Ryder 2001; Misurda et al. 2005). Software instrumentation collects information of program execution by inserting instrumentation statements which might print out program location or variable values (Titzer and Palsberg 2005). Instrumenting *printf* statements is a naive approach which is tedious and inflexible. It might also result in "probe effect".

Software failure is expensive (IEE 2005) and system monitoring is one method to detect faults before they become failures (Jiang et al. 2009). There are some different flavors of system monitoring. Metric-correlation models use correlations among management metrics in software systems to detect bugs and localize their cause (Jiang et al. 2009). By posing queries, a software-based framework was proposed to monitor the state and performance of running programs (Cheung and Madden 2008). Structural monitoring determines which program entities, such as statements, branches and data-flow or control-flow relationships, are executed (Santelices and Harrold 2007). Previous research has developed efficient monitoring methods for states and branches (Ball and Larus 1992, 1994) which insert various counters into the program executed. An efficient branch-based monitoring approach is DUA (definition-use associations) (Frankl and Weyuker 1988; Hutchins et al. 1994), which results in additional overhead (Santelices and Harrold 2007).

One use for software instrumentation is to monitor control-flow. From the perspective of when to instrument the program, there are usually two types of software instrumentations: the static instrumentation insert the instrumentation code to the program before it executes, while the dynamic instrumentation instruments the program when it is running. The most commonly used static binary instrumentation tool is ATOM (Srivastava and Eustace 2004), which was implemented by extending OM and provides a framework for building a customized program analysis tools. For dynamic binary instrumentation, Pin provided by Intel is a valid option. Pin (Luk et al. 2005) follows the model of ATOM, but it does not instrument an executable statically by rewriting it, but rather adds the code dynamically while the executable is running. However, by instrumenting the executable with extra code, these software instrumentation methods might change the timing of the execution of the program unexpectedly and unpredictably, thus they are not soundly applicable to the real-time systems where timing has the top priority. Related work investigated software-support perspectives (Ball and Larus 1994) and hardware-based approach (Zhang et al. 2005). Meanwhile, monitoring control-flow is especially expensive, and there is little work done so far to characterize or bound its cost.

There are already several works that apply the concept of sampling into program debugging: using random sampling in statistical debugging to isolate bugs (Liblit et al. 2005); debugging programs given sampled data from thousands of user runs (Zheng et al. 2006); a sampling infrastructure for gathering information from a large number of executions (Liblit et al. 2003). These works focused on using the sampling concept to gather run-time information from program executions in workstation softwares.

3. System Model & Terminology

This work concentrates on multi-process single-threaded applications like the ones found in background/foreground systems. This structure dominates the embedded software domain due to its maintainable structure and efficient resource utilization (Labrosse 2002; Fischmeister and Lee 2007). Note that about 85 percent of all embedded systems use 8-bit or smaller architectures (Tennenhouse 2000).

We also assume that the system supports interrupts and has at least one high-precision timer as commonly found in microcontrollers. For example, the ATmega128 microcontroller has four timers.

3.1 Model Definition and Terminology

To analyze and reconstruct the execution path of the application, we convert a source program to a directed graph, representing the program's control flow, which is defined as the control-flow graph $G = \langle V, E \rangle$.

In G, each vertex ($v \in V$) represents a basic code block in a program. The entry vertex v_{en} is the start of the program. The exit vertex v_{ex} is the termination of the program. Edge $e := \langle v_s, v_d \rangle$ represents the specific transition from a source vertex v_s to a destination vertex v_d. It assumes that G is an *unweighed graph* with $e := \langle v_s, v_d \rangle = 0$, which means that the transition between two vertices requires no time.

We define the function $c : V \to \mathbb{N}$, which specifies the required execution time for a vertex v. For example, $c(v_0) = 10$ means that the basic code block at vertex v_0 requires 10 time units for its execution.

A path p is defined as a sequence of adjacent vertices $v_i \to v_{i+1} \to \ldots \to v_k$. The execution time of a path is the sum of the execution times of all vertices and is defined as $c_p(p) = \sum c(v_i)$ for all $v_i \in p$. An execution path r, which is a special p, is the actual path executed from the entry vertex v_{en} to the exit vertex v_{ex}.

Our approach periodically takes samples from the execution information and program state. In this context, we define a sample as a triple $s := \langle state, v, t \rangle$ where v represents the vertex sampled, t represents the time stamp when the sample is taken and $state$ represents the program state(e.g. the values of some variables) at that time stamp. The sampling period T is defined as the constant time interval Δt between two adjacent samples, that is, $T = \Delta t = t_{i+1} - t_i$ for two adjacent samples $s_i := \langle state_i, v_i, t_i \rangle$ and $s_{i+1} := \langle state_{i+1}, v_{i+1}, t_{i+1} \rangle$.

Furthermore, to evaluate the quality of the sampling period, we define the function $\mathrm{pathfind_t}(v_i, v_j, \Delta t)$ with $\Delta t = t_j - t_i$ returning all possible paths between two vertices while Δt represents the execution time interval between v_i and v_j. We define the sampling period as *too long*, if multiple paths exist between two vertices of two samples, which is indicated by $|\mathrm{pathfind}(v_i, v_j, \Delta t)| > 1$,where $v_i, v_j \in V$. We define a sampling period as *sufficient*, if only one path exists between two vertices.

The concept of *optimality* for the sampling period is formed with respect to both a vertex and a complete control-flow graph. If a sampling period of T is sufficient and a sampling period of $T + \epsilon$ is too long, then T is the optimal sampling period for the the starting vertex in the given control-flow graph. In other words: sampling after T permits only one path between the two samples and $T + \epsilon$ permits multiple. Stating this formally, starting from a specific node v_i, the sampling period T is said to be *optimal* for the node v_i, if $|\mathrm{pathfind_t}(v_i, v_{next}, T)| = 1$ while $\left|\mathrm{pathfind_t}(v_i, v_{next}^{'}, T + \epsilon)\right| > 1$.

For the whole control-flow graph $G = (V, E)$, the *optimal* sampling period is the minimum of the *optimal* sampling periods of vertices in the control-flow graph. Thus, the *optimal* sampling period T_{opt} is defined as $T_{opt} = min(T_1, \ldots, T_k)$ where T_i is the *optimal* sampling period for $v_i \in V$ with $V = (v_1, \ldots, v_k)$.

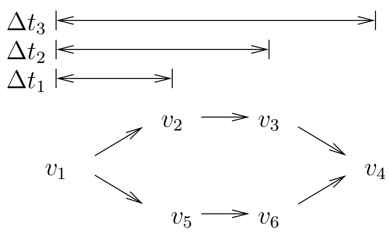

Figure 1. Different sampling periods for one control flow

Example 1. *Figure 1 shows an example of a control-flow graph, a starting vertex v_1, and several sampling periods $\Delta t_1 = 1$, $\Delta t_2 = 2$, and $\Delta t_3 = 3$. All basic blocks have the same execution time $c(v_i) = 1$. From our definitions, the sampling period Δt_1 is sufficient, since $|\mathrm{pathfind_t}(v_1, v_2, \Delta t_1)| = |\mathrm{pathfind_t}(v_1, v_5, \Delta t_1)| = 1$;$\Delta t_2$ is optimal, since $|\mathrm{pathfind_t}(v_1, v_3, \Delta t_2)| = 1$ while $|\mathrm{pathfind_t}(v_1, v_4, \Delta t_2 + 1)| = 2$; Δt_3 is too long, since $|\mathrm{pathfind_t}(v_1, v_4, \Delta t_3)| = 2$.*

3.2 Markers

To increase the sampling period which will reduce the monitoring overhead, we introduce the concept of markers and extend a sample with state information. A marker can be a system element such as the program counter, very useful, because a vertex in the control-flow graph is a basic block in the source code. Besides, a marker can also be a newly introduced variable solely used for the purpose of monitoring the software.For the remainder of this paper, we will only use this extended sample and thus $s := \langle state, v, t \rangle$ where $state$ is also a tuple defined as $state := \langle m_1, \ldots, m_k \rangle$ with m_i representing a marker of a system state such as memory, processor word, I/O registers, or our introduced variables.

We thus refine the pathfind function as $\mathrm{pathfind_s}(v_i, v_j, state_i, state_j, \Delta t)$ where $state_i$ and $state_j$ are the state elements of the corresponding samples. Using the function $\mathrm{pathfind_s}$, a sampling period T is *optimal*, if $|\mathrm{pathfind_s}(v_i, v_{next}, state_i, state_{next}, T_i)| = 1$ while $\left|\mathrm{pathfind_s}(v_i, v_{next}^{'}, state_i, state_{next}^{'}, T_i + \epsilon)\right| > 1$.

As stated above, markers are special variables that can be used for extending the optimal sampling period.We introduce such new markers and increment their values at strategically well-placed locations. We give the following example to show how the markers work.

Example 2. *Figure 2 shows a program control flow. All basic blocks have the same execution time $c(v_i) = 1$.*

Without introducing a monitoring variable a, we use function $\mathrm{pathfind_t}(v_i, v_j, \Delta t)$ to find the optimal *sampling period. Starting from vertex v_1, there are three possible paths afterwards. If we take the sample after time 1, then $|\mathrm{pathfind_t}(v_1, v_i, 1)| = 1$ with $i = 2, 3, 4$. However, if we take the sample after time 2, then $|\mathrm{pathfind_t}(v_1, v_5, 2)| = 2$. Figure 3 shows this mechanism. Thus, the* optimal *sampling period for node v_1 is $T_1 = 1$. Applying the same mechanism, for every other vertex v_i with $i = 2, 3, 4, 5$ in the control-flow graph, the* optimal *sampling period T_i with $i = 2, 3, 4, 5$ is $4, 3, 3, 2$ respectively. Thus, for the whole control-flow graph $G = \langle V, E \rangle$, the* optimal *sampling period T_{opt} is 1.*

Using the monitoring variable a, Figure 4 shows the resulting optimal sampling period. We will use function $\mathrm{pathfind_s}(v_i, v_j, state_i, state_j, \Delta t)$ to select the optimal *sam-*

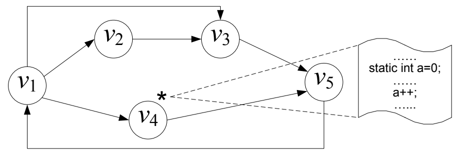

Figure 2. Control-flow graph with marker instrumented

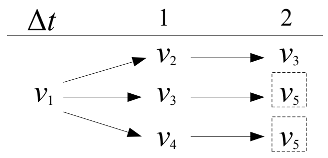

Figure 3. $\mathrm{pathfind_t}(v_i, v_j, \Delta t)$

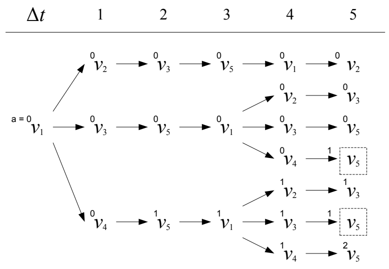

Figure 4. $\mathrm{pathfind_s}(v_i, v_j, state_i, state_j, \Delta t)$

pling period. Starting from vertex v_1, the mechanism is shown in Figure 4. While $|\mathrm{pathfind_s}(v_1, v_i, state_1, state_i, 4)| = 1$ with $i = 2, 3, 4, 5$ and state $:= \langle a \rangle$, $|\mathrm{pathfind_s}(v_1, v_5, state_1, state_5, 5)| = 2$. Thus, for vertex v_1, the optimal *sampling period is 4. Applying the same mechanism, for every other vertex v_i with $i = 2, 3, 4, 5$ in the control-flow graph, the* optimal *sampling period T_i with $i = 2, 3, 4, 5$ is $7, 6, 6, 5$ respectively. Thus, for the whole control-flow graph $G = \langle V, E \rangle$, the* optimal *sampling period T_{opt} is 4. Compared with the previous example, introducing marker a increases the* optimal *sampling period T_{opt} by a factor of 4.*

4. Theoretic Optimum

By using markers we can increase the sampling period without losing any essential information about the execution paths. We call the process of inserting such markers into vertices instrumentation.

An important problem is to understand the limitations of such marker-based instrumentation. We therefore provide theorems to find the theoretic sampling period and decide termination conditions.

We require two additional definitions for the theorems. A pathpair *pp* can be defined as two paths which have the same entrance vertex and the same exit vertex with their exit vertices overlapping in time, but no other vertices between the entrance vertex and the exit vertex overlap in time. That is, $|\mathrm{pathfind_s}(v_{en}, v_{ex}, state_{en}, state_{ex}, \Delta t_{ex})| = 2$ while $|\mathrm{pathfind_s}(v_{en}, v_{next}, state_{en}, state_{next}, \Delta t_{next})| = 1$, where $0 < \Delta t_{next} < \Delta t_{ex}$. A path set is defined as a set of paths of which any two paths constitute a pathpair as defined above.

Lemma 1. *In a pathpair pp, a path which starts from the entrance vertex v_{en} and ends at any other vertex except the exit vertex v_{ex} is unique. Formally, $|\mathrm{pathfind_s}(v_{en}, v_{next}, state_{en}, state_{next}, \Delta t)| = 1$, where $v_{next} \in V_{pp}$ and $v_{next} \neq v_{ex}$.*

From the definition of a pathpair, we can draw the following conclusion:

Lemma 2. *(Optimal Vertex Sampling Period)*

In a control-flow graph, all pathpairs starting at vertex v_i constitute a vector $PP_{v_i}\langle pp_1, pp_2, \ldots, pp_k, \ldots, pp_m \rangle$, with each pathpair starting at time t_{ien} and ends at time t_{kex}, with $k = 1, 2, \ldots, m$. The optimal sampling period of this v_i is defined as $T_{opt_i} = \min |(t_{kex} - t_{ien})| - \epsilon$, with $k = 1, 2, \ldots, m$.

Therefore, to calculate the theoretic optimal sampling period, it is essential to find pathpairs for every vertex in the control-flow graph. We propose the following approach to find pathpairs starting from v_i: we construct an array of vertex states ordered by time; starting from v_i, we search v_i's child vertices and their corresponding states; then, we compare the state of child vertex v_j with that of v_k in the state array. If they meet the pathpair conditions, we will say that the path which starts from v_i and ends at v_j and the path which starts from v_i and ends at v_k constitute a pathpair; if the conditions are not met, we will treat that child vertex as a father vertex, add it to the state array after sorting and continue to search for pathpair.

Theorem 1 (Optimal Sampling Period)**.** *For a control-flow graph with N vertices, the optimal sampling period of the whole graph is the minimum sampling period of all sampling periods for all vertices. Formally, $T_{opt} = min(T_{opt_1}, \ldots, T_{opt_N})$.*

By choosing a proper strategy to find the vertices which are instrumented with markers and thus making the states of overlapping exit vertices different, we can extend the pathpair and therefore increase the sampling period. However, the number of markers to instrument with is limited. With increasing the number of markers, the following situation would occur: After the number of the markers reaches a certain value, no matter how to instrument the vertices with markers, we can not extend pathpair any further. In other words, we cannot increase the sampling period any more. In this situation, regardless how many markers are available, we can no longer distinguish the two paths in the pathpair. Obviously, we should terminate the instrumentation process at this point. We propose the following theorem to draw this termination condition:

Theorem 2 (Pathpair Termination)**.** *For two paths p_1 and p_2 in a pathpair, if they meet the following conditions:*

- *they have the same vertices with the same number of appearances but possibly different order in time respectively*
- *the states of the corresponding vertices are the same*

we can no longer instrument vertices with markers to differentiate these two paths, thus reach the theoretical optimum sampling period for this pathpair.

Proof. We use "reductio ad absurdum" to prove our theorem. Suppose that when reaching a pathpair with its two paths (p_1 and p_2) violating the above conditions in Theorem 2. For example, the two paths have different vertices between them or the two paths have exactly the same vertices but the numbers of their appearances in these two paths differ. We assume that the sampling period $T_{falseopt}$ we get here is the theoretic optimum sampling period. However, if we instrument the distinct vertices of the two paths or the identical vertices having different numbers of appearances in two paths with markers, we can still extend this pathpair and form a new pathpair whose sampling period is larger than $T_{falseopt}$. This contradicts the assumption that $T_{falseopt}$ is the theoretic optimum sampling period, because we can still distinguish these two paths. In this way, we can prove that our theorem is correct. □

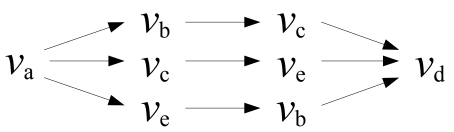
$$\begin{tikzcd}
 & v_b \arrow[r] & v_c \arrow[dr] & \\
v_a \arrow[ur] \arrow[r] \arrow[dr] & v_c \arrow[r] & v_e \arrow[r] & v_d \\
 & v_e \arrow[r] & v_b \arrow[ur] &
\end{tikzcd}$$

Figure 5. Scenario when theoretical optimum is reached using only one marker

When a pathpair satisfies the condition in Theorem 2, we can terminate the instrumentation process since we can no longer distinguish the two paths through instrumenting vertices with markers.

Theorem 3 (Single Marker Termination)**.** *The instrumentation of a path set with a single marker is essentially an SAT problem.*

Proof. For a path set of n paths, there are $N = C_n^2$ pathpairs which share the same entrance and exit vertices. For a pathpair pp_k, all the vertices except the entrance and exit vertices constitute an internal vertices set Ω_k. In Ω_k, all vertices that can be used to instrument with markers constitute the set Φ_k, with all the other vertices which can not be instrumented constituting the set Ψ_k. Apparently, $\Omega_k = \Phi_k + \Psi_k$. When $cond_k = (v_{k1} \vee v_{k2} \vee, \ldots, \vee v_{kj}) \wedge (\overline{v_{kj+1}} \vee \overline{v_{kj+2}} \vee, \ldots, \vee \overline{v_{kj+m}})$,with $\Phi_k = \{v_{k1}, \ldots, v_{kj}\}$ and $\Psi_k = \{v_{kj+1}, \ldots, v_{kj+m}\}$, is satisfiable, we can distinguish the two paths in pathpair k.

The theoretical optimum for a graph using only one marker is reached, when we get to a path set, where $\Upsilon = cond_1 \wedge cond_2 \wedge \ldots \wedge cond_N$ can never be satisfied. □

Example 3. *As shown in Figure 5 we get to a path set $S = \{p_1, p_2, p_3\}$ with $p_1 = v_a \to v_b \to v_c \to v_d, p_2 = v_a \to v_c \to v_e \to v_d$ and $p_3 = v_a \to v_e \to v_b \to v_d$. For pathpair $pp_{12} = \{p_1, p_2\}$, the two paths p_1 and p_2 can be distinguished using only one marker, because $cond_{12} = (v_b \vee v_e) \wedge \overline{v_c}$ is satisfiable. So do pathpairs $pp_{13} = \{p_1, p_3\}$ and $pp_{23} = \{p_2, p_3\}$, with $cond_{13} = (v_c \vee v_e) \wedge \overline{v_b}$ and $cond_{23} = (v_b \vee v_c) \wedge \overline{v_e}$ satisfiable. However, $cond_{12}$, $cond_{13}$ and $cond_{23}$ can not be satisfied at the same time using only one marker, in other words, $\Upsilon = cond_{12} \wedge cond_{13} \wedge cond_{23}$ can never be satisfied. At this point, we reach the theoretic optimum using only one marker.*

5. Calculating the Sampling Period

From Theorem 3, we know that calculating the theoretical optimum for a path set using one marker is actually an NP-hard problem. However, in practice, we encounter pathpairs much more often than path set as the likelihood is quite small for three or more paths to have the same entry and exit vertices with the same time span. Thus, it is both practical and important to develop an algorithm that has a polynomial runtime complexity to calculate the sampling period for pathpairs.

Given a control-flow graph $G = \langle V, E \rangle$, to calculate the optimal sampling period for a vertex in the pathpair, we propose the following algorithm based on the breath-first-search(BFS) (Cormen et al. 2001) to implement $pathfind_s$.

This algorithm is based on breadth-first search (BFS). Firstly, we pick a starting vertex $vertex$ by setting its $state$ as OPEN. We also build a set V_{open} which contains all vertices that are adjacent to vertex $vertex$ and set it to OPEN as well. In set V_{open}, we choose the vertex which has the least execution time t_{min} as the next starting vertex v_{next} to move to. At the same time, we update the sampling period by increasing it by t_{min} and the execution time of all vertices in set V_{open} by decrementing them by t_{min}. We build another set V_{toopen} which contains all the vertices that are both adjacent to and reachable from v_{next}. At last, we check

```
Vertex $v := \langle state, time \rangle$,
Edge $e := \langle v_{src}, v_{dst}, cond, updates \rangle$
**for** all $v \in V$ **do**
    $v.state \Leftarrow$ CLOSED
**end for**
$v_{en}.state \Leftarrow$ OPEN
$tResult \Leftarrow 0$
$V_{toopen} \Leftarrow \{\}$
$V_{open} \Leftarrow \{\}$
**loop**
    **if** $V_{open}$ is empty **then**
        **for** all $v \in V_{toopen}$ **do**
            $v.state \Leftarrow$ OPEN
        **end for**
        $V_{open} \bigcup \{v | v \in V and v.state =$ OPEN$\}$
        $V_{toopen} \Leftarrow \{\}$
    **end if**
    $t_{min} \Leftarrow min(v.time)$ of all $v \in V_{open}$
    $v_{next} \Leftarrow v$ where $v \in V_{open}$ with $v.time = t_{min}$
    $tResult \Leftarrow tResult + t_{min}$
    **for** all $v$ in $V_{open}$ **do**
        $v.time \Leftarrow v.time - t_{min}$
    **end for**
    **for** all $e \in E$ with $e.v_{src} = v_{next}$ and $eval(e.cond) = T$ **do**
        **if** $e.v_{dst}.state =$ OPEN and $e.v_{dst}.time > 0$ **then**
            break from $loop$
        **else**
            create state for $e.v_{dst}$ and execute updates (e.updates) on this
            state
            $V_{toopen} \Leftarrow V_{toopen} \bigcup e.v_{dst}$
        **end if**
    **end for**
    $v_{next}.state \Leftarrow$ CLOSED
**end loop**
**return** $tResult - 1$
```

Algorithm 1: Find optimal sampling period for a vertex

the set V_{toopen}. If it contains a vertex whose $state$ is OPEN and execution time is greater than zero, we say the optimum sampling period for that vertex is reached and return the current sampling period as *optimum*. If not, we repeat the above procedure until we reach the *optimum* conditions stated above.

Since the algorithm uses BFS, the runtime complexity for our algorithm is $O(|V| + |E|)$.

6. Instrumenting Control Flows

As stated above, in order to increase the sampling period, we introduce markers into the control-flow graph. In this section, we present our instrumentation approaches, analyze the related issues caused by the instrumentation and give our strategies to resolve these issues.

6.1 Increment VS Assignment

Instrumentation algorithms can use markers in different ways. One method is to increment the value of the marker each time the marker is hit. The other assigns absolute number to the marker. We provide the following two examples to prove that neither of these two options is better than the other.

Figure 5 shows a path set. By adding the marker with a different assignment to the last vertex before the exit vertex (e.g., $v_c \leftarrow a = 1$, $v_e \leftarrow a = 2$ and $v_b \leftarrow a = 3$) in each path, we can distinguish these three paths in a sample taking at v_d. However, according to Theorem 3, we cannot distinguish these three paths by increment-based marker methods. Thus, the assignment-based marker method can instrument cases that the increment-based one cannot.

Figure 6 shows another case. We can instrument v_4 or v_7 with increment-based markers and distinguish the two paths in pathpair

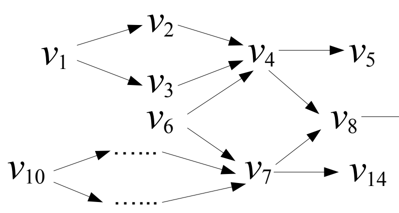

Figure 6. Scenario where increment works but not assignment

$pp_{(6,8)}$. However, since v_4 and v_7 are also the exit vertices of two other pathpairs $pp_{(1,4)}$ and $pp_{(10,7)}$, instrumenting these two vertices by assignments renders any previous instrumentations invalid, because several paths will share the same marker value and can invalidate another instrumentation at a later point. As shown, if we use increment instead, we will be able to solve this problem and distinguish the paths. Thus, the increment-based method can instrument cases that the assignment-based method cannot.

Since each method can instrument at least one case that the other cannot instrument, both methods have their justification as they can instrument different sets of control-flow problems.

6.2 Interference

Greedy instrumentation suffers from the problem that the instrumentation at a subsequent step may influence the instrumentations of previous steps. This can happen in either a direct or an indirect way. In *direct interference*, the subsequent instrumentation adds a marker to a vertex which is already part of a previous pathpair or path set and this breaks the original instrumentation for that particular pathpair. In *indirect interference*, the subsequent instrumentation adds a marker to a vertex which reveals a new pathpair with a shorter time span. The following two examples show these effects.

Example 4 (Direct interference)**.** *We assume a control-flow graph, a greedy instrumentation strategy with only one available marker, and that each vertex has an execution time of one time step. Figure 7(a) shows the initial state of a pathpair $pp_{(1,4)}$. Any greedy strategy will pick either v_2 or v_5 to instrument with a marker. Here we assume that the strategy picks v_2 as shown in Figure 7(b).*

v_5 is also part of another pathpair $pp_{(6,11)}$ with a longer time span as shown in Figure 7(c). In this example, there will be a problem, if the greedy strategy picks v_5 in the subsequent instrumentation step instead of v_{10}, v_7, or v_8. By instrumenting v_5, it breaks the original instrumentation for pathpair $pp_{(1,4)}$, since both paths in pathpair $pp_{(1,4)}$ are then instrumented and can not be distinguished from each other.

Example 5 (Indirect interference)**.** *We make the same assumptions as that in Example 4. As shown in Figure 8, the greedy algorithm first discovers the pathpair $pp_{(1,4)}$ and instruments v_5 to distinguish the two paths. By instrumenting v_5, the greedy algorithm also distinguishes the pathpair $pp_{(6,11)}$, so the algorithm will not notice it—we now call it hidden—and instead see the pathpair $pp_{(12,14)}$ as the next pathpair with the shortest time span. If the algorithm now instruments v_8, the hidden pathpair will cause a decrease in the sampling period.*

While a greedy algorithm can eliminate direct interference—see our SAT-based algorithms—eliminating indirect interference is hard, because it requires the algorithm to search for hidden pathpairs with all possible marker configurations.

6.3 Algorithms

We design seven different greedy algorithms (strategies) to find a suitable instrumentation. In our algorithms, the potential candidate

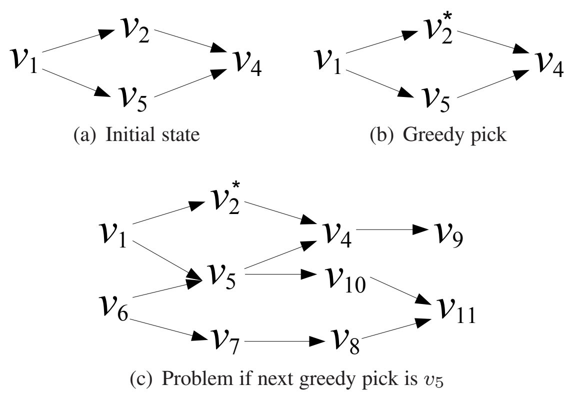

Figure 7. The problem of interference during greedy instrumentation

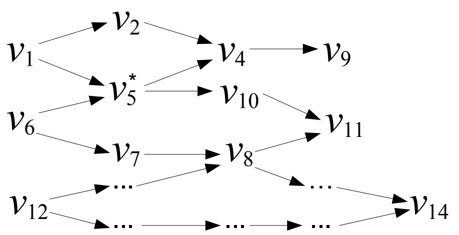

Figure 8. The indirect interference caused by greedy pick

vertex to instrument with markers to distinguish the two paths in a pathpair is the one that is distinct in either of the two paths or is contained in both paths but has different numbers of appearances.

The seven strategies can be divided into three categories:

- Degree-Based: The algorithms make decisions based upon the largest or smallest sum of in-degree and/or out-degree of the vertices in the context of the whole directed graph.
- Frequency-Based: The algorithms make decisions based upon the the occurrence frequency in the pathpair, such as the most or least frequently occurring vertex.
- SAT-Based: The algorithms transform the instrumentation into an SAT problem and compute a solution to find the instrumentation. The weighted SAT algorithm tries to combine the frequency-based with the SAT-based ideas.

7. Experimental Method

To validate the theorems and the concepts of this work, we build an instrumentation engine that instruments control flow graphs. The engine provides the framework to test different heuristics but also computes the theoretic optimum following Theorem 2. The outputs are the instrumentation vertices, the required execution time, and the resulting sampling period.

Our inputs are realistic and statistically significant. The input data consist of 5 000 control flow graphs which model typical C program flows (Thorup 1998). We generate these control flows with a customized version of Task Graphs For Free (Dick et al. 1998). Regarding the SAT-based heuristic, we implement it using a sat solver called SAT4J (09: 2009a) in our instrumentation engine. One experiment run works as follows: we select a control flow graph, a heuristic (or the optimum algorithm), and the number of available markers and pass these values to the instrumentation engine. The engine computes the input control-flow graph and

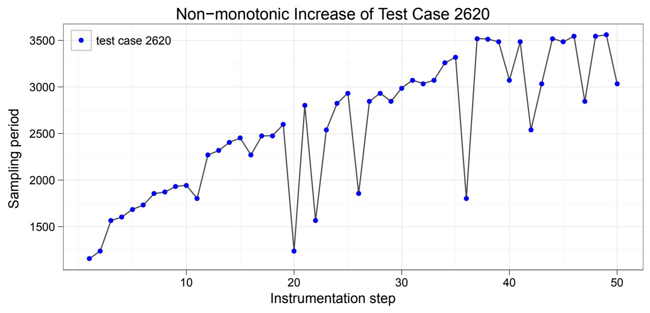

Figure 9. Example of interference of greedy instrumentation

returns the sampling period, vertex to instrument, and the required execution time. Since the computational work is quite intensive, we perform our simulation through the Canadian super computing cluster called SHARCNET (09: 2009b) collecting about 3.2 million instrumentation data points from up to 50 instrumentation steps, seven strategies, and several multi-marker configurations.

Our data successfully pass these integrity checks: (1) the execution time of the heuristic increases with the number of instrumentation steps, (2) no sampling period found by a heuristic is greater than the optimal sampling period, and (3) on average, the sampling period increases with the increase in the number of instrumentation steps.

The data distribution differs from a normal distribution (Shapiro-Wilk normality test for the data series varies around $p = 1^{-15}$). Thus, we rely on median values and testing procedures free of the normality assumption.

7.1 Instrumentation Performance Metric

To compare the performances of the algorithms, we take the maximum sampling period achieved in each run per algorithm and sum them up; see Eq. (1). This metric is robust against direct and indirect interference outlined in Section 6.2.

$$P = \sum max(T_i) \tag{1}$$

7.2 Monotonicity Metric

Interference in the greedy instrumentation algorithms has an unpleasant effect on the monotonicity of the algorithms: a subsequent instrumentation may decrease the sampling period. This is contrary to what the user expects as each instrumentation increases the overhead and thus should increment the sampling period. Figure 9 shows an example of this behavior.

We use the following monotonicity metric to evaluate the algorithms:

$$M = \frac{N}{\sum d_i} \tag{2}$$

with

$$d_i = \begin{cases} 0 & \text{if } run_i - run_{i+1} \leq 0, \\ run_i - run_{i+1} & \text{otherwise} \end{cases}$$

In the metric, we use d_i to denote the decrement between two instrumentation steps run_i and run_{i+1}, if the sampling period of one instrumentation step run_i is greater than that of its subsequent instrumentation step run_{i+1}. $\sum d_i$ denotes the sum of decrements in the entire instrumentation steps for a test case. N denotes the number of the total instrumentation steps. This monotonicity describes the reciprocal of average decrement across the entire process of instrumentation steps for a test case using a specific strategy and gives a general assessment of that strategy, since the decrement represents the interference introduced by instrumenting vertices with markers.

7.3 Execution time

We measure the execution time by comparing the time stamp when the execution of heuristic starts with the time stamp after the instrumentation step is completed. The sum of all these times for one run provides the total execution time for that run. While we will not be able to compare quantitative results, because of the heterogeneity of SHARCNET, we will draw conclusions based on similarity of the algorithms.

8. Results and Discussion

Following the experimental methods presented, we give our experiment results which are sound and show statistically integrity. We also discuss the results and give the corresponding interpretation.

8.1 Instrumentation Performance

We follow the recommended guidelines for multiple testing (Benjamini and Hochberg 1995). We check that all input data for calculating the performance metric have roughly the same shape (single bell-like shape with a cut-off left tail) for all algorithms. The instrumentation performance differs significantly among the algorithms (Kruskal-Wallis Rank Sum Test returns $p = 2^{-6}$). Using a Bonferroni correction for multiple testing among our algorithms, we test an individual algorithm with a $p \leq \frac{0.05}{91}$ to be accepted.

Figure 10 shows the result of the performance measurements for up to 50 instrumentation steps and compares it with the theoretic maximum achievable following Theorem 3. The higher the performance value, the better. For the single-marker algorithms—the right part of the figure—the degree-based algorithms outperform the others except the ‘max impact’ algorithm. We use the Wilcoxon rank sum test with continuity correction and it shows that the differences among the degree-based algorithms are insignificant while it shows a difference between all degree-based algorithms and the ‘min impact’ as well as the SAT-based algorithms. An interesting point is that the SAT-based algorithms perform significantly worse than any of the other algorithms. Part of this is, because bad early decisions in the SAT algorithm cannot be undone by a later instrumentation. While, for example, the degree-based algorithms may break a previous instrumentation and causes direct interference, the SAT-based ones cannot do this, because they preserve all previous instrumentations.

Using multiple markers improves the performance and asymptotically approaches the optimum. The middle part of Figure 10 shows the ‘max impact’ algorithm with different markers. Since ‘max impact’ performs similarly to other degree-based algorithms, if we were using a different algorithm, it would result in the same data. The gains achieved with low marker increases are significant. However, once the number of markers grows beyond ten, the results no longer differ using the Wilcoxon rank sum test with the adjusted significance level.

8.2 Monotonicity

Besides instrumentation performance, we also investigate monotonicity by the monotonicity metric defined in Section **??**. Figure **??** shows the monotonicity of all heuristics normalized to SAT. The higher the monotonicity value, the better. The left part of the figure shows the results of using only one marker. We use the same statistical test procedures as mentioned above to establish statistical significance. The SAT-based algorithms clearly outperform all other algorithms. The reason is that the SAT-based heuristics always carry forward the previous pathpairs and thus guarantee that a

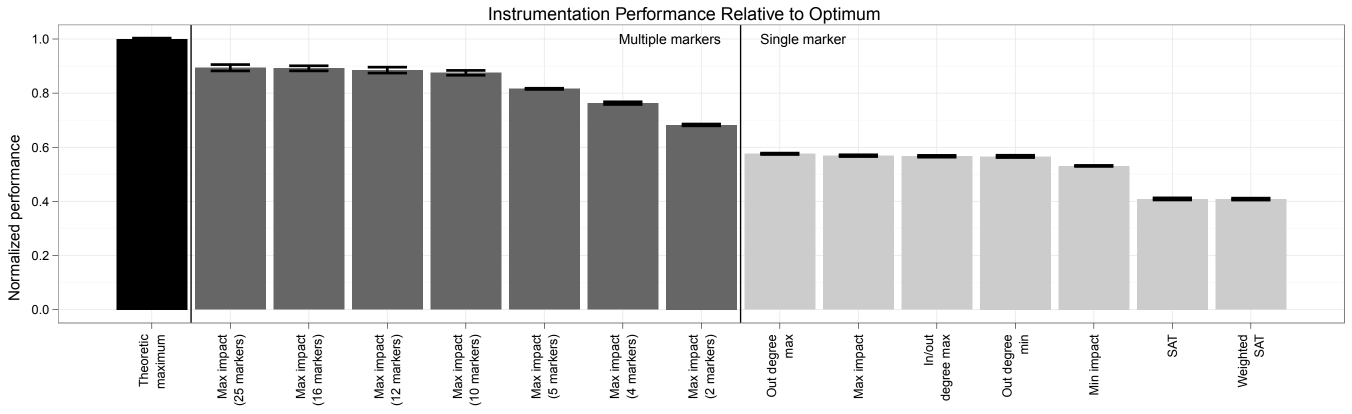

Figure 10. Instrumentation performance of all algorithms.

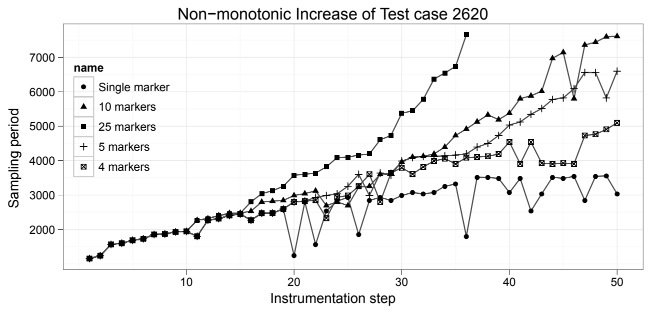

Figure 12. Improving monotonicity with multiple markers

subsequent instrumentation avoids interfering with a previous one. The remaining monotonicity only originates from indirect interference. We can also conclude that in general approximately 20% of the interference in the instrumentation is indirect interference while 80% is direct interference.

Using multiple markers, we try to: (1) increase the achieved sampling period, and (2) improve monotonicity. We try to increase the sampling period, because if one marker is no longer sufficient (Theorem 3), we can use another marker until we hit the optimum for pathpairs (Theorem 2). Figure 10 shows that we have achieved this. We also hoped to improve monotonicity by reducing the interference between subsequent instrumentation steps. Whenever we switch to a new marker, we avoid interfering with a previous instrumentation.The results are quite surprising.

The right part of Figure 11 shows monotonicity with multiple markers. While using multiple markers improves the monotonicity of the heuristics, the improvements are still rather limited, and at some point become insignificant in general. However, individual cases can benefit significantly, as Figure 12 shows. The SAT-based heuristics still shows a better overall monotonicity than the 'max impact' heuristic with 25 markers.

8.3 Execution Time

We use SHARCNET to compute the results and collected the execution time of each instrumentation step. Based on the differences in the available computation time and platforms on SHARCNET, the results are purely informal and allow us to draw only conclusions when we can justify them algorithmically. For example, the weighted SAT algorithm bases on the SAT algorithm and uses a timeout to bound the execution time. Its execution time is about three orders of magnitudes greater than the SAT-based test. However, the complexity of the weighted SAT algorithm yields no improvement as seen in the performance and monotonicity analysis before.

9. Conclusion

Determining the execution path of a program helps locate bugs in a program. However, for real-time systems the developer needs to bound the instrumentation overhead.

In our paper, we proposed a framework for sampling-based monitoring to determine the execution path of the program and analyzed different algorithms for instrumenting a control-flow graph. We defined the system model and proposed two theorems based on it to determine when to stop instrumentation. While all heuristics worked to increase the sampling period, the degree-based heuristics outperformed the SAT-based ones, but the SAT-based ones achieved a higher monotonicity. Through normalized comparison, SAT-based heuristic proved to be superior to others in terms of monotonicity based on which we further concluded that only 20% of interference was from indirect interference.

We showed how to increase the sampling period by using multiple markers. However, this method had limitation in that overusing markers did not pay off as much as we expected in the long run.

The presented work fills the first pieces in a holistic framework for sampling-based execution monitoring. There is room for optimization by improving the algorithms to achieve both longer sampling periods and better monotonicity. However, we also need to investigate decision criteria when to switch markers before moving on to industrial case studies.

10. Acknowledgements

We would like to thank Byoung-gi Lee for his help with implementing the test case generator based on Task Graphs For Free. We would also like to thank the SHARCNET, Canada's shared hierarchical academic research computing network, that permitted us to use their vast computing resources for simulations. This research was supported in part by NSERC DG 357121-2008 and ORF RE03-045.

References

IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, Dec 1990.

SAT4J. web page, Oct 2009a. `www.sat4j.org`.

SHARCNET: Shared Hierarchical Academic Research Computing Network. web page, Oct. 2009b. `www.sharcnet.ca`.

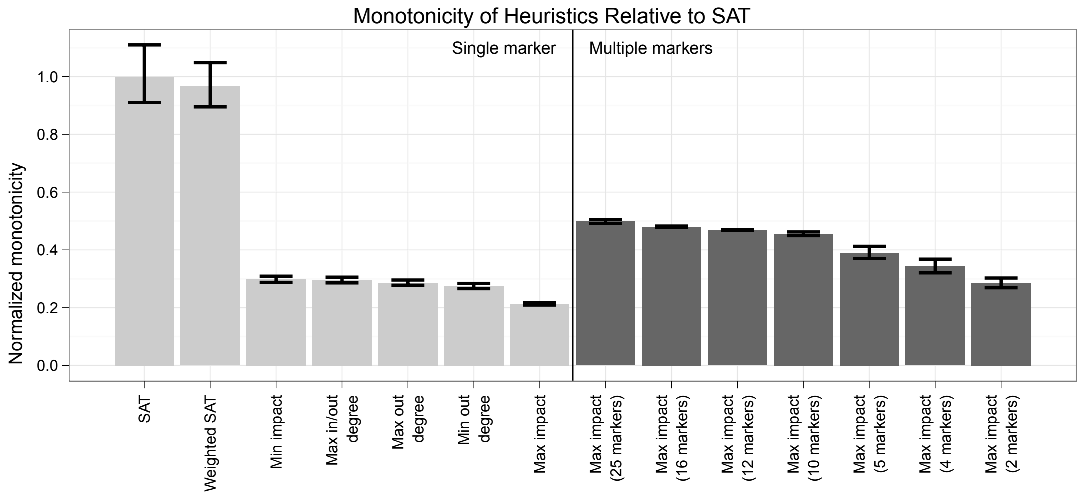

Figure 11. Monotonicity of heuristics

M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 168–179, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. doi: http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/378795.378832.

T. Ball and J. R. Larus. Optimally profiling and tracing programs. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 59–70, New York, NY, USA, 1992. ACM. ISBN 0-89791-453-8. doi: http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/143165.143180.

T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, 1994. ISSN 0164-0925. doi: http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/183432.183527.

Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):289–300, 1995. ISSN 00359246. URL `http://www.jstor.org/stable/2346101`.

M. Biberstein, V. C. Sreedhar, B. Mendelson, D. Citron, and A. Giammaria. Instrumenting annotated programs. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 164–174, New York, NY, USA, 2005. ACM. ISBN 1-59593-047-7. doi: http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/1064979.1065002.

B. Bouyssounouse and J.Sifakis, editors. *Embedded Systems Design: The ARTIST Roadmap for Research and Development*, volume 3436 of *LNCS*. Springer, first edition, May 2005.

A. Cheung and S. Madden. Performance profiling with endoscope, an acquisitional software monitoring framework. *Proc. VLDB Endow.*, 1 (1):42–53, 2008. doi: http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/1453856.1453866.

I. Chun and C. Lim. Es-debugger : the flexible embedded system debugger based on jtag technology. *Advanced Communication Technology, 2005, ICACT 2005. The 7th International Conference on*, 2:900–903, 0-0 2005. doi: 10.1109/ICACT.2005.246099.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

R. Dick, D. Rhodes, and W. Wolf. Tgff: task graphs for free. In *Hardware/Software Codesign, 1998. (CODES/CASHE '98) Proceedings of the Sixth International Workshop on*, pages 97–101, Mar 1998. doi: 10.1109/HSC.1998.666245.

S. Elbaum, H. N. Chin, M. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *Software Engineering, IEEE Transactions on*, 35(1):29–45, Jan.-Feb. 2009. ISSN 0098-5589. doi: 10.1109/TSE.2008.103.

S. Fischmeister and I. Lee. *Handbook on Real-Time Systems*, chapter Temporal Control in Real-Time Systems: Languages and Systems, pages 10–1 to 10–18. Information Science Series. CRC Press, 2007.

P. Frankl and E. Weyuker. An applicable family of data flow testing criteria. *Software Engineering, IEEE Transactions on*, 14(10):1483–1498, Oct 1988. ISSN 0098-5589. doi: 10.1109/32.6194.

M. Gallaher and B. Kropp. The Economic Impacts of Inadequate Infrastructure for Software Testing. National Institute of Standards & Technologg Planning Report 02–03, May 2002.

M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-5855-X.

Learning From Software Failure. IEEE Spectrum, September 2005.

M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. Ward. System monitoring with metric-correlation models: problems and solutions. In *ICAC '09: Proceedings of the 6th international conference on Autonomic computing*, pages 13–22, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-564-2. doi: http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/1555228.1555233.

S. H. Kirani, I. A. Zualkernan, and W.-T. Tsai. Evaluation of expert system testing methods. *Commun. ACM*, 37(11):71–81, 1994. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/188280.188373.

N. Kumar, B. R. Childers, and M. L. Soffa. Low overhead program monitoring and profiling. In *PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 28–34, New York, NY, USA, 2005. ACM. ISBN 1-59593-239-9. doi: http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/1108792.1108801.

J. J. Labrosse. *MicroC OS II: The Real Time Kernel*. CMP Books, 2002.

E. Lee and C. Zilles. Branch-on-random. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 84–93, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-978-4. doi: http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/1356058.1356070.

B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 141–154, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5. doi: http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/781131.781148.

B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI '05: Proceedings of the 2005 ACM*

SIGPLAN conference on Programming language design and implementation, pages 15–26, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/1065010.1065014.

C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: http://doi.acm.org/10.1145/1065010.1065034.

E. Metz, R. Lencevicius, and T. F. Gonzalez. Performance data collection using a hybrid approach. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 126–135, New York, NY, USA, 2005. ACM. ISBN 1-59593-014-0. doi: http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/1081706.1081729.

J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. Demand-driven structural testing with dynamic instrumentation. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 156–165, New York, NY, USA, 2005. ACM. ISBN 1-59593-963-2. doi: http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/1062455.1062496.

W. Orme. *Debug and Trace for Multicore SoCs*. ARM, September 2008. `http://www.arm.com/pdfs/CoresightWhitepaper.pdf`.

R. Santelices and M. J. Harrold. Efficiently monitoring data-flow test coverage. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 343–352, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/1321631.1321682.

A. Shye, M. Iyer, V. J. Reddi, and D. A. Connors. Code coverage testing using hardware performance monitoring support. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 159–163, New York, NY, USA, 2005. ACM. ISBN 1-59593-050-7. doi: http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/1085130.1085151.

A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. *SIGPLAN Not.*, 39(4):528–539, 2004. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/989393.989446.

D. Tennenhouse. Proactive computing. *Commun. ACM*, 43(5):43–50, 2000. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/332833.332837.

M. Thorup. All structured programs have small tree width and good register allocation. *Inf. Comput.*, 142(2):159–181, 1998. ISSN 0890-5401. doi: http://dx.doi.org/10.1006/inco.1997.2697.

B. L. Titzer and J. Palsberg. Nonintrusive precision instrumentation of microcontroller software. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 59–68, New York, NY, USA, 2005. ACM. ISBN 1-59593-018-3. doi: http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/1065910.1065919.

T. Zhang, X. Zhuang, S. Pande, and W. Lee. Anomalous path detection with hardware support. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 43–54, New York, NY, USA, 2005. ACM. ISBN 1-59593-149-X. doi: http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/1086297.1086305.

A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 1105–1112, New York, NY, USA, 2006. ACM. ISBN 1-59593-383-2. doi: http://doi.acm.org.proxy.lib.uwaterloo.ca/10.1145/1143844.1143983.