Lowering Overhead in Sampling-based Execution Monitoring and Tracing

Johnson J Thomas
Dept. of Elect. and Comp. Engineering
University of Waterloo
j22thoma@uwaterloo.ca

Sebastian Fischmeister
Dept. of Elect. and Comp. Engineering
University of Waterloo
sfischme@uwaterloo.ca

Deepak Kumar
Dept. of Elect. and Comp. Engineering
University of Waterloo

Abstract

Debugging is an important phase in the embedded software development cycle because of its high proportion in the overall cost in the product development. Debugging is difficult for real-time applications as such programs are time-sensitive and must meet deadlines in often a resource constrained environment.

A common approach for real-time systems is to monitor the execution instead of stepping through the program, because stepping will usually violate all deadline constraints. We consider a sampling-based approach for monitoring, because of its predictable overhead for the system compared to traditional monitoring. However, the sampling-based approach can easily have high overhead depending on the length of branches and the granularity of the monitoring effort. To reduce this overhead, we instrument the program with markers that will permit us to sample less frequently and thus reduce the overhead.

This leads to the interesting problems of (a) where to place the markers in the code and (b) how to manipulate the markers. While related work investigates the first part, in this work, we investigate the second component of the problem. We investigate different instrumentation schemes and propose two new schemes based on bitvectors that significantly reduce the overhead for sampling-based execution monitoring.

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

In software development, debugging is the phase where developers remove software defects from the program. Studies show that 30 to 50 percent of the development cost is spent on testing and debugging hence suggesting that debugging is costly [18]. Consequently, it is important to investigate new methods for debugging to increase productivity.

We use software instrumentation as the debugging technique to perform tracing. In this context, we instrument a control flow graph and then run the instrumented control flow graph to produce a trace. This trace is used to determine the execution path that the program had taken and hence helping in resolving conflicts that had happened during the program execution.

Recently, sampling-based debugging has been suggested [15] as a method to trace program execution especially for real-time systems. The key advantage of this approach is that it provides bounded overhead when tracing programs as the sampling period of the tracer is inversely proportional with the overhead for debugging and tracing. Traditional approaches to monitoring insert such as the one used by GNU gprof insert tracing code in the program and it is impossible to estimate the impact of the profiling code on the system. This is especially inconvenient for real-time programs where the application must meet deadlines or follow specific periodic behaviour.

Despite the advantage of predictability, the major problem is that the sampling-based approach can incur high overhead. The sampling period essentially defines the overhead as it specifies how frequently the tracer investigates the program status. Obviously the more frequently this happens, the higher the overhead. While as long as the tasks meet the deadline, such overhead may not necessarily pose a problem for real-time systems but still limits the applicability to those systems and real-time applications that have sufficient slack. The less slack an application has with respect to the deadlines, the less overhead it can tolerate. Thus, reducing the overhead increases the range of applicability of this approach.

The common approach, as proposed in related work, is to introduce markers in the program to be able to differentiate among similarly looking paths when considering the taken samples. This concept introduces two interesting problems: (a) where to place the markers in the code and (b) how to manipulate the markers. While related works has investigated the problem of where to place markers [15], this work looks into different marker schemes and functions.

For example, we propose the BITVEC$^+$ scheme which uses a combination of setting, clearing bits as well as increments. Based on the given set of paths to instrument, the scheme uses the mechanism that is more appropriate or feasible. This property led us to believe that the scheme will outperform the related approaches. As we will show in the evaluation, this intuition is correct.

The contributions of this work include:

- *Evaluating expressiveness of different schemes for markers.* We investigated four different schemes with different mechanisms for markers to see whether one of them is strictly more expressive than the others. This was also part of the learning experience to finally design BITVEC$^+$.
- *Two new expressive marker schemes.* Based on the experience looking at the different schemes, we designed two new

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
LCTES'11 April 11-14, 2011, Chicago, Illinois, USA.
Copyright ©2011 ACM 978-1-4503-0555-6/11/04…10.00.

expressive schemes for handling markers called BITVEC and BITVEC$^+$. Each of them is used in different applications.

- *Theorems for failure conditions of the new schemes.* Related work [15] showed that instrumentation schemes might result in a livelock situations where the scheme cannot make progress or cannot solve the instrumentation problem. Analogous to this concept, we present failure conditions for our two proposed schemes.
- *Improvement over related work.* In general, BITVEC$^+$ and BITVEC result in better performance than related work by a factor of two after 50 instrumentations. Also, the two schemes reduce interference [15] and thereby increase monotonicity by a factor of two over related approaches with similar memory demands and a factor of 15 when considering the basic SAT solution scheme.

The paper is structured as follows: Section 2 introduces sampling based execution monitoring. We then define the system model and certain terminology used in the paper (Section 3) which is followed by looking into the expressiveness of instrumentation schemes (Section 4) which involves an overview of various schemes and comparison of these schemes. We then propose our approach BITVEC and BITVEC$^+$ (Section 5) which consist of the theorem for its failure condition and algorithm for its working. Following, we introduce the various experimental methods and metrics (Section 6) we use in order to perform and get results. We then proceed to interpret the results (Section 7) and discussion on some further observations (Section 8). By drawing conclusions, we close the paper in Section 10 .

2. Sampling-based Execution Monitoring & Problem Motivation

In execution monitoring, the developer wants to record an execution trace of the program under test for the purpose of, for instance, debugging, profiling, testing, or runtime verification. In our setting the system consists of two parts: the executing program, and a monitor. The monitor observes the executing program and needs to log the program's execution path. In a sampling-based approach, the monitor periodically examines the state of the program and stores the state in a file. For example, the monitor will store the program counter and time stamp each time it takes a sample.

The key advantage of sampling-based execution monitoring is the bounded overhead of the monitoring system. The overhead linearly decreases with the sampling period and sample size, so a high sampling period generally leads to lower overhead than a low sampling period assuming the same sample size.

The key problem in sampling-based execution monitoring is to increase the sampling period. Notorious cases, such as programs with short conditional branches, will result in a low sampling period, if the resolution needs to be given at the granularity of basic blocks.

Listing 1 shows a simple C program with three basic blocks labeled A, B, and C. Figure 1(a) shows the resulting control flow graph. If the developer wants to monitor the execution using the sampling-based method, then the monitor will have to execute at the speed of shortest best-case execution time of $A + B$ or $A + C$; otherwise, the developer might be unable to reconstruct the execution flow assuming that it records the basic block id (vertex A, B, or C) and a time stamp. Figure 1(b) shows the timing diagram for the example. It demonstrates that, assuming all basic blocks take an execution time of 1 time unit, after two time units, it will be impossible to decide whether the program took the path $A \to B \to A$ or $A \to C \to A$. Thereby the sampling rate for the program will be $\Delta t = 2$ (more details on the formal model are in Section 3).

```
1 A:  if ( x < 5 ) {
  B:    x ++
3       goto A;
      } else {
5 C:    x-=10;
        goto A;
7     }
```

Listing 1: Illustrative example.

To increase the sampling period and thereby reduce the overhead, we introduce markers in the program. A marker is a normal variable that the monitor and the program together control to permit the developer to decide which paths the program executed even with long sampling periods.

In the example, we introduce the marker m_1 and instrument the vertex C. Figure 1(c) shows the instrumented control flow graph. Vertex C will increment the value of the marker m_1. The monitoring program will store the basic block id (vertex A, B, or C), the current value of m_1, and a time stamp. The timing diagram in Figure 1(d) shows that introducing the marker increases the sampling period to $\Delta t = 4$, because only after five time units will the program have two or more paths with the same number of increments of m_1 and the same basic block ids.

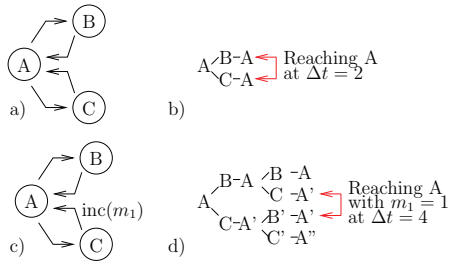


Figure 1: Example of a single instrumentation to extend Δt

3. System Model & Terminology

The following describes the system model and terminology. Our model closely follows the one presented in [15]. However, we extend the model with a generic instrumentation function to manipulate markers.

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. The resulting control-flow graph is defined as $G := \langle V, E \rangle$. Each vertex $v \in V$ represents a basic block in program. An edge $e := \langle v_s, v_d \rangle$ represents a transition from source vertex v_s to a destination vertex v_d. The transition itself takes no time. Each basic block has a best-case execution time (i.e., the shortest time that it takes to execute the program block considering all software and hardware side effects). The best case execution time of blocks can be calculated using either static analysis tools or standard measurement-based analysis tools [1]. We define this execution time via $c(v)$ and in our graphical presentation show $c(v)$ on the outgoing edges of v whenever necessary. If an edge lacks an annotation, we will assume an execution time of one (i.e., if the edge $e = \langle v_s, v_d \rangle$ has no annotation, then $c(v_s) = 1$).

A *path* is a walk $v_i \to v_{i+1} \to \ldots \to v_k$ in the graph G with a *start vertex* v_i and a *end vertex* v_k. The execution time of a path, denoted as $c_p(p)$, is the sum of all vertices along that path (i.e., $c_p(p) = \sum c(v_i)$ for all $v_i \in p$)

To bound overhead, our approach samples the executing program on a periodic basis. We define a sample as a tuple $s :=$

$\langle t, v, state \rangle$ with t being the time stamp when the sample has been taken, v being the basic block (vertex), and state being some additional program state information such as stack, variables, history. A sample is taken in periodic intervals based on the sampling period Δt.

Two paths p_1 and p_2 *intersect* with respect to a sampling period Δt, iff after taking two samples, one at time t_1 and one at time $t_2 = t_1 + \Delta t$, the two samples for both paths are identical with respect to timing and state.

In our approach, the state recorded in a sample is one or multiple marker variables. We will insert code in the program to manipulate the marker variables at run time and thus distinguish among different paths (see Section 2 for the example). We will use different schemes that change a marker's value. The marker function $\mathcal{I}$ specifies how the marker changes. In a simple case, the function might be an increment: $\mathcal{I}(m) = m + 1$; however, we will also discuss complex functions. A vertex will be able to apply the marker function several times in sequence, if the scheme permits this. The result is a simple composition of the function resulting in $(\mathcal{I} \circ \mathcal{I})(x)$.

The *instrumentation problem* for a pair of paths is as follows: Given a control flow graph G and a set of intersecting paths p_1, p_2 with respect to Δt, where to apply the marker function $\mathcal{I}$ such that the two paths no longer intersect.

We solve the instrumentation problem in an *instrumentation step.* It consists of the following sub steps: (a) finding a set of paths that intersect and need to be differentiated, (b) deciding which vertices to instrument along these paths, and (c) inserting the marker function $\mathcal{I}$ in these vertices. For example, Figures 1(a), 1(b), and 1(c) show an instrumentation step. Figure 1(a) shows the original control flow graph. In Phase (a) of the instrumentation step, we find the path pairs shown in Figure 1(b). An algorithm then decides to increment vertex C (Phase (b)), and finally applies the marker function to C. Figure 1(c) shows the resulting control-flow graph after completing the instrumentation step.

We call an instrumentation *successful,* if it solves the present instrumentation problem, meaning that it can apply the marker function in vertices to differentiate between the given intersecting paths.

Obviously, the instrumentation problem occurs iteratively for a given control flow graph. Figure 1(b) shows the instrumentation problem for $\Delta t = 2$. After successfully instrumenting the graph, a new instrumentation problem occurs for $\Delta t = 4$ as shown in Figure 1(d).

4. Expressiveness of Instrumentation Schemes

The underlying idea is to insert markers in the program and including them in the sample. If these markers are well placed in the program, then it will drastically increase the optimal sampling period [15]. A longer sampling period translates into less overhead for a monitoring system.

A good instrumentation depends on both, choice of type of changes to be made to the marker and places in the software where these changes occurs. In a particular scheme of instrumentation we make a particular type of change to marker. We now investigate different schemes to manipulate the marker and compare their expressiveness with each other.

4.1 Overview of Different Schemes

Single Increment Scheme. The first scheme is the *single increment scheme* (proposed in [15]). In this scheme a marker is a single value initially set to zero. The function $\mathcal{I}$ is defined as $\mathcal{I}(m) = m + 1$ which increments the marker's value by one.

Multiple Increment Scheme. The second scheme, called *multiple increment scheme* (proposed in [15]) extends the single increment scheme by permitting multiple applications of the marker function $\mathcal{I}$ in a vertex. The scheme also uses a single variable which is initially set to zero and the marker function $\mathcal{I}(m) = m + 1$; however, a vertex can increment the marker multiple times. The key strength of the the two different increment schemes is that it can maintain history information. We will see in the examples that we can construct cases that require such history information.

Assignment Scheme. Our next scheme for instrumentation is the *assignment scheme* (similar scheme proposed in [8]). This scheme assumes the marker to be a single variable, and it assigns values to markers rather than incrementing them. As such, the marker function is $\mathcal{I}(m) = k$ where k is an arbitrary value that might differ in each vertex.

Assigning a new value to the marker implies that the last assigned value, which represents history information, is lost. This is unlike the single and multiple increment scheme in which the history of previous applications of the marker function is preserved. As we will show later, the assignment scheme solves some simple instrumentation problems which have no solution using increment scheme.

Bit vector Scheme. The *bit vector scheme* uses markers as a bit fields instead of variables. The scheme assumes that the bit fields are initially set to zero. The marker function then sets and clears bits at a specific location in the bit field. For example a marker function can be $\mathcal{I}(x) = x[1] \uparrow \& \; x[0] \downarrow$, which will set bit two and clear bit one in the bit vector x. Every vertex can have a different version of the marker function and thus set and clear different bits.

This scheme is interesting, because it can maintain but also selectively clear history information. For example, if each vertex sets a different bit, then the scheme will maintain history information. If one vertex clears a bit that has been set in a previous vertex, then the scheme will selectively clear the history that the execution has passed through that vertex. Yet a salient part is that the scheme falls behind the increment-based schemes when it comes to building history, because the bit vector scheme, as we use it, requires the marker function to know what bits to set and clear and thus the history is finite with respect to the defined marker functions. In the increment-based schemes, the history is infinite as the marker value can always be incremented. In other words, increment-based schemes can count to infinity while the bit vector scheme can only count up to the point that the marker function has been defined. The length of the bit vector at any instant would be equal to the number of *instrumentation steps* performed, since every *instrumentation step* would use a single bit of the bit vector to distinguish the paths that intersect.We will demonstrate examples that show the difference between the increment scheme and the bit vector scheme.

4.2 Comparison in Expressiveness

Table 1 summarizes the results for the different schemes and points to the counter examples for the different schemes. The table is read in a way that the row indicates the scheme under scrutiny and the column indicates the scheme to which we compare it to. So for example the element in row one, column three points to Figure 2 which is the counter example in which the single increment scheme is superior to the assignment scheme.

The table also contains the entry *less powerful* when comparing the single increment scheme to the multiple increment scheme. This means that any case which can be instrumented by single increment can also be instrumented by multiple increment.

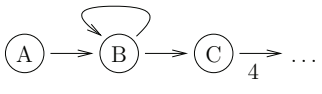


Figure 2: Counter example 1: self loop.

	Single Increment	Multiple Increment	Assignment	Bit Vector
Single Increment	-	Less powerful	Figure 2	Figure 2
Multiple Increment	Figure 4	-	Figure 5	Figure 2
Assignment	Figure 4	Figure 6	-	Less powerful
Bit vector	Figure 4	Figure 6	Figure 5	-

Table 1: Comparison table for different schemes

Figure 2 presents the counter example with the self loop. The assignment and bit vector schemes are unable to successfully instrument this control flow graph, while, the single and multiple increment-based schemes can find an instrumentation. The reason is that the example requires to build history by repetitively applying the marker function $\mathcal{I}$, as the execution repeatedly passes through B.

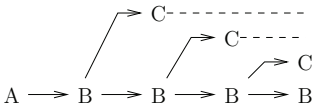


Figure 3: Timing diagram for Figure 2.

Figure 3 shows the timing diagram for the example. As stated in the model, all edges with no annotation have a delay of one. The edge leaving C has a delay of 4, thus the path $A \rightarrow B \rightarrow C$ intersects with $A \rightarrow B \rightarrow B \rightarrow C$ at $\Delta t = 4$.

Increment-based schemes can successfully instrument this graph, because no marker value is a fix-point for the marker function $\mathcal{I}(x) = x + 1$, whereas they are fix points for assignment and the bit vector scheme.

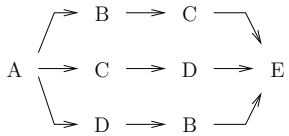


Figure 4: Counter example 2: Timing diagram that shows a permutation.

Figure 4 shows a counter example that lists the weakness of the single increment scheme over other schemes. The example has already been presented in previous work [15], and we list it here for the sake of completeness.

The graph contains three paths $p_1 = A \rightarrow B \rightarrow C \rightarrow E, p_2 = A \rightarrow C \rightarrow D \rightarrow E$, and $p_3 = A \rightarrow D \rightarrow B \rightarrow E$. To distinguish the path pair $\{p_1, p_2\}$ we have to instrument either B or D but not both, similarly for $\{p_2, p_3\}$ either C or B and for p_3, p_1 either D or C. Clearly there is no successful instrumentation with the single increment scheme because it produces the same value at the end of both the paths of the pathpair. We can instrument the above graph with multiple increment scheme by incrementing marker once at B and twice at C. Same problem can be solved with assignment or bit-vector based scheme by assigning different values to the marker or setting different bits of the marker at vertices B, C, D respectively.

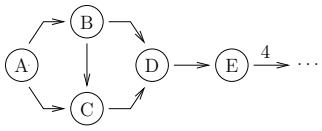


Figure 5: Counter example 3: delay.

Figure 5 shows a problem for the assignment scheme for the marker function. In the first instance of the instrumentation problem at $\Delta t = 3$, we have two intersecting paths $p_1 = A \rightarrow B \rightarrow D$ and $p_2 = A \rightarrow C \rightarrow D$. These two paths can be distinguished using the assignment scheme by assigning a marker value in either B or C.

Let's assume, that the scheme assigns the value $m_1 = x_B$ at B and we proceed. Having resolved this path pair now we get a set of intersecting paths $\{p_3, p_4, p_5\}$ with $p_3 = A \rightarrow B \rightarrow D \rightarrow E, p_4 = A \rightarrow C \rightarrow D \rightarrow E$, and $p_5 = A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$. Paths p_3 and p_5 have the same value for $m_1 = x_B$ and need to be differentiated.

At this point, the assignment-based scheme faces a problem, because it can neither instrument D or C. Vertex D is shared in both paths and any instrumentation of D in the future will overwrite the value of m_1. Therefore, it will decrease the sampling period again to $\Delta t = 3$ as the current paths p_1 and p_2 will become indistinguishable again. The assignment-based scheme also cannot instrument C, because then paths p_4 and p_5 will be indistinguishable. Thus, the assignment-based scheme fails at this example.

Other schemes can easily instrument this by incrementing the marker in B and C or by setting different bits at B and C.

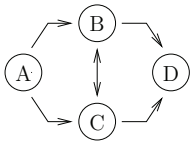


Figure 6: Counter example 4: diamond.

Figure 6 demonstrate problem associated with the increment-based schemes. Basic idea is that, although we may be incrementing a marker at different vertices on different paths, at the end, the aggregated value of the marker is similar. This property makes the increment-based scheme ineffective when one path in a path pair is just the permutation of vertices in the other path of path pair, in such case any instrumentation (single or multiple increment, whereas Figure 4 only applies to single increment) will lead to same value of the marker at the end. Figure 6 generate this kind of path pairs which are shown in Figure 7.

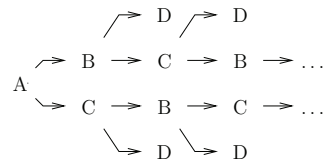


Figure 7: Timing diagram for Figure 6.

There are two types of intersecting path pairs: one with odd path length and other with even path length. If we consider a path pair having paths of odd length, then we can instrument the paths. If we consider a path pair having paths of even length, then one path will be the permutation of vertices of another path. Paths

$A \rightarrow B \rightarrow C \rightarrow D$ and $A \rightarrow C \rightarrow B \rightarrow D$ form a pair of intersecting paths of that kind. Obviously no instrumentation using single or multiple increments can solve this instrumentation problem, but simply assigning two different values to the marker at vertex B and C successfully differentiates the paths.

The above examples clearly suggest that none of the methods when applied individually are expressive enough to make the paths of a path pair distinguishable in all cases.

5. Our Approach

Based on the insights gained from observing the expressiveness of different schemes as discussed in Section 4 and especially Table 1, we now investigate two schemes based on bit vectors. BITVEC is the direct implementation of a bit vector scheme and BITVEC$^+$ is a hybrid of increment and a bit vector scheme.

5.1 BITVEC

The BITVEC algorithm follows a greedy approach in finding the vertices that can be instrumented in order to distinguish them at the time of sampling. It tries to find the distinct vertices on the paths or set of vertices that can manipulate a bit either by setting or clearing, so that the paths are distinguishable at the time of sampling.

Terminology and Definitions. G is the control flow graph. p_1 and p_2 are the two intersecting paths. G' is the instrumented control flow graph that has successfully solved the instrumentation problem. We denote $V(p)$ to represent the set of vertices that lie on the path p. The function $first_v(p)$ returns the first vertex on the path p. The function $locc_v(v, p)$ returns the index of the last occurrence of the vertex v in path p. For example, $locc_v(A, [A \rightarrow B \rightarrow B \rightarrow A])$ = 4 and $locc_v(A, [A \rightarrow B \rightarrow A])$ = 3.

Function 1 shows the algorithm for BITVEC . We first calculate δ which is the set difference between the union set of $V(p_1)$ and $V(p_2)$ and the intersection set of $V(p_1)$ and $V(p_2)$. If the set δ contains elements, then we pick one of the vertices from δ for instrumentation; otherwise, we need to investigate whether BITVEC can successfully instrument the path pair.

If δ is empty, then we reverse the paths (remember that the last assignment to the marker overwrites previous ones), and store all vertices that distinctively occur last in one path into a temporary set V_{able}. All other vertices, e.g., vertices that occur on both paths at the same position, are removed. Once we went through the whole path, if V_{able} is empty, then BITVEC is unable to instrument the path pair (following Theorem 1). If V_{able} contains vertices, then we pick the first two of them since these were the first two vertices that differed.

The complexity of the algorithm is linear with respect to the number of vertices in V. Calculating δ is linear because represent each of the paths p_1 and p_2 as an array of bits of size $|V|$ and perform bit operations to obtain union, intersection and difference operations. In the function itself, the while loop traverses through the paths and thus has linear complexity as well.

Theorem 1 (BITVEC Failure Condition)**.** *For two intersecting paths p_1 and p_2,* BITVEC *will be unsuccessful in distinguishing the paths, if and only if the following conditions hold:*

1. $(V(p_1) \cup V(p_2)) - (V(p_1) \cap V(p_2)) = \emptyset$
2. $\nexists v_1, v_2$ *such that* $locc_v(v_1, p_1) = locc_v(v_2, p_2)$

Proof. Proof is in the form of two parts: if and only if.

if: We use a proof by contradiction method for one and two vertices. Assuming that the graph can be instrumented with the conditions mentioned above being true.

We assume to instrument one vertex. Assume that you find one instrumentable vertex that after instrumentation will distinguish the two paths. This vertex must be unique for the two paths. This clearly cannot hold, because condition one in the theorem states that such a vertex does not exist.

We assume to instrument two vertices. Assume that you find two instrumentable vertices that after instrumentation will distinguish the two paths. First, the two vertices must be shared between the paths (c.f., condition 1 in the Theorem). If you still find two such vertices, then the two vertices must set and clear a bit at distinct positions in the paths with no other vertex overwriting the bit in the remaining parts of the paths. There exist no such two vertices, because of condition 2 in the theorem.

We assume to instrument n vertices. Trying to instrument n vertices is similar to instrumenting one or two vertices. If the remainder of $\frac{n}{2}$ is 1, then if we can instrument the paths, we will be able to also instrument them with one vertex only. If the remainder of $\frac{n}{2}$ is 0, then if we can instrument the paths, with two vertices only. The argument for both cases is that if the nth vertex makes the difference, the algorithm would have found it as the first vertex to try; and if the $n-1$th and nth vertices make the different, the algorithm would have found them when trying to instrument with two vertices.

only if: We use a proof by contradiction for the 'only if' part. We assume that either $(V(p_1) \cup V(p_2)) - (V(p_1) \cap V(p_2)) \neq \emptyset$ or $\exists v_1, v_2$ such that $locc_v(v_1, p_1) = locc_v(v_2, p_2)$ or both.

- Case 1 $(V(p_1) \cup V(p_2)) - (V(p_1) \cap V(p_2)) \neq \emptyset$: $(V(p_1) \cup V(p_2))$ is the union of the set of vertices on paths p_1 and p_2. If $(V(p_1) \cup V(p_2)) - (V(p_1) \cap V(p_2)) \neq \emptyset$ then we have vertices present on either only p_1 or p_2. We can instrument any of these vertices to distinguish the two paths.
- Case 2 $\exists v_1, v_2$ such that $locc_v(v_1, p_1) = locc_v(v_2, p_2)$: Since two such vertices exist, we can instrument one of these vertices to set a bit and instrument the other vertex to clear the same bit, hence making it distinguishable at the time of sampling.

Function 1 Instrument graph with BITVEC

Input: Control flow graph G, paths p_1, p_2
Output: Instrumented control flow graph G'

```
δ ⇐ (V(p1) ∪ V(p2)) − (V(p1) ∩ V(p2))
if δ = ∅ then
    reverse paths p1 and p2
    while |p1| ≠ 0 ∨ |p2| ≠ 0 do
        if first_v(p1) ≠ first_v(p2) then
            add first_v(p1) and first_v(p2) to V_able
        end if
        remove first_v(p1) and first_v(p2) from p1 and p2
    end while
    if V_able = ∅ then
        BITVEC terminates
    else
        add first two vertices added to V_able to V_instr
    end if
else
    add one vertex of δ to V_instr
end if

if |V_instr| = 2 then
    set a bit in one vertex, clear the same bit in the other vertex
end if
if |V_instr| = 1 then
    set a bit for the vertex
end if
```

- Case 3 $(V(p_1) \cup V(p_2)) - (V(p_1) \cap V(p_2)) \neq \emptyset$ and $\exists v_1, v_2$ such that $locc_v(v_1, p_1) = locc_v(v_2, p_2)$: From Case 1 and Case 2, it follows that in Case 3, the paths can be instrumented.

From cases 1 to 3, it follows that if any of the two conditions in the theorem is true, then the paths can be instrumented and hence contradict the initial fact that the graph cannot be instrumented. □

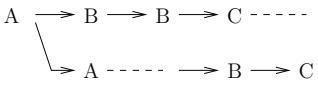


Figure 8: Two paths that cannot be instrumented using BITVEC .

Example 1. *Figure 8 shows two paths that cannot be instrumented using* BITVEC *. Note that vertex A and C has a delay of two, while B has a delay of 1. Given the two paths, we can compute the following elements:*
$V(p_1) = \{A, B, C\}$
$V(p_2) = \{A, B, C\}$
$V(p_1) \cup V(p_2) = \{A, B, C\}$
$V(p_1) \cap V(p_2) = \{A, B, C\}$
$(V(p_1) \cup V(p_2)) - (V(p_1) \cap V(p_2)) = \emptyset$ *(Condition 1 in Theorem 1)*

As seen in Figure 8, condition 2 of Theorem 1 also holds as $\nexists v_1, v_2$ *such that* $locc_v(v_1, p_1) = locc_v(v_2, p_2)$*. Since both conditions hold, the example cannot be instrumented with* BITVEC .

Note that for example the increment-based scheme is able to instrument this by simply incrementing a marker at B.

5.2 BITVEC$^+$

BITVEC$^+$ also follows a greedy approach similar to BITVEC in finding vertices that can be instrumented so that the paths are distinguishable at the time of sampling. The only difference of BITVEC$^+$ and BITVEC is that BITVEC$^+$ uses increment based scheme for those cases where BITVEC fails. The advantage of BITVEC$^+$ is a further increase in sampling period compared to BITVEC but introduces interference due to the use of increment based schemes.

Terminology and Definitions. The inputs, outputs, $first_v(p)$ and $locc_v(v, p)$ are similar in structure to Function 1. The function $\delta_{freq}(p_1, p_2)$ of two paths calculates the set of vertices that occur differently often on the two paths considering also the state information in the vertex (in our case the marker values).

Function 2 shows the algorithm for BITVEC$^+$. We first calculate the $\delta_{freq}(p_1, p_2)$ and then remove all vertices shared between the two paths under consideration and assign this to δ'. If the δ' set still contains elements, then we can pick one of the vertices for instrumentation; otherwise, we need to investigate whether BITVEC$^+$ can successfully instrument the path pair.

If δ' is empty, then we reverse the paths (remember that the last assignment to the marker overwrites previous ones), and store all vertices that distinctively occur in one path last in V_{able}. All other vertices, e.g., vertices that occur on both paths at the same position, are removed. Once we went through the whole path, if V_{able} is empty, then we check to see whether δ is an empty set or not. If δ is an empty set then BITVEC$^+$ is unable to instrument the path pair (following Theorem 2) otherwise we pick up one vertex in δ, initialize a marker i at the starting vertex of G and instrument this vertex with i++ . If V_{able} contains vertices, then we pick the first two of them.

The complexity of the algorithm is linear with respect to the number of vertices in V. Calculating δ_{freq} is linear, because executes a single pass over both paths and counts how often vertices occur. It then subtracts the shared vertices between the paths and returns the delta set. In the function itself, the while loop traverses through the paths and thus has linear complexity as well.

Function 2 Instrument graph with BITVEC$^+$

Input: Control flow graph G, paths p_1, p_2
Output: Instrumented control flow graph G'

```
δ ⇐ δ_freq(p1, p2)
δ' ⇐ δ − {V(p1) ∩ V(p2)}
if δ' = ∅ then
    reverse paths p1 and p2
    while |p1| ≠ 0 ∨ |p2| ≠ 0 do
        if first_v(p1) ≠ first_v(p2) then
            add first_v(p1) and first_v(p2) to V_able
        end if
        remove first_v(p1) and first_v(p2) from p1 and p2
    end while
    if V_able = ∅ then
        if δ = ∅ then
            BITVEC+ terminates
        else
            add one vertex of δ to V_instr
            increment_flag ⇐ true
        end if
    else
        add first two vertices added to V_able to V_instr
    end if
else
    add one vertex of δ' to V_instr
end if

if increment_flag is true then
    initialize marker i at the starting vertex of G
    perform i++ at v ∈ V_instr
else
    if |V_instr| = 2 then
        set a bit in one vertex, clear the same bit in the other vertex
    end if
    if |V_instr| = 1 then
        set a bit for the vertex
    end if
end if
```

Theorem 2 (BITVEC$^+$ Failure Condition)**.** *For two intersecting paths* p_1 *and* p_2*,* BITVEC$^+$ *will be unsuccessful in distinguishing the paths, if and only if the following conditions hold:*

1. $\delta_{freq}(p_1, p_2) = \emptyset$
2. $\nexists v_1, v_2$ *such that* $locc_v(v_1, p_1) = locc_v(v_2, p_2)$

Proof. The condition $\delta_{freq}(p_1, p_2) = \emptyset$ signifies that the order of vertices on path p_1 is a permutation of the order of vertices on path p_2. Hence, $(V(p_1) \cup V(p_2)) - (V(p_1) \cap V(p_2)) = \emptyset$. The proof trivially follows by combining Theorem 1 and the theorem on single path pair termination in [15]. □

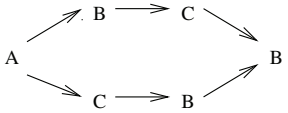


Figure 9: Two paths that cannot be instrumented using BITVEC$^+$.

Example 2. *Figure 9 shows two paths that cannot be instrumented using* BITVEC$^+$ *. Given the two paths, we can compute the following elements:* $\delta_{freq}(p_1, p_2) = \emptyset$ *(Condition 1 in Theorem 2).*

As seen in Figure 9, condition 2 of Theorem 2 also holds as $\nexists v_1, v_2$ *such that* $locc_v(v_1, p_1) = locc_v(v_2, p_2)$*. Since both conditions hold, the example cannot be instrumented with* BITVEC$^+$.

6. Experimental Method

To validate the theorems and the concepts of this work, we extended an existing instrumentation engine [15]. The instrumentation engine provides a framework to test different instrumentation schemes for the instrumentation problem defined above. We have added the BITVEC and BITVEC$^+$ schemes to the instrumentation engine. The implementation of these schemes is as easy or as difficult as implementing a counter or the single increment scheme.The outputs of this engine are the instrumented vertices, the required execution time, the resulting sampling period, and the amount of extra memory used in the instrumented program.

For our input set, we generated about 5000 control flow graphs using a modified version of Task Graphs For Free [14]. Each control flow graph has on an average 114 basic blocks and 218 edges. The control flow graphs follow C program flows [29]. Control flow graphs of real programs from test benchmarks are future currently in progress. One experiment run works as follows: we first select a control flow graph and a scheme (either increment based with 10 or 25 markers, BITVEC , or BITVEC$^+$) and then pass them to the instrumentation engine. The engine computes the instrumented control flow graph and returns the sampling period, vertices to instrument, the required execution time, and the amount of extra memory. We performed the computation on a standard dual-core workstation with 2GB of memory and the simulations took reasonable execution time (tens of seconds per step). Since the instrumentation process is performed offline, the actual execution time is negligible as long as it is tolerable for the developer.

The data successfully passed the integrity checks of the engine which were: (1) in BITVEC the increase in sampling period is strong monotonically increasing and (2) on average, the sampling period increases with the increase in the number of instrumentation steps.

The various metrics used for this experimentation are similar to those chosen in related work [15]; except that we chose the name monotonicity instead of usability. The metrics are described in the sections below.

6.1 Instrumentation Performance Metric

To compare the performance of BITVEC and BITVEC$^+$ over the increment schemes with 10 or 25 markers, we take the maximum sampling period achieved in each run per algorithm and sum them up: $P = \sum max(T_i)$. This metric is robust against direct and indirect interference defined in [15].

6.2 Monotonicity Metric

Monotonicity describes how often the sampling period decreases after an instrumentation step. This is important to know, because it means that although the instrumentation takes place (and overhead increases), the sampling period actually decreased. Related work calls this property usability. We use the following monotonicity metric to evaluate various heuristics: $M = \frac{N}{\sum d_i}$ with

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

The term d_i denotes the decrement between two instrumentation steps run_i and run_{i+1}, if the sampling period of run_i is greater than the subsequent 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 instrumentation steps. The decrement represents the interference introduced by instrumenting the vertices. Since monotonicity is the reciprocal of the sum of decrements, the lesser the sum of decrements, the greater the monotonicity of the strategy used.

6.3 Memory Use in the Instrumented Program

In contrast to related work, we also evaluate memory use in the instrumented program. As the schemes instrument the program, they progressively need more memory as they use new markers. While using more memory not necessarily invalidates approaches, we use this metric to evaluate the performance especially for schemes with low memory demands (e.g., single increment scheme). We compute the metric as follows: $mu = \frac{\Delta t}{mem}$.

7. Results

We used the experimental methods as discussed in Section 6 to compare the results of the BITVEC and BITVEC$^+$ with the increment based schemes discussed in [15]. The results are categorized and discussed below.

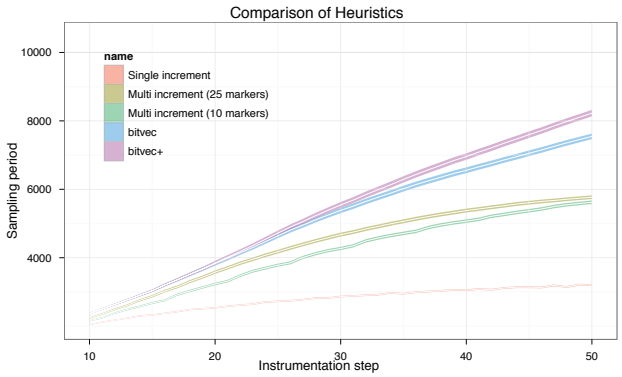


Figure 10: Comparing achieved sampling period of different schemes

The instrumentation performance of BITVEC and BITVEC$^+$ are much better than the single and multi-increment schemes even if they use multiple markers. Figure 10 shows the comparison of the sampling period of different schemes with the increase in the number of instrumentation steps. The x-axis shows the number of instrumentation steps (i.e., 50 times solving an instrumentation problem to increase the sampling period). The y-axis shows the achieved sampling period. The higher the sampling period, the better, because it means the lower the overhead. The graph clearly shows the improvement in the sampling period using BITVEC$^+$ and BITVEC schemes over the other schemes as they level off much earlier. This is due to the fact that in BITVEC scheme only the last change made to a bit on a path will be visible at the time of sampling and we try to find two different vertices that can be instrumented (one vertex to be assigned to set a bit and other vertex is assigned to clear the same bit) which occur at the same time stamp. Another reason for the increase in sampling period for BITVEC is the use of multiple bits to remember the past traces. BITVEC$^+$ shows further improvement compared to BITVEC due to the fact that BITVEC$^+$ can instrument more cases than BITVEC . The consequence of this result is that we can lower the overhead of sampling-based monitoring with the new schemes BITVEC and BITVEC$^+$.

Figure 11 shows the progressive gain of sampling period over adding memory. The x-axis shows the number of instrumentation

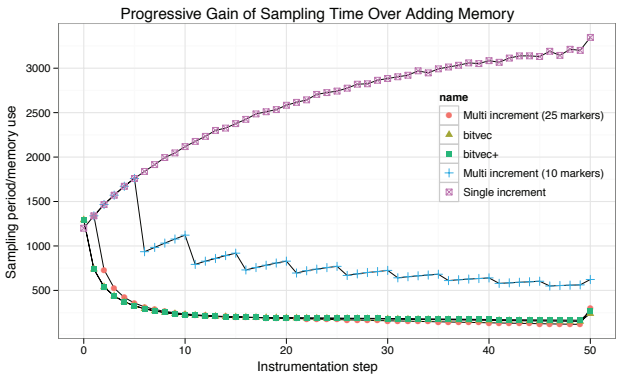


Figure 11: Progressive gain of sampling period over adding memory

steps. The y-axis shows the ratio of sampling period to memory used. The higher the value, the more sampling period we receive for using memory in the instrumented application. In other words, the higher the value, the more efficiently we are using the available memory.

As the graph indicates, the single marker scheme performs best. The main reason is that in the single marker scheme, the memory used for each instrumentation step is constant (one marker) and the sampling period increases as the instrumentation steps increases; hence an increase in the ratio of sampling period to memory used. BITVEC and BITVEC$^+$ attain lower values, as the number of instrumentation steps increases. This is due to the fact that BITVEC and BITVEC$^+$ uses one bit of the bit vector in each instrumentation step. Interestingly, the multi marker scheme also uses about the same amount of memory as BITVEC and BITVEC$^+$, however, it fails to achieve similar performance (c.f., Figure 10). This experimentation result shows that the BITVEC and BITVEC$^+$ scheme should be used in those systems where sufficient memory can be allotted for the bit vector.

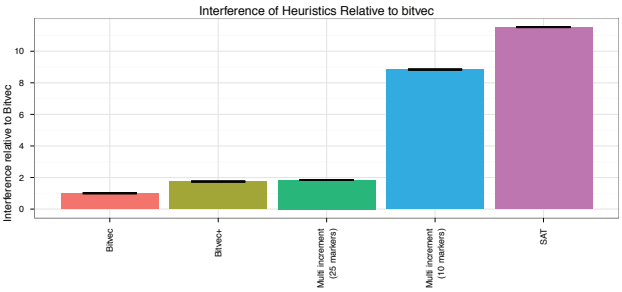


Figure 13: Interference of different heuristics relative to Bitvec

	Name	Normalized	SEM	Absolute
1	Bitvec	1	0.0001	2
2	Bitvec+	2	0.0001	3
3	Multi increment (25 markers)	2	0.0001	3
4	Multi increment (10 markers)	9	0.0003	14
5	SAT	12	0.0004	19
6	Single increment	109	0.0017	177

Table 2: Interference for different techniques normalized to bitvec

The interference level of BITVEC is very low or none because BITVEC eliminates direct as well as indirect interference to a great extent as conjectured in Section 5. Figure 13 shows the comparison of the interference levels of different schemes relative to BITVEC The x-axis shows the different schemes. The y-axis shows the interference relative to BITVEC . The black bars in Figure 13 show the standard error of the mean. Table 2 shows the values in table form and also includes the single marker case. Table 2 also shows the standard error of the mean (SEM) for the different schemes. SEM is the degree of deviation from the mean. The higher the value of SEM, the more deviation and hence lower consistency in the results. The figure and the table show that BITVEC has very good monotonicity as it eliminates all interference. This makes it useful for tooling, because users will never experience a drop in the sampling period as they use the tool.

The monotonicity of BITVEC is high because BITVEC eliminates both direct as well as indirect interference. The lesser the interference, the more is the monotonicity. Figure 12 shows the monotonicity of different schemes. The x-axis shows the number of instrumentation steps. The y-axis shows the sampling period.

Figure 12(a) shows the monotonic increase in test case 103. Figure 12(a) shows that the BITVEC and BITVEC$^+$ both follow the technique until the point where BITVEC must terminate, because it cannot further instrument. BITVEC$^+$ can continue, because it uses the single increment method for that particular problem. Since BITVEC$^+$ sometimes uses increment, one can see a drop in monotonicity in the scheme. Test case 103 also shows that the bit vector schemes sometimes are slower at the beginning but might then catch up later. This means that the single increment can sometimes be the better scheme.

Figure 12(b) shows test case 104. The case is interesting, because it shows the erratic behaviour of single increment schemes (see steps 40 to 50). The test case also shows a scenario where BITVEC$^+$ experiences interference, because it uses markers (see steps 47 and 48).

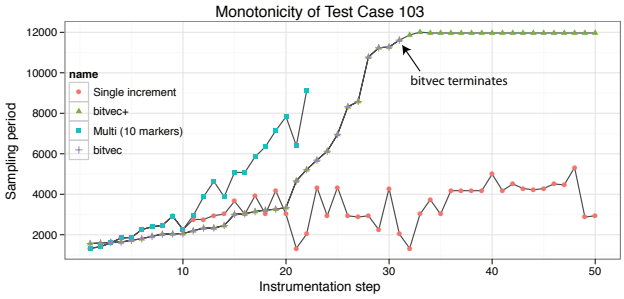
Figure 12(c) shows a notorious test case where BITVEC and BITVEC$^+$ never catches up with the increment technique within the first 50 instrumentation steps. We ran the test case for 100 steps and it eventually creates steep increases on the sampling period and then surpasses the multi increment scheme. The control flow graph has 296 basic blocks and 619 edges. When looking at the instrumentation trace, it seems that BITVEC$^+$ is stuck by solving many individual problems while the single increment seems to be able to solve them more quickly.

Figure 12(d) shows a test case where BITVEC$^+$ is far superior to the other techniques. BITVEC follows the same path as BITVEC$^+$ until some point after which it can no longer instrument the paths. The single marker and 10 markers techniques are the almost the same with single marker having more interference.

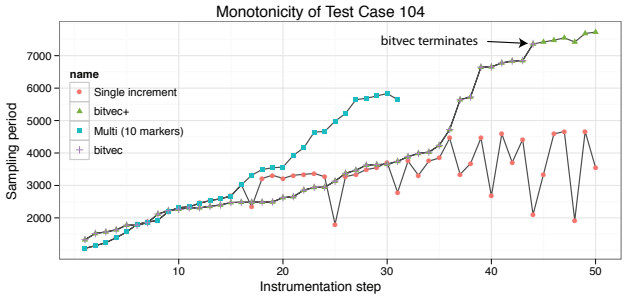
	Name	Absolute	Ratio	Overall
1	bitvec	11045	0.834	0.189
2	Increment	2201	0.166	0.038

Table 3: Comparing when one technique is superior than the other in the empirical data

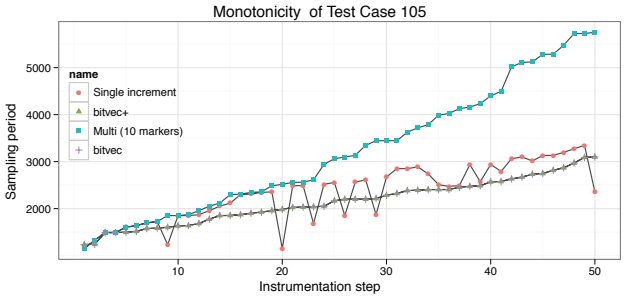
Table 3 shows the comparison of the number of test cases where BITVEC was better than the increment scheme and vice versa. As shown in the table, BITVEC is superior to increment in 83.4% of the test cases that were used during experimentation while increment was superior in the remaining 16.6%. The reason behind the superiority of BITVEC over increment is that BITVEC can empirically instrument more cases than the single increment scheme (See Theorem 1 vs the path pair termination in [15]). This result shows that BITVEC is reliable in producing higher sampling periods and hence reducing the overhead for sampling based monitoring.



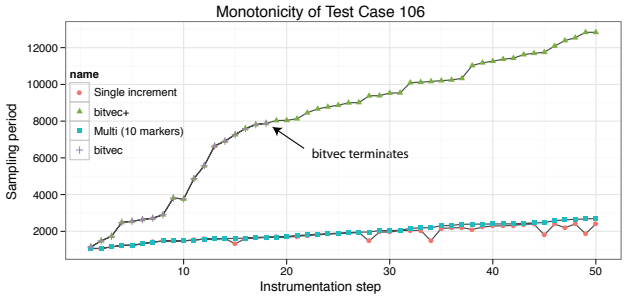
(a) BITVEC$^+$ catching up late.



(b) Single marker showing erratic behaviour.



(c) BITVEC$^+$ never catching up until step 50.



(d) BITVEC$^+$ far superior than the others.

Figure 12: Examples of behaviour with respect to monotonic increase in the sampling period.

8. Discussion

Some interesting observations can be made from the results obtained. We study the monotonic behaviour of BITVEC and BITVEC$^+$ and observe that although the graph is monotonic in nature, there is a case where BITVEC and BITVEC$^+$ gradually grows. Figure 12(c) shows a case where BITVEC and BITVEC$^+$ never catches up compared to instrumentation by a single marker. We suspect the reason for this being that although both increment scheme and BITVEC are greedy algorithms, increment based schemes make better decisions which solve number of hidden paths as mentioned in related work [15] while BITVEC solves one problem at a time. This observation leads to interesting future work to try and decide based on the graph structure which instrumentation scheme to use.

The applicability of the BITVEC and BITVEC$^+$ schemes depend on the trade off between interference and sampling periods. If the user rejects the notion that applying more instrumentation may decrease the sampling period, then BITVEC is the primary choice for the algorithm. Whereas if the user accepts this behaviour, then BITVEC$^+$ is more suitable, because it will achieve better overall results. As clearly indicated in Figure 12(b), the graphs of BITVEC and BITVEC$^+$ are the same until a certain instrumentation step after which BITVEC can no longer instrument while BITVEC$^+$ uses increment based scheme to further instrument the graph. However, Figure 10 also shows that although BITVEC$^+$ can achieve higher sampling periods compared to BITVEC , but it can introduce interference as indicated by a drop in Figure 12(b) between steps 40 and 50. Also another observation is that adding a marker incurs only negligible overhead in code size and execution time since these are single statements, however, investigating the influence of these markers on different metrics is a potential area for future work.

9. Related Work

A large body of work has been established in the area of tracing with general work on trace-based debugging [11, 21, 26]. In general, the problem of placing probes and counters efficiently in a program is hard [17] and many different variations of the tracing problem exist [7]. Different approaches to instrument and reduce the overhead exist, for example, coalescing probes [20], labelling paths similar to our assignment scheme [8], flexibility in setting probes and thus reducing overhead [22, 5, 28], dynamic binary rewriting to change the instrumentation at run time [24, 9, 6], running the instrumentation only once per iteration [25], simulator-based approaches [30], time-aware instrumentation [16], and code duplication [3].

Tracing can also be implemented in hardware. Solutions such as JTAG [12], Nexus [23], and ARM CoreSight [27] with, for instance, the ETM permit inspecting and tracing the system at a hardware level.

Often sampling has been used for profiling without the need of full accuracy for recreating execution flows as necessary for debugging and monitoring [19, 31, 32, 2, 13, **?**, 10, 4]. Our work uses sampling in a different context. Our work does not primarily aim at profiling, but at begin able to reconstruct whole execution paths using sampling.

10. Conclusion

Monitoring execution and tracing using sampling is an important technique for real-time embedded applications that need to meet the timing deadlines. The sampling-based approach permits computing the overhead and thus permits engineering the system.

In the sampling-based approach, the major drawback is the overhead that originates from required high sampling rates. In this work, we have investigated several schemes for using markers to reduce the required sampling rate and thus reduce the overhead.

Specifically, we have proposed the BITVEC and BITVEC$^+$ schemes, established their failure conditions, and showed that they provide superior performance than related work with respect to increasing the sampling period, interference, and monotonicity.

Future work is to look into the specific problems shown in test case 105 and defining a heuristic when to apply which instrumentation scheme. This could then also lead to a general framework that uses different instrumentation schemes based on the best applicability given the current instrumentation problem to be solved.

Acknowledgements

This research was supported in part by NSERC DG 357121-2008, ORF RE03-045, ORE RE-04-036, ISOP IS09-06-037, the MITACS Globalink Program and CMC Microsystems through CFI 20314.

References

[1] RapiTime. web page. `http://www.rapitasystems.com/rapitime`.

[2] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.-T.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, 1997.

[3] M. Arnold and B.G. Ryder. A framework for reducing the cost of instrumented code. In *Proc. of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, pages 168–179, 2001.

[4] M. Arnold and P.F. Sweeney. Approximating the calling context tree via sampling. Technical Report RC 21789, IBM T.J. Watson Research Center, July 200.

[5] A. Ayers, R. Schooler, C. Metcalf, A. Agarwal, J. Rhee, and E. Witchel. Traceback: First fault diagnosis by reconstruction of distributed control flow. *ACM SIGPLAN Not.*, 40(6):201–212, 2005.

[6] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proc. of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, 2000.

[7] T. Ball and J.R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, 1994.

[8] T. Ball and J.R. Larus. Efficient path profiling. In *Proc. of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 46–57, 1996.

[9] D. Bruening, T. Garnett, and S. Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *Proc. of the International Symposium on Code Generation and Optimization (CGO)*, pages 265–275, 2003.

[10] M. Burrows, U. Erlingsson, S.-T. A. Leung, M. T. Vandevoorde, C. A. Waldspurger, K. Walker, and W. E. Weihl. Efficient and flexible value sampling. *ACM SIGPLAN Not.*, 35(11):160–167, 2000.

[11] J.-D. Choi, B.P. Miller, and R.H.B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. Program. Lang. Syst.*, 13(4):491–530, 1991.

[12] I. Chun and C. Lim. Es-debugger: the flexible embedded system debugger based on jtag technology. *Proc. of the 7th International Conference on Advanced Communication Technology (ICACT)*, 2:900–903, 0-0 2005.

[13] J. Dean, J.E. Hicks, C.A. Waldspurger, W.E. Weihl, and G. Chrysos. Profileme: Hardware support for instruction-level profiling on out-of-order processors. In *Proc. of the 30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 1997.

[14] R.P. Dick, D.L. Rhodes, and W. Wolf. Tgff: Task graphs for free. In *Proc. of the Sixth International Workshop on Hardware/Software Codesign (CODES/CASHE)*, pages 97–101, Mar 1998.

[15] S. Fischmeister and Y. Ba. Sampling-based Program Execution Monitoring. In *Proc. of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 133–142, 2010.

[16] S. Fischmeister and P. Lam. Time-aware Instrumentation of Embedded Software. *IEEE Transactions on Industrial Informatics*, 2010.

[17] I.R. Forman. On the time overhead of counters and traversal markers. In *Proc. of the 5th International Conference on Software Engineering (ICSE)*, pages 164–169, 1981.

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

[19] S.L. Graham, P.B. Kessler, and M.K. Mckusick. Gprof: A call graph execution profiler. *ACM SIGPLAN Not.*, 17(6):120–126, 1982.

[20] N. Kumar, B.R. Childers, and M.L. Soffa. Low overhead program monitoring and profiling. In *Proc. of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 28–34, 2005.

[21] J. R. Larus. Abstract execution: a technique for efficiently tracing programs. *Softw. Pract. Exper.*, 20(12):1241–1258, 1990.

[22] J.R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *Proc. of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, 1995.

[23] Ashling Microsystems Ltd. *IEEE-ISTO 5001TM-1999, The Nexus 5001 Forum Standard*. Nexus 5001 Forum, 2000.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005.

[25] 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: Proc. of the 27th International Conference on Software Engineering*, pages 156–165, 2005.

[26] R.H.B. Netzer and M.H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *PLDI '94: Proc. of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 313–325, New York, NY, USA, 1994. ACM.

[27] W. Orme. *Debug and Trace for Multicore SoCs*. ARM, September 2008.

[28] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. *ACM SIGPLAN Not.*, 39:528–539, 2004.

[29] M. Thorup. All structured programs have small tree width and good register allocation. *Inf. Comput.*, 142(2):159–181, 1998.

[30] B.L. Titzer and J. Palsberg. Nonintrusive precision instrumentation of microcontroller software. In *LCTES '05: Proc. of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 59–68, 2005.

[31] J. Whaley. A portable sampling-based profiler for java virtual machines. In *Proc. of the ACM 2000 Conference on Java Grande*, pages 78–87, 2000.

[32] Y. Zhong and W. Chang. Sampling-based program locality approximation. In *Proc. of the 7th International Symposium on Memory Management (ISMM)*, pages 91–100, 2008.