Time-triggered Program Monitoring

By

Johnson J. Thomas

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2012

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Johnson J. Thomas

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. Runtime monitoring aims at analyzing the well-being of a system at run time in order to detect errors and steer the system towards a healthy behavior. Such monitoring is a complementary technique to other approaches for ensuring correctness, such as formal verification and testing. We consider a time-triggered approach for program monitoring at runtime, resulting in bounded and predictable overhead. Gaining such characteristics for overhead is highly desirable for designing and engineering time-critical applications, such as safety-critical embedded systems.

The two techniques, which we investigate in this work are, *time-triggered execution monitoring* and *time-triggered self-monitoring*. In time-triggered execution monitoring, a monitor runs as a separate process in parallel with an application program under scrutiny and samples the program's state periodically to evaluate a set of properties. Applying this technique in computing systems, results in bounded and predictable overhead. However, the time-triggered 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 [27] 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 time-triggered execution monitoring.

Although time-triggered execution monitoring results in bounded and predictable overhead, it suffers from several drawbacks. Some of the drawbacks are; the time-triggered monitor requires certain synchronization features at the operating system level and may

suffer from various concurrency and synchronization dependencies and overheads as well as possible unreliability of synchronization primitives in a real-time setting. Furthermore, the time-triggered execution monitoring scheme requires the embedded environment to provide multi-tasking features, which might not always be readily available (eg. TinyOS). To address the problems associated with time-triggered external monitoring, we propose a new method, where the program under inspection is instrumented, so that it *self-samples* its state in a periodic fashion without requiring assistance from an external monitor or an internal timer. We call this technique time-triggered self-monitoring. We formulate an optimization problem for minimizing the number of points in a program, where self-sampling instrumentation instructions must be inserted. We show that this problem is NP-complete. Consequently, we propose a SAT-based solution and a heuristic to cope with the exponential complexity. The experimental results show that a time-triggered self-monitored program performs significantly better in terms of execution time, binary code size, and context switches when compared to the same program monitored by an external time-triggered monitor.

Acknowledgements

I am grateful and thankful to my God and Saviour Jesus Christ for helping me reach until this stage of my life. I would like to thank all the people who made this work possible. Firstly, I would like to thank Dr Sebastian Fischmeister who mentored and guided me through my masters. I thank him for his support, advice, technical guidance and help which facilitated me in performing the work presented in this thesis. I would also like to thank Dr Borzoo Bonakdarpour for his guidance and support for the two papers that we had worked together.

I would like thank the members of Real-Time Embedded Software Group (RESG) for their support and motivation during the tough times of my masters. I would also like to thank my friends Samaneh Navabpour, Chirag Ravishankar, Aayush Prakash, Katy Desai, Wallace Wu and Jithu Sam for making my time in Waterloo, an enjoyable experience. I would also like to thank Mr Somit Gupta for his guidance and help during the initial stages of moving and settling in Waterloo.

Lastly but not the least, I would like to thank my parents and siblings for motivating and encouraging me during the tough and distressing times during my masters. I simply would not be the person that I am today without their unflinching help and unconditional love. I thank my parents for the awesome education that they have provided me and guidance through every step of my life.

Dedication

Dedicated to my supporting and caring family; My parents, Jojee and Anitha Thomas, my siblings, Jobin and Anna Thomas; my grandparents M.G.Kurian and Aleyamma Kurian and my uncle and aunt, Anil and Ashwathy Kurian.

Fear not, for I am with you;
Be not dismayed, for I am your God.
I will strengthen you.
Yes, I will help you.
I will uphold you with my righteous right hand.

Isaiah 41:10

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In computing systems, *correctness* refers to the assertion that a system satisfies its specification. *Verification* is a technique for checking such an assertion. *Runtime verification* [22, 57, 11, 12, 34, 31] refers to a lightweight technique, where a *monitor* checks at run time whether or not the execution of a system under inspection satisfies a given correctness property. Runtime verification complements exhaustive verification methods, such as model checking and theorem proving, as well as incomplete solutions, such as testing and debugging. This is because exhaustive verification often requires developing a rigorous, abstract model of the system and suffers from the state-explosion problem. Testing and debugging, on the other hand, provide us with under-approximated confidence about the correctness of a system. These methods only check for the presence of defects under specific scenarios.

Most monitoring approaches in runtime verification are *event-triggered*. In these approaches, the occurrence of new events (e.g., change of value of a variable) trigger the monitor. This constant invocation of the monitor leads to *unpredictable overhead* and potentially *bursts* of monitoring intervention at run time. These defects can cause serious issues especially in real-time embedded safety/mission-critical systems. To tackle

these drawbacks, in [27], the authors propose *time-triggered* execution monitoring, where a time-triggered monitor runs in parallel with the program and samples the program's state periodically to evaluate a set of properties. Employing such a monitor results in observing bounded and predictable overhead at runtime, which are critical design parameters for a developer of embedded time-sensitive systems. Traditional approaches to monitoring such as the one used by GNU *gprof* [32], insert tracing code in the program. Once completed, 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 with the time-triggered approach is that it can incur high overhead. The sampling period essentially defines the overhead as it specifies how frequently the tracer investigates the program status. 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 however, it may still limit 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 [27], 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 [27], it is interesting to investigate different marker schemes and functions that could increase the sampling period, hence reducing the overhead incurred on the program. Some of the marker schemes considered were single increment scheme, multiple increment scheme, BITVEC$^+$ schemes, etc. The single increment scheme requires using a monitoring variable to be initialized at the start of the program and incremented once for every instrumentation. The multiple increment scheme has a monitoring variable which can be incremented several times. BITVEC$^+$ scheme, on the other hand uses a combination of setting and clearing bits as well as increments. We discuss these marker schemes and functions in Chapter 3. Experimental results explained

in Section 3.8 indicate that using different marker schemes could increase the sampling period or reduce overhead incurred on the program.

Apart from the overhead associated with time-triggered monitoring, there are certain other complexities, as well. For example, the monitor needs to run as a separate process or thread. The first drawback of such a structure is the high cost of context switching and synchronization. Moreover, synchronization data structures require the underlying operating system to provide kernel-level system call primitives and inter-process communication features. In fact, some of the widely used embedded environments (e.g., TinyOS) lack such multi-tasking features. Moreover, if the program under scrutiny becomes blocked (e.g., for I/O), the monitor continues trying to sample the program periodically. This will waste system resources. Furthermore, a monitor process coupled with a program creates a tight dependency between them at run time. For instance, if the monitor crashes while evaluating properties, it may never resume the program's normal operation. Finally, since the monitor cannot directly read the state of the program, it will keep taking samples from the program even if no new events have occurred between two samples.

In Chapter 4, we introduce a new concept called time-triggered *self-monitoring* to address the aforementioned problems. The idea is to instrument the program under scrutiny with instructions in such a way that it *self-samples* (i.e., records the program's state) itself periodically without maintaining an internal timer. In order to facilitate the program to self-sample itself, we analyze the control-flow graph of the program and compute locations where self-sampling instructions are inserted. The approach adopted in this work applies only to *sequential* programs.

To ensure optimal instrumentation, we require that the number of instrumented vertices in the control-flow graph are minimum. We show that the corresponding optimization decision procedure is NP-complete in the size of the program's control-flow graph in Section 4.2.2. To remedy the exponential complexity, we follow two approaches. First, we propose a mapping from our optimization problem to the Boolean satisfiability problem in Section 4.3.1. This mapping enables us to utilize powerful SAT-solvers to solve our optimization problem. Secondly, we propose a heuristic that finds nearly optimal solutions to the problem in Section 4.3.2.

In order to find the optimal instrumentation using the techniques mentioned above, we implemented a toolchain. The tool takes a C program and sampling period as inputs and computes the instrumentation locations where self-sampling instructions have to be inserted so that the program will self-sample itself, on execution. The experimental results discussed in Section 4.6, show that self-monitored programs perform significantly better than externally monitored programs in terms of execution time, code size, context switches, etc. This is simply due to the elimination of the cost of synchronization and context switching for programs monitored by an external process.

1.2 Summary of Contributions

The summary of contributions stated in this thesis are as follows:

- **Lowering the overhead of Time-triggered Execution Monitoring** [64]: The major contributions associated with this work are:
  - Evaluating expressiveness of different schemes for instrumentation markers. 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 that lead to the design of BITVEC$^+$.
  - Two new, expressive instrumentation marker schemes. Based on the experience looking at the different schemes, two new, expressive schemes for handling markers called BITVEC and BITVEC$^+$. Each of them can be used in different applications.
  - Theorems for failure conditions of the new schemes. Related work [27] showed that instrumentation schemes might result in livelock situations where the scheme cannot make progress or cannot solve the instrumentation problem. Analogous to this concept, failure conditions for the 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 [27] and thereby increase monotonicity by a factor of two over related approaches with similar memory demands.

- **Time-triggered Program Self-monitoring** [15]: The major contributions associated with this work are:
  - Time-triggered Self-monitoring. A new proposed monitoring technique known as Time-triggered Self-monitoring, whereby we instrument the program under inspection such that it *self-samples* its state in a periodic fashion without requiring assistance from an external monitor or internal timer.
  - Formulation of an optimization problem. We formulated an optimization problem known as Self-Monitoring Instrumentation problem (SMI) which minimizes the number of points in a program, where self-sampling instrumentation instructions must be inserted.
  - Complexity of SMI and reduction to SAT. We discovered that SMI belongs to the NP-Complete class of problems. In order to cope with the exponential complexity, we proposed a SAT-based solution and a heuristic.
  - Improvement over time-triggered monitoring using an external monitor. Experimental results show that a time-triggered self-monitored program performs significantly better in terms of execution time, binary code size and number of context switches, compared to the same program monitored by an external time-triggered monitor.

1.3 Thesis Organization

Chapter 2 presents the background work and related work on various monitoring schemes, event-based monitoring schemes, etc. Chapter 3 introduces Time-triggered Execution Monitoring. It consists of the system model, various instrumentation schemes, two new instrumentation schemes BITVEC and BITVEC$^+$. We also discuss experimental results of BITVEC and BITVEC$^+$ to compare its performance against other schemes. Chapter 4 introduces Time-triggered Self-monitoring with Minimum Instrumentation. It consists of the problem

description, complexity analysis, two solutions (SAT-based and heuristic) to cope with the exponential complexity. We also discuss experimental results to compare time-triggered self-monitoring with time-triggered monitoring using an external time-triggered monitor. In Chapter 5, we discuss the Conclusion and Future work.

Chapter 2

Background and Related Works

In this chapter, we discuss the necessary background on time-triggered program monitoring and summarize various related works. This chapter is organized as follows; in Section 2.1, we discuss related work on trace-based debugging. In Section 2.2, we discuss the related works on monitoring and runtime verification. Finally, we discuss the work conducted on time-triggered monitoring in Section 2.3.

2.1 Trace-based Debugging

A large body of work exists in the area of tracing with general work on trace-based debugging [20, 46, 53]. In [20], the authors propose a debugging technique known as *Flowback Analysis*, which allows programmers to inspect dynamic dependencies in a program execution history without the need to re-execute the program. Flowback analysis uses a traced-based approach to ensure low overhead in terms of execution time and space consumption. In [46], the authors propose a technique for efficiently tracing programs. The work proposes a technique known as *Abstract Execution* in which, a small set of events recorded during the traced program's execution, are used to generate a complete trace by re-executing certain regions of the original program. This technique reduces both cost of tracing in terms of the execution time overhead incurred on the original program and size

of the generated trace files. Another work on trace-based debugging proposed in [53] uses *checkpoints* to *incrementally* replay or restart from intermediate points in the program execution. This technique is useful when debugging process involves a lot of replays of the execution of the program and for long-running programs. Since introducing checkpoints into a program incurs high time and space costs, the authors propose *adaptive* tracing techniques that provide bounded replay times and reduced overhead on the execution of the program.

The wide use of trace-based debugging leads to a common problem regarding the placements of probes and counters optimally in a program. In [29], we see that the placement of such probes and counters efficiently in a program is hard and this results in many interesting variations of the tracing problems [9]. Different approaches to instrument and reduce the overhead incurred on the program do exist. In [44], the authors propose 3 methods of reducing overhead in terms of 1) cost of instrumentation points, 2) number of executed instrumentation points and 3) cost of the instrumentation code. The authors propose techniques such as coalescing instrumentation probes, partial in-lining to select between minimal and maximal versions of the runtime instrumentation code and partial register context storing and restoring to facilitate efficient instrumentation code. In [10], the authors propose an approach that labels execution paths in order to efficiently profile execution paths of the program. In [47], Larus and Schnarr propose a machine independent executable editing technique that provides flexibility in setting probes and hides much of the complexity involved in editing executables. In [7, 61], proposed techniques indicate flexibility in setting probes and thus reduces overhead.

Another interesting area of instrumentation to reduce overhead is dynamic binary rewriting which changes the instrumentation at run time. Pin [49] is a popular tool used for dynamic instrumentation. Pin provides an architectural independent set of API's for performing instrumentation. In [8], Bala et al. propose Dynamo which is a dynamic optimization system that is capable of improving the performance of native code while executing on the processor. DynamoRIO [1] is a tool built on the work proposed in [8]. In [17], Bruening et al. implements an interface for building external components for DynamoRIO [1]. This interface abstracts many low level details of DynamoRIO and provides a comprehensive application interface. In [51], Misurda et al. proposes a demand-driven

technique based on execution paths to improve test coverage. This approach uses dynamic instrumentation that can be included whenever needed, to maintain low overhead. Some of the other approaches of reducing overhead as a result of instrumentation include simulator-based approaches [66], time-aware instrumentation [28], and code duplication [5].

Tracing can also be implemented in hardware. Solutions such as JTAG [21], Nexus [48], and ARM CoreSight [54] with, for instance, the ETM permit inspecting and tracing the system at a hardware level.

2.2 Monitoring and Runtime Verification

Throughout the years, monitoring has become a popular area of interest and research. Monitoring techniques developed include the use of frameworks, code instrumentation, better understanding of the application and quality assurance. Of these techniques, a few studies discussed include the works of Bertino et al. work on event-based temporal objects, Janusz Sosnowski et al's work on online monitoring framework combining hardware and software and Mohlalefi Sefika et al's work on the original/intended design of software being deviated from during implementation. We will also touch on monitoring semantics by Amir Kishon et al, software tomography by Jim Bowing et al, and lastly the development of AIMS by Jerry C. Yan.

Bertino et al [13] introduced the concept of event-based temporal object model. This concept is one that is a cost efficient approach to monitoring applications and can be accomplished through selectively recording information, instead of storing the entire history of data. An approach to accomplishing this concept is to use a monitoring framework. A monitoring framework will record the history of a data object at the point which a specific event occurs.

The next study discussed is that of Janusz Sosnowski et al [60]. Sosnowski developed the idea of using both hardware and software monitoring creating an online monitoring framework. In the work, he discusses in detail three techniques which largely encompassed the online monitoring framework.

Research done by Mohlalefi Sefika et al. [59] discusses how software system implementation often results in a system that is divergent from the envisioned design. Sefika discusses the benefits of using codified design principles alongside regular updates and checkups in order to ensure that it meets the intended design and functionality. Sefika introduces the idea of using visualizations and logic-based static analysis in order to help ensure deviation occurs as infrequently as possible. This approach also enables the ability to view code from many different standpoints.

The next study that will be momentarily looked at is based on monitoring semantics. The notion of monitoring semantics by Amir Kishon suggests using an extension of language's ordinary semantics in order to observe activity [43]. Kishon proposes that the monitoring of semantics be implemented into a program as it allows for modularity and safety.

A subsequent technique for monitoring is that of software tomography presented by Jim Bowring et al [16]. The idea behind software tomography is that it creates a means of monitoring which is nominally invasive and does not present much impact. The technique creates subtasks out of tasks and disperses the subtasks to one instance of software for monitoring purposes. The retrieval of monitoring information is then generated by fusing together the information from each subtask.

Another interesting work on monitoring is that of Jerry C. Yan who indicated that seizing and viewing the execution of an application can be a valuable source of information [68]. To validate his findings, Yan created a parallel monitoring system by the name of AIMS, which captures a collection of data for MPI programs. In order to monitor the sections of an application which are of interest, instrumentation must be done before execution. This enables the AIMS application to generate performance data, which Yan claims is important for understanding and observing the behavior of an application. It also allows the user to trace the sequence of events and operations that occur in the application, if needed.

Another closely related area of research linked to the work presented in this thesis is runtime verification. In classic runtime verification [57], a system consists of an external observer, called the monitor. This monitor is normally a finite state automaton based on a set of properties. Most of the monitors used for runtime verfication are event-triggered [45]

in the sense that, any change in the state of the system invokes the monitor for analysis.

From the logical and language points of view, runtime verification has mostly been studied in the context of Linear Temporal Logic (LTL) [11, 35, 38, 31, 36, 63] and in particular safety properties [58, 37]. Other languages and frameworks have also been developed for facilitating specification of temporal properties [70, 41, 42]. Runtime verification of ω-languages was considered in [23]. In [26], the authors address runtime verification of safety-progress [50, 19] properties.

Overall, the techniques and concepts mentioned above only begin to scratch the surface on the work developed throughout the years. Being aware of the research done in the field of monitoring allows us to both apply the knowledge we have gained as well as further the discussion to explore and enhance different monitoring methods, two of which will be discussed in-depth later.

2.3 Time-triggered Monitoring

Time-triggered approaches are popularly used in areas where full accuracy for recreating execution flows as necessary for debugging and monitoring is not needed. Some of the related works that use a time-triggered approach are discussed as follows. In [32], the authors propose *gprof*, a call graph execution profiler which uses a time-triggered approach to profile the routines in the program. In [67], the author proposes a time-triggered based profiler for Java Virtual Machines (JVM). The time-triggered profiler facilitates the compiler in making better informed optimization decisions by providing it with continuous system performance measurements in real time. In [69], Zhong and Chang propose an improved systematic sampling approach to overcome the high overhead associated with reuse distance measurements used in program analysis and optimizations. In [4], Anderson et al. describes a continuous profiling infrastructure which performs its profiling in a sampling-based approach. The continuous profiler infrastructure supports a wide variety of systems and uses high sampling rates while maintaining low overhead to profile the system. Analysis tools augmented with the continuous profiler system produce precise and accurate sampled measurement data which helps users to easily identify performance

bottlenecks in their application. In [24], Dean et al. propose a hardware solution for facilitating sampling-based instruction level profiling for out-of-order processors. The resulting profiled data is useful in analyzing latencies and concurrency issues, cache miss data, branch history, etc. Other works that use a time-triggered approach include flexible value sampling [18], approximating the calling context tree [6].

Time-triggered monitoring approaches are used in real-time systems due to the predictable and bounded overhead incurred on the program under inspection. In [27, 64, 14], the authors introduce a time-triggered execution monitoring and runtime verification techniques. They propose a framework that allows quantitative analysis of issues tackled in time-triggered techniques. In the same context, in [55, 56], the authors propose the language Copilot for developing hard real-time monitors. The goal of this language is to develop programs where the monitor does not affect the functionality of the program with minimal overhead incurred to the program

Finally, in [39], the authors propose a technique that uses control theory for discrete event systems to control the overhead of software monitoring. In this work [39], overhead is controlled by temporarily disabling/enabling the invoking of the monitor so that the overhead is within the bounds of user specified threshold. Another relevant work to this line of research is [62], where the authors propose runtime verification using state estimation. In particular, they use Hidden Markov Models (HMM) to check whether states that were missed due to disabling/enabling of the invoking of the monitor based on the technique used in [39], resulted in violation of user-specified properties.

Chapter 3

Time-triggered Execution Monitoring

In this chapter, we introduce Time-triggered Execution Monitoring and compare various schemes used to effectively reduce the overhead incurred on an application program as a result of monitoring the program in a time-triggered fashion [64]. The organization of this chapter is as follows: in Section 3.1, we present the overview and background of time-triggered execution monitoring, We introduce time-triggered execution monitoring and discuss the problem motivation in Section 3.2. In Section 3.3, we discuss the system model used to formalize time-triggered execution monitoring and terminology used in the model. We discuss the various instrumentation schemes and its measure of expressiveness in Section 3.4. We introduce the two new novel instrumentation schemes based on bit vectors in Section 3.5. Finally, we discuss the framework used to compare the various instrumentation schemes, experimental method and results in Section 3.6, 3.7 and 3.8, respectively.

3.1 Overview

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 [30]. Consequently, it is important to investigate new methods for debugging to increase productivity.

It is a popular practice to use software instrumentation as the debugging technique to perform tracing. In this context, the developer instruments a control flow graph and then executes the instrumented control flow graph to produce a trace. The developer uses the generated trace 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 was suggested [27] 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. The sampling period of the tracer is inversely proportional with the overhead for debugging and tracing. Traditional approaches to monitoring, such as the approach by GNU(gprof) [32], insert tracing code into the program. This makes it 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 gain in predictability, time-triggered monitoring approach could incur high overhead on the program under inspection. The sampling period, which specifies how often the tracer investigates the program status, defines the overhead incurred on the program. The program incurs more overhead as the tracer interrupts the program more frequently. In real-time systems, the amount of slack available to the program with respect to the deadline determines the overhead that the program can tolerate. The more slack a program has with respect to its deadline, the more overhead is tolerable to the program. Hence, time-triggered monitoring approach with short sampling periods or high overheads limits its applicability to those systems and real-time applications that have sufficient slack. Thus, reducing the overhead of the time-triggered monitoring approach increases its range of applicability.

Related work [27] proposes an approach where markers are introduced 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 [27], the work presented in this chapter looks into different marker schemes and functions to increase the sampling period, hence reducing the overhead incurred on the program under inspection. For example, we introduce a

new instrumentation marker scheme known as BITVEC$^+$ scheme, which uses a combination of setting and 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 helped achieve higher sampling periods compared to related approaches and hence having the least overhead when compared to the related approaches.

3.2 Time-triggered 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 the current 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 time-triggered execution monitoring is the bounded overhead of the monitoring system. The overhead linearly decreases with the sampling period and sample size. The result is a high sampling period which generally leads to lower overhead Ð in comparison to low sampling periods, assuming the sample sizes are the same.

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 3.1 shows a simple C program with three basic blocks labeled A, B, and C. Figure 3.1(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 3.1(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 period for the program will be $\Delta t = 2$ (more details on the formal model are in Section 3.3).

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.

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

Listing 3.1: Illustrative example.

In the example, a marker m_1 is used to instrument the vertex C. Figure 3.1(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 3.1(d) shows that introducing the marker increases the sampling period to $\Delta t = 4$, because only after four 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.

3.3 System Model & Terminology

The following describes the system model and terminology. The model closely follows the one presented in [27]. However, we extend the model with a generic instrumentation function to manipulate markers.

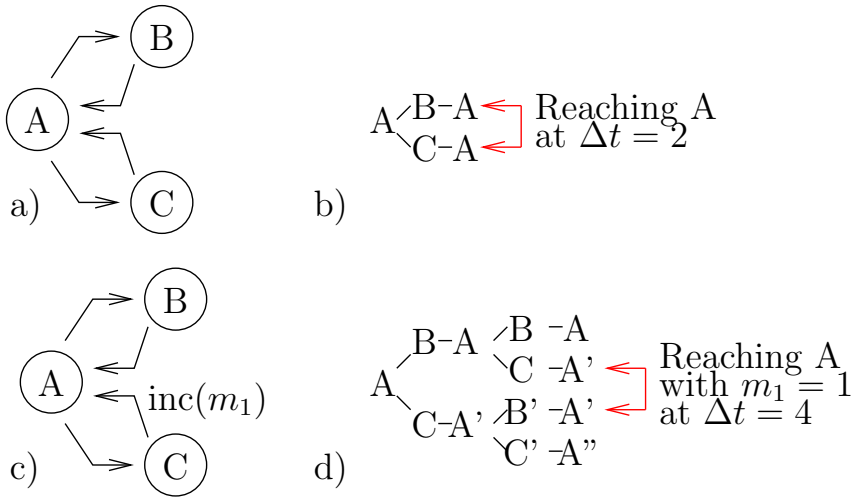

Figure 3.1: Example of a single instrumentation to extend Δt

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 the program. An edge $e := \langle v_s, v_d \rangle$ represents a transition from a 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 [2]. 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 execution times 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, etc. We take a sample 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 3.2 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 and 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 3.1(a), 3.1(b), and 3.1(c) show an instrumentation step. Figure 3.1(a) shows the original control flow graph. In Phase (a) of the instrumentation step, we find the path pairs shown in Figure 3.1(b). An algorithm then decides to increment vertex C (Phase (b)), and finally applies the marker function to C. Figure 3.1(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 3.1(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 3.1(d).

3.4 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 [27]. 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 occur. 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.

3.4.1 Overview of Different Schemes

Single Increment Scheme. The first scheme is the *single increment scheme* (proposed in [27]). In this scheme, a marker is a single value initially set to zero. We define a function $\mathcal{I}$ 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 [27]) 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 [10]). 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 schemes 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.

3.4.2 Comparison in Expressiveness

Table 3.1 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 3.2 which is the counter example in which the single increment scheme is superior to the assignment scheme.

	Single Increment	**Multiple Increment**	**Assignment**	**Bit Vector**
Single Increment	-	Less powerful	Figure 3.2	Figure 3.2
Multiple Increment	Figure 3.4	-	Figure 3.5	Figure 3.2
Assignment	Figure 3.4	Figure 3.6	-	Less powerful
Bit vector	Figure 3.4	Figure 3.6	Figure 3.5	-

Table 3.1: Comparison table for different schemes

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 3.2 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.

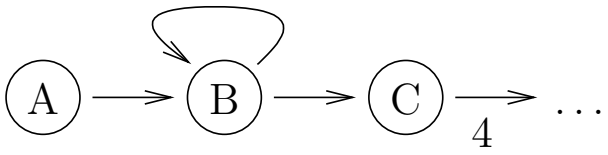

Figure 3.2: Counter example 1: self loop.

Figure 3.3 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 $A \to B \to C$ intersects with $A \to B \to B \to C$ at $\Delta t = 4$.

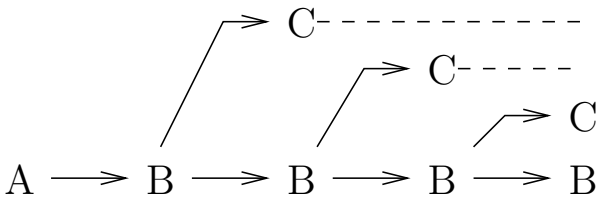

Figure 3.3: Timing diagram for Figure 3.2.

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.

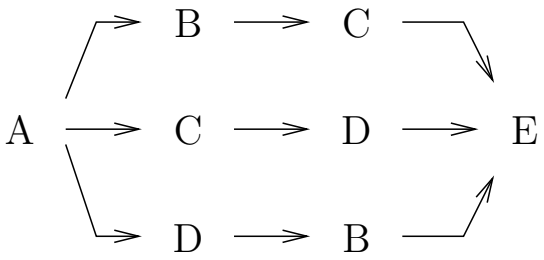

Figure 3.4: Counter example 2: Timing diagram that shows a permutation.

Figure 3.4 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 [27], and we list it here for the sake of completeness.

The graph contains three paths $p_1 = A \to B \to C \to E, p_2 = A \to C \to D \to E$, and $p_3 = A \to D \to B \to E$. 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 the 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.

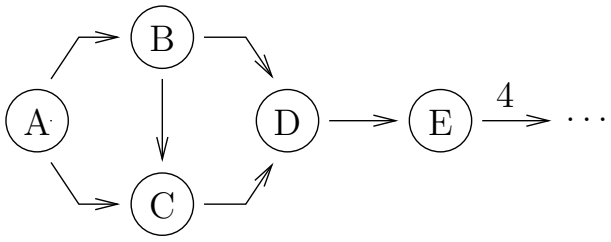

Figure 3.5: Counter example 3: delay.

Figure 3.5 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 $p_1 = A \to B \to D$ and $p_2 = A \to C \to D$. 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 $p_3 = A \to B \to D \to E, p_4 = A \to C \to D \to E,$ and $p_5 = A \to B \to C \to D \to E$. 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 3.6 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

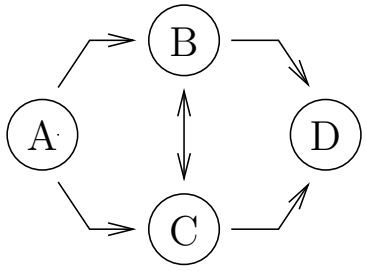

Figure 3.6: Counter example 4: diamond.

vertices in the other path of path pair, in such case any instrumentation (single or multiple increment, whereas Figure 3.4 only applies to single increment) will lead to the same value of the marker at the end. Figure 3.6 generate this kind of path pairs which are shown in Figure 3.7.

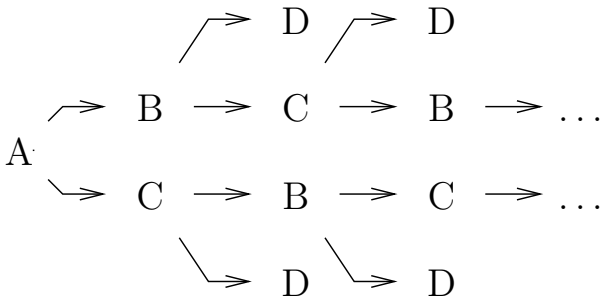

Figure 3.7: Timing diagram for Figure 3.6.

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.

3.5 Two Instrumentation Schemes based on Bit Vectors

Based on the insights gained from observing the expressiveness of different schemes as discussed in Section 3.4 and especially Table 3.1, 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.

3.5.1 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 $ftlo_p^{\#} : V(p) \rightarrow \mathbb{Z}^+$ (**f**inish **t**ime of **l**ast **o**ccurrence) is a hash function that maps each unique vertex v that lies on path p to the finish time of the last occurrence of v on path p. For example, let $p = [A \rightarrow B \rightarrow B \rightarrow A]$ and each basic block have a BCET of 1 time unit, $ftlo_p^{\#}(A) = 4$ and $ftlo_p^{\#}(B) = 3$, implying that the finish time of last occurrence of A on path p is 4 and finish time of last occurrence of B on path p is 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 check $C_2^{|V(p)|}$ combinations of vertices on path p where p can be either p_1 or p_2 and store all pairs of vertices that satisfy the condition that one vertex distinctively occurs last on one path and the other distinctively occurs last on the other path, into a temporary set V_{able}. On completion of checking all $C_2^{|V(p)|}$ combinations of vertices, 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 polynomial 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. The nested for loop used for comparing $C_2^{|V(p)|}$ combinations of vertices on path p where p is either p_1 or p_2 results in $O(|V|^2)$ complexity.

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. $\nexists v_1, v_2 \in \{V(p_1) \cup V(p_2)\}$ *such that:* $ftlo^{\#}_{p_1}(v_1) < ftlo^{\#}_{p_1}(v_2)$*:* $ftlo^{\#}_{p_2}(v_2) < ftlo^{\#}_{p_2}(v_1)$ *for* $v_1 \neq v_2$

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

Function 1 Instrument graph with BITVEC

Input: Control flow graph G, paths p_1, p_2

Output: Instrumented control flow graph G'

1: $\delta \Leftarrow (V(p_1) \cup V(p_2)) - (V(p_1) \cap V(p_2))$
2: **if** $\delta = \emptyset$ **then**
3: $\mathcal{V} \leftarrow V(p_1)$
4: **for** $i = 0$ *to* $|\mathcal{V}|$ **do**
5: **for** $j = i + 1$ *to* $|\mathcal{V}|$ **do**
6: **if** $[ftlo^{\#}_{p_1}(\mathcal{V}[i]) < ftlo^{\#}_{p_1}(\mathcal{V}[j])$ && $ftlo^{\#}_{p_2}(\mathcal{V}[j]) < ftlo^{\#}_{p_2}(\mathcal{V}[i])]$ ||
7: $[ftlo^{\#}_{p_1}(\mathcal{V}[i]) > ftlo^{\#}_{p_1}(\mathcal{V}[j])$ && $ftlo^{\#}_{p_2}(\mathcal{V}[j]) > ftlo^{\#}_{p_2}(\mathcal{V}[i])]$ **then**
8: add $\mathcal{V}[i]$ and $\mathcal{V}[j]$ to V_{able}
9: **end if**
10: **end for**
11: **end for**
12: **if** $V_{able} = \emptyset$ **then**
13: BITVEC terminates
14: **else**
15: add first two vertices added to V_{able} to V_{instr}
16: **end if**
17: **else**
18: add one vertex of δ to V_{instr}
19: **end if**
20:
21: **if** $|V_{instr}| = 2$ **then**
22: set a bit in one vertex, clear the same bit in the other vertex
23: **end if**
24: **if** $|V_{instr}| = 1$ **then**
25: set a bit for the vertex
26: **end if**

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 $\exists v_1, v_2 \in \{V(p_1) \cup V(p_2)\}$ such that $[ftlo^{\#}_{p_1}(v_1) < ftlo^{\#}_{p_1}(v_2)] \wedge [ftlo^{\#}_{p_2}(v_2) < ftlo^{\#}_{p_2}(v_1)]$ for $v_1 \neq v_2$ 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 $\exists v_1, v_2 \in \{V(p_1) \cup V(p_2)\}$ such that $[ftlo^{\#}_{p_1}(v_1) < ftlo^{\#}_{p_1}(v_2)] \wedge [ftlo^{\#}_{p_2}(v_2) < ftlo^{\#}_{p_2}(v_1)]$ for $v_1 \neq v_2$: 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 with the assurance that one wont overwrite the other, hence making it distinguishable at the time of sampling.

- Case 3 $(V(p_1) \cup V(p_2)) - (V(p_1) \cap V(p_2)) \neq \emptyset$ and $\exists v_1, v_2 \in \{V(p_1) \cup V(p_2)\}$ such that $[ftlo^{\#}_{p_1}(v_1) < ftlo^{\#}_{p_1}(v_2)] \wedge [ftlo^{\#}_{p_2}(v_2) < ftlo^{\#}_{p_2}(v_1)]$ for $v_1 \neq v_2$: 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. □

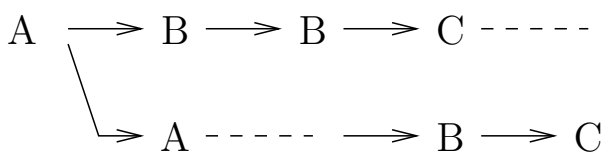

Figure 3.8: Two paths that cannot be instrumented using BITVEC .

Example 1. *Figure 3.8 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 3.8, condition 2 of Theorem 1 also holds as $\nexists v_1, v_2 \in \{V(p_1) \cup V(p_2)\}$ *such that* $[ftlo^{\#}_{p_1}(v_1) < ftlo^{\#}_{p_1}(v_2)] \wedge [ftlo^{\#}_{p_2}(v_2) < ftlo^{\#}_{p_2}(v_1)]$ *for* $v_1 \neq v_2$. *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.

3.5.2 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 between 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

the sampling period compared to BITVEC but introduces interference due to the use of increment based schemes.

Terminology and Definitions. The inputs, outputs and $ftlo_p^{\#}(v)$ 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 check $C_2^{|V(p)|}$ combinations of vertices on path p where p can be either path p_1 or path p_2 and store all pairs of vertices that satisfy the condition that one vertex distinctively occurs last on one path and the other distinctively occurs last on the other path, into a temporary set V_{able}. On completion of checking all $C_2^{|V(p)|}$ combinations of vertices, 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 polynomial with respect to the number of vertices in V. Calculating δ_{freq} is linear, because it involves executing 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. The nested for loop used for comparing $C_2^{|V(p)|}$ combinations of vertices on path p where p is either p_1 or p_2 results in $O(|V|^2)$ complexity.

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. $\delta_{freq}(p_1, p_2) = \emptyset$

Function 2 Instrument graph with BITVEC$^+$

Input: Control flow graph G, paths p_1, p_2

Output: Instrumented control flow graph G'

1: $\delta \Leftarrow \delta_{freq}(p_1, p_2)$
2: $\delta' \Leftarrow \delta - \{V(p_1) \cap V(p_2)\}$
3: **if** $\delta' = \emptyset$ **then**
4: $\quad \mathcal{V} \leftarrow V(p_1)$
5: $\quad$ **for** $i = 0$ *to* $|\mathcal{V}|$ **do**
6: $\quad\quad$ **for** $j = i + 1$ *to* $|\mathcal{V}|$ **do**
7: $\quad\quad\quad$ **if** $[ftlo^{\#}_{p_1}(\mathcal{V}[i]) < ftlo^{\#}_{p_1}(\mathcal{V}[j])$ && $ftlo^{\#}_{p_2}(\mathcal{V}[j]) < ftlo^{\#}_{p_2}(\mathcal{V}[i])]$ ||
8: $\quad\quad [ftlo^{\#}_{p_1}(\mathcal{V}[i]) > ftlo^{\#}_{p_1}(\mathcal{V}[j])$ && $ftlo^{\#}_{p_2}(\mathcal{V}[j]) > ftlo^{\#}_{p_2}(\mathcal{V}[i])]$ **then**
9: $\quad\quad\quad$ add $\mathcal{V}[i]$ and $\mathcal{V}[j]$ to V_{able}
10: $\quad\quad$ **end if**
11: $\quad\quad$ **end for**
12: $\quad$ **end for**
13: $\quad$ **if** $V_{able} = \emptyset$ **then**
14: $\quad\quad$ **if** $\delta = \emptyset$ **then**
15: $\quad\quad\quad$ BITVEC$^+$ terminates
16: $\quad\quad$ **else**
17: $\quad\quad\quad$ add one vertex of δ to V_{instr} and set increment_flag to *true*
18: $\quad\quad$ **end if**
19: $\quad$ **else**
20: $\quad\quad$ add first two vertices added to V_{able} to V_{instr}
21: $\quad$ **end if**
22: **else**
23: $\quad$ add one vertex of δ' to V_{instr}
24: **end if**

```
25: if increment_flag is true then
26:     initialize marker i at the starting vertex of G
27:     perform i++ at v ∈ V_instr
28: else
29:     if |V_instr| = 2 then
30:         set a bit in one vertex, clear the same bit in the other vertex
31:     end if
32:     if |V_instr| = 1 then
33:         set a bit for the vertex
34:     end if
35: end if
```

2. $\nexists v_1, v_2 \in \{V(p_1) \cup V(p_2)\}$ *such that:* $ftlo^{\#}_{p_1}(v_1) < ftlo^{\#}_{p_1}(v_2)$: $ftlo^{\#}_{p_2}(v_2) < ftlo^{\#}_{p_2}(v_1)$ *for* $v_1 \neq v_2$

Proof. The condition $\delta_{freq}(p_1, p_2) = \emptyset$ signifies that the order of vertices on the 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 [27]. □

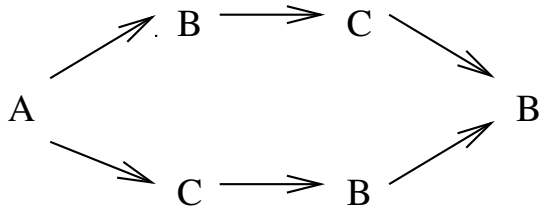

Figure 3.9: Two paths that cannot be instrumented using BITVEC$^+$.

Example 2. *Figure 3.9 shows two paths that cannot be instrumented using* BITVEC$^+$ *. Note that all the vertices have a delay of 1. 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 3.9, condition 2 of Theorem 2 also holds as $\nexists v_1, v_2 \in \{V(p_1) \cup V(p_2)\}$ *such that* $[ftlo^{\#}_{p_1}(v_1) < ftlo^{\#}_{p_1}(v_2)] \wedge [ftlo^{\#}_{p_2}(v_2) < ftlo^{\#}_{p_2}(v_1)]$ *for* $v_1 \neq v_2$. *Since both conditions hold, the example cannot be instrumented with* BITVEC$^+$.

3.6 Framework

To validate the theorems and the concepts of this work, we extended an existing instrumentation engine [27]. 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.

3.7 Experimental Method

In this section, we discuss the parameters, metrics and the experimental setup used to compare the performance of BITVEC and BITVEC$^+$ over the schemes introduced in related work [27].

For the input set, we generated about 5000 control flow graphs using a modified version of Task Graphs For Free [25]. Each control flow graph has on an average 114 basic blocks and 218 edges. The control flow graphs follow C program flows [65]. Control flow graphs of real programs from test benchmarks are future currently in progress. The parameters used for comparing BITVEC and BITVEC$^+$ with other schemes are the input control flow graph and the instrumentation scheme (either increment based with 10 or 25 markers, BITVEC , or BITVEC$^+$). 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 [27]; except that we chose the name monotonicity instead of usability. We describe the metrics in the sections below.

3.7.1 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 [27].

3.7.2 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.

3.7.3 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: $mu = \frac{\Delta t}{mem}$.

3.8 Experimental Results and Analysis

We used the experimental methods as discussed in Section 3.7 to compare the results of the BITVEC and BITVEC$^+$ with the increment based schemes discussed in [27].

The instrumentation performance of BITVEC and BITVEC$^+$ are much better than the single and multi-increment schemes even if they use multiple markers. Figure 3.10 shows the comparison of instrumentation performance for different schemes relative to the single increment scheme, according to the performance metric defined in Subsection 3.7.1. The higher the sampling period, the better, resulting in lower overhead incurred. As seen in the graph, the BITVEC and BITVEC$^+$ perform nearly 1.75 and 2 times, respectively, faster than the single increment scheme.

Figure 3.11 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

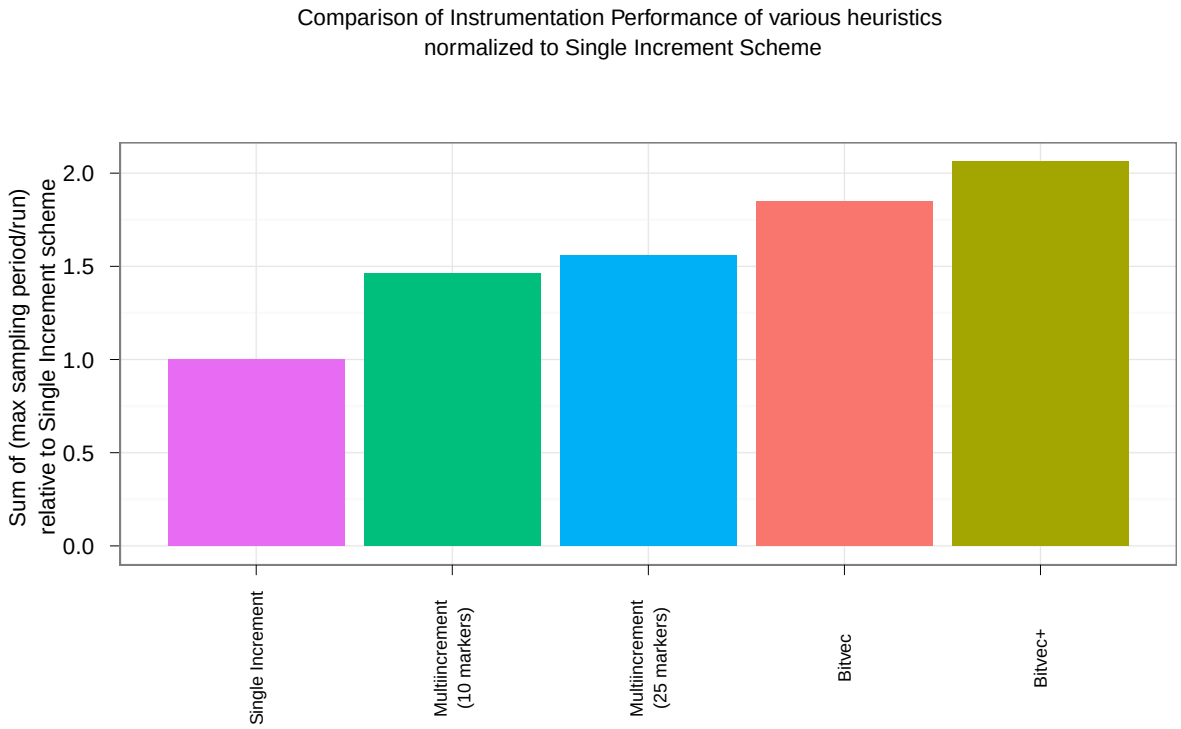

Figure 3.10: Comparing achieved sampling period of different schemes

period, the better, because it results in lower 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 assigned to clear the same bit) such that one does not overwrite the bit once set or cleared by the other. 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

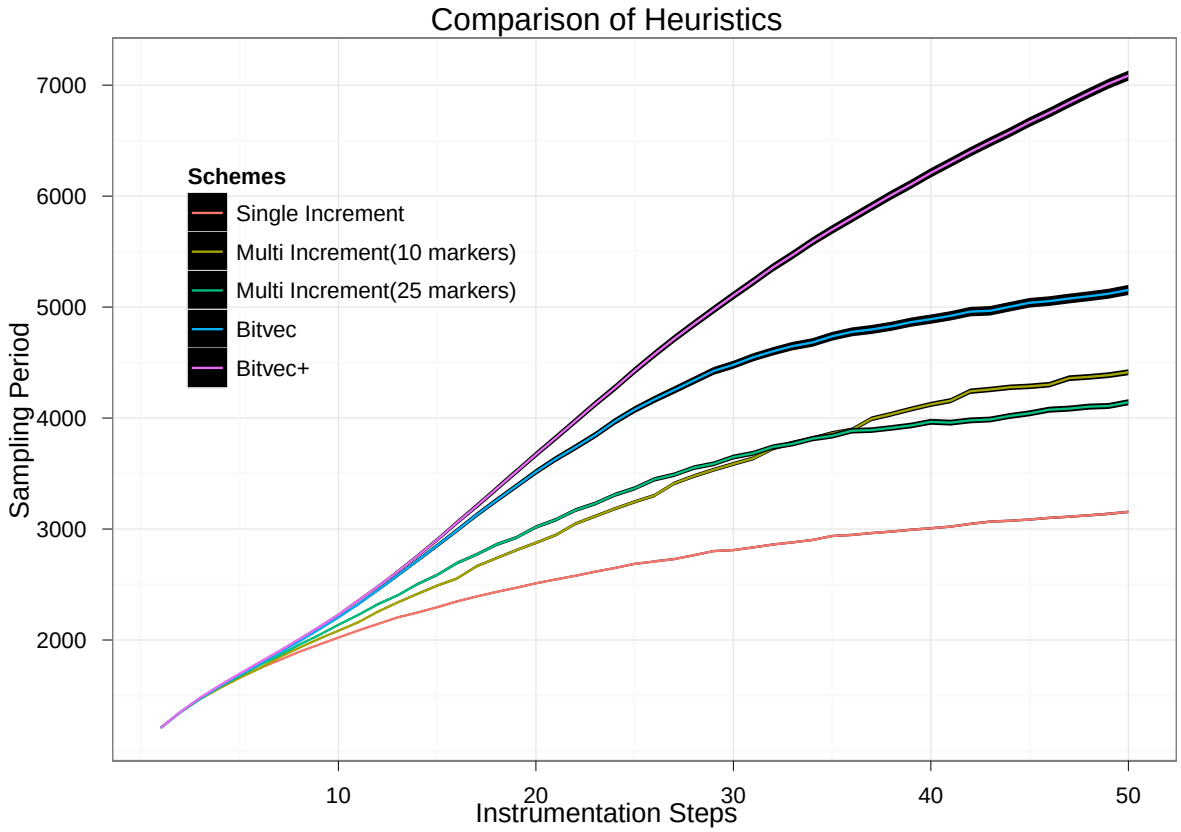

Figure 3.11: Comparing achieved sampling period of different schemes

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

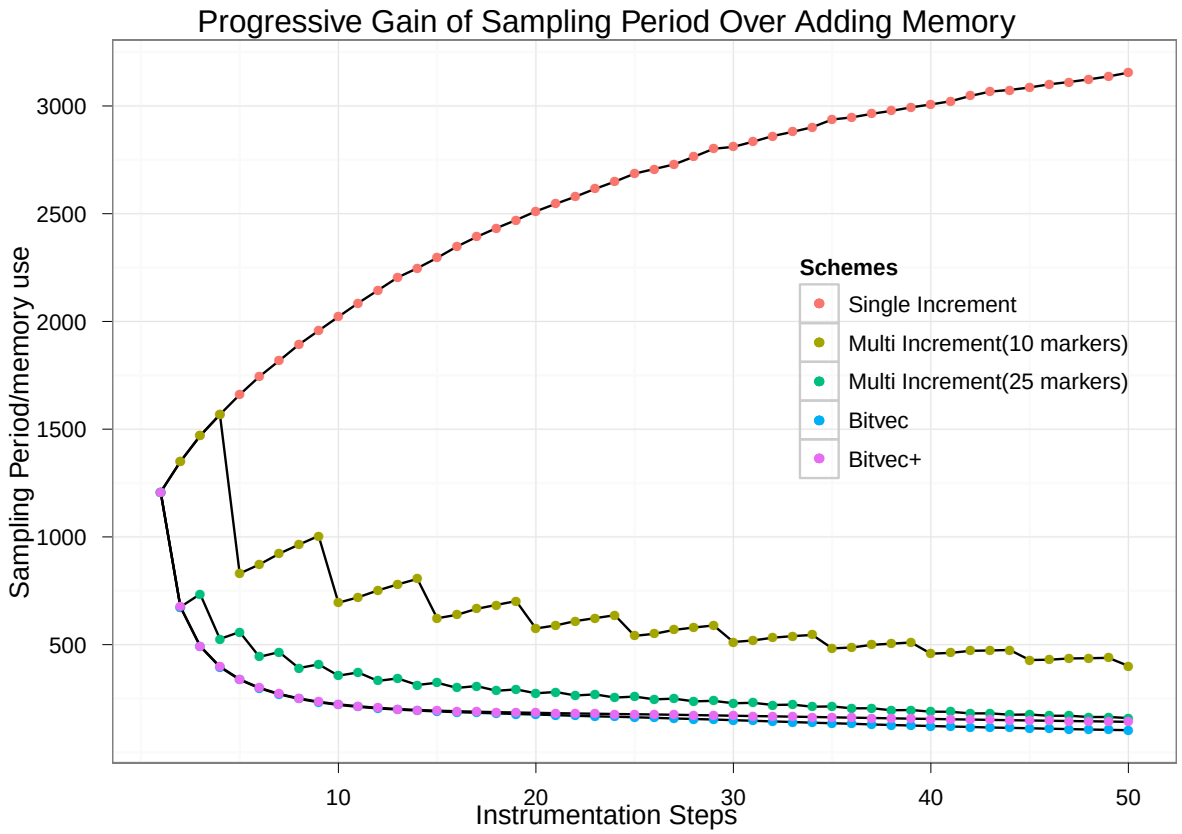

Figure 3.12: Progressive gain of sampling period over adding memory

constant (one marker). Also, 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 3.11). 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.

The interference level of BITVEC is very low or none because BITVEC eliminates direct

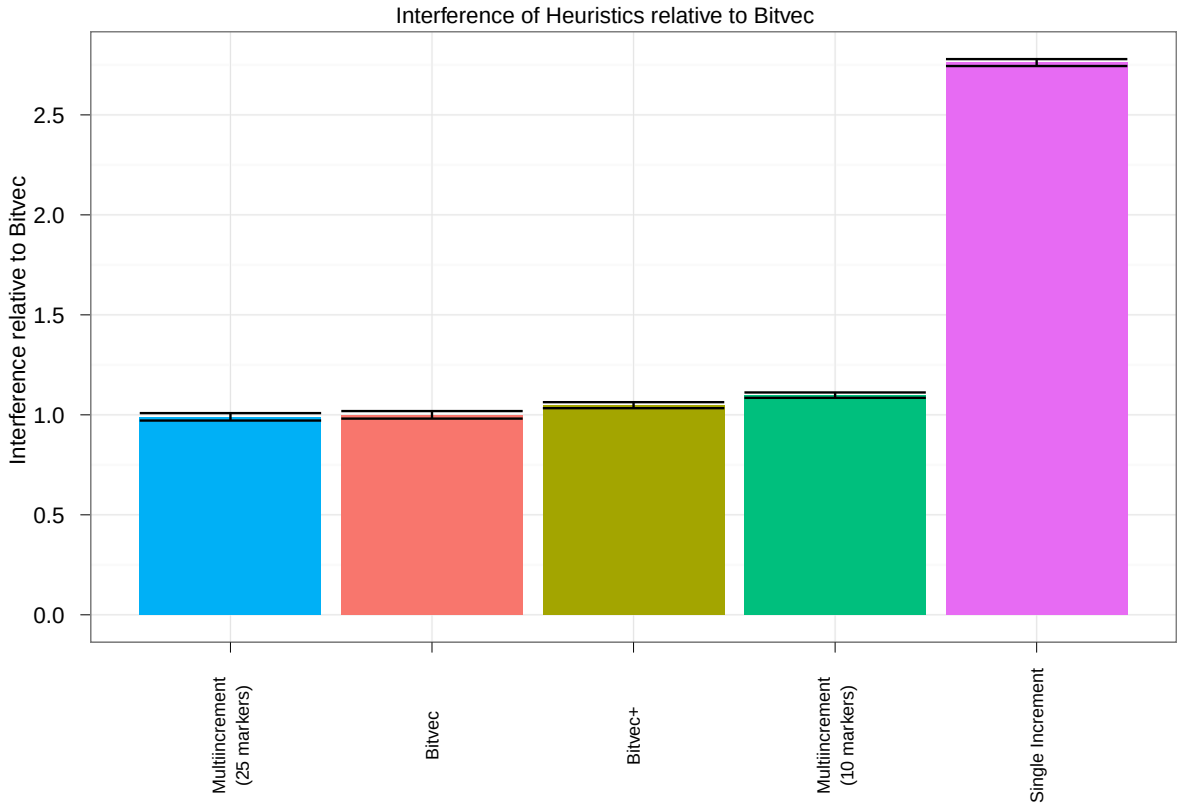

Figure 3.13: Interference of different heuristics relative to Bitvec

as well as indirect interference to a great extent as conjectured in Section 3.5. Figure 3.13 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 3.13 show the standard error of the mean. The figure shows 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 3.14

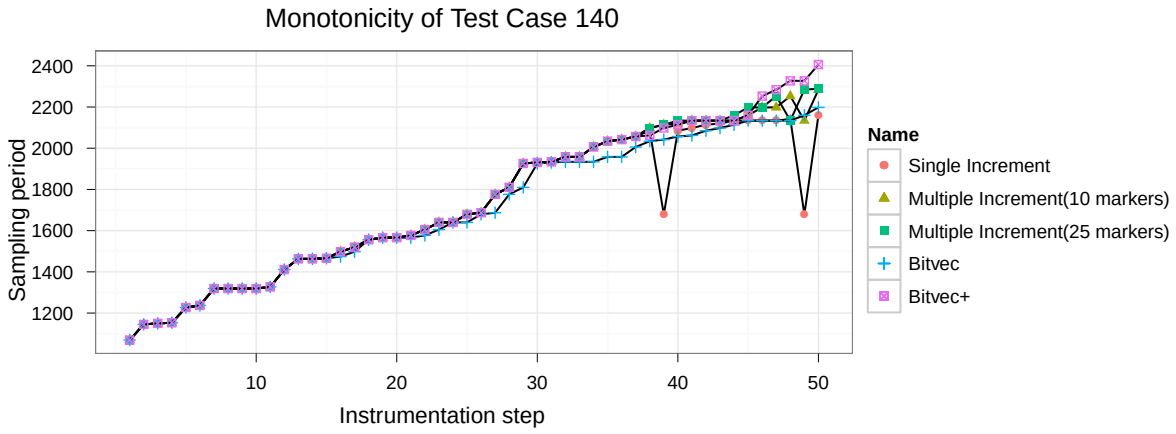

Figure 3.14: All schemes leading to almost same sampling period

to 3.17 shows the monotonicity of certain interesting test cases. The x-axis shows the number of instrumentation steps. The y-axis shows the sampling period.

Figure 3.14 shows the monotonic increase in test case 140. Figure 3.14 shows a case where all the schemes attain the sampling period at the end of 50 steps. It can also be observed that the single increment scheme encounters more interference compared to other schemes as indicated by the two downward spikes.

Figure 3.15 shows test case 14. The case is interesting, because it shows the erratic behaviour of single increment schemes (see steps 10 to 50). Figure 3.15 shows that the BITVEC and BITVEC$^+$ both follow the technique until BITVEC must terminate. BITVEC 's termination is due to its inability to 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.

Figure 3.16 shows a case where the BITVEC and BITVEC$^+$ terminate at early instrumentation steps while single increment scheme continues to solve instrumentation problems upto 50 steps. One can reason out the occurrence of such a test case by the argument that, the instrumentations that BITVEC and BITVEC$^+$ pick while solving initial instrumentation problems were “smarter" than the instrumentations made by the single increment scheme, hence resulting in high sampling periods within 30 instrumentation steps. Another evi-

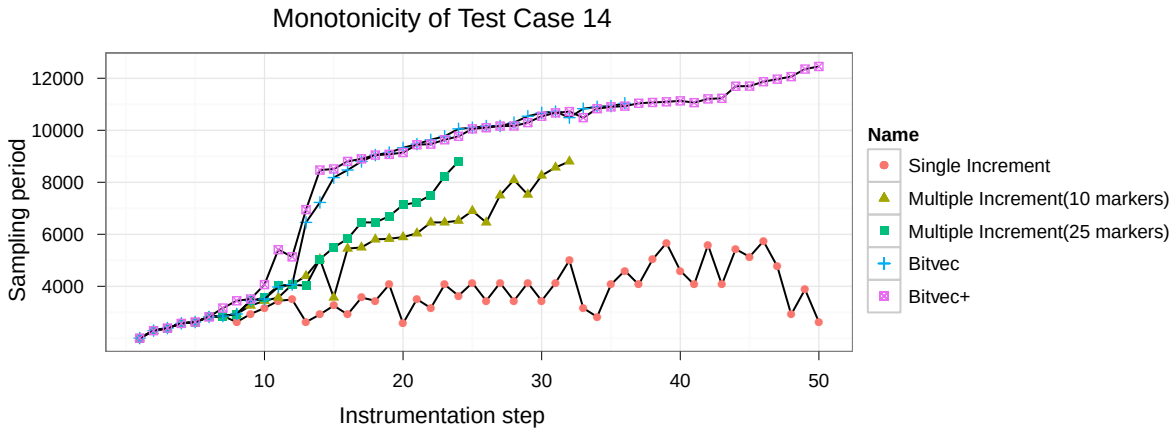

Figure 3.15: Single marker showing erratic behaviour.

dence of the "smart" decisions made by BITVEC and BITVEC$^+$ over the single increment scheme is the high interference in the sampling period of the single increment scheme from steps 10 to 50.

Figure 3.17 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	205592	0.987	0.180
2	Increment	2621	0.013	0.0023

Table 3.2: Comparing when one technique is superior than the other in the empirical data

Table 3.2 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 98.7% of the test cases used during experimentation while increment was superior in the remaining 1.3%. The reason behind the superiority of BITVEC over increment is that BITVEC can empirically instrument more cases than the single increment scheme

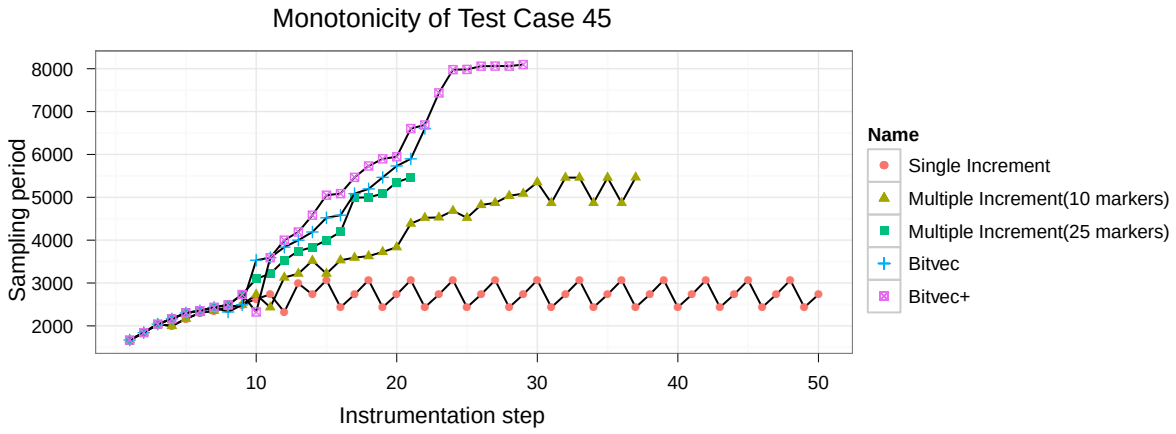

Figure 3.16: BITVEC$^+$ and BITVEC instrumenting "smarter" than single increment scheme

(See Theorem 1 vs the path pair termination in [27]). This result shows that BITVEC is reliable in producing higher sampling periods and hence reducing the overhead for sampling based monitoring.

3.9 Inference & Discussion

Some interesting observations can be made from the results obtained. We study the monotonic behaviour of BITVEC and BITVEC$^+$ and observe that these schemes on an average make better decisions (Figure 3.16) which solve a number of hidden paths, as mentioned in related work [27], hence suggesting that these schemes result in low overhead incurred on the application program.

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 3.15, the graphs of BITVEC and BITVEC$^+$ are the same until a certain instrumentation step after which

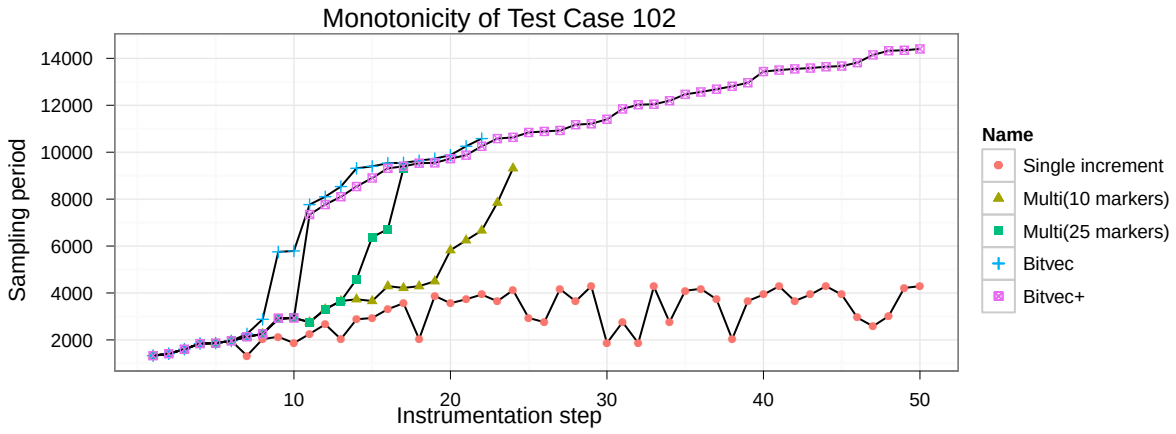

Figure 3.17: BITVEC$^+$ far superior than the others.

BITVEC can no longer instrument while BITVEC$^+$ uses increment based scheme to further instrument the graph. However, Figure 3.11 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 3.15. 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.

Chapter 4

Time-triggered Self Monitoring with Minimum Instrumentation

In this chapter, we introduce *time-triggered self-monitoring* [15], a technique that facilitates the program under inspection to *self-sample* itself periodically without maintaining an internal timer. In order to evaluate the effectiveness of the self-monitoring scheme, we compare the overhead incurred on the application program due to monitoring using the self-monitoring scheme and monitoring using the external time-triggered monitor. The organization of this chapter is as follows: in Section 4.1, we present the overview and background of time-triggered self monitoring. In Section 4.2, we formally define the notion of time-triggered self-monitoring and analyze the complexity of identifying the minimum number of instrumentation points for enabling self-monitoring. In Section 4.3, we introduce our SAT-based solution and the heuristic. We describe the framework used to instrument programs in order to facilitate it to *self-sample* itself periodically in Section 4.4. In Section 4.5, we discuss the experimental methods and settings. The results of experiments are analyzed in Section 4.6. Finally, in Section 4.7, we discuss the inference obtained as a result of studying time-triggered self-monitoring scheme.

4.1 Overview

Runtime monitoring aims at analyzing the well-being of a system at run time in order to detect errors and steer the system towards a healthy behavior. Such monitoring is a complementary technique to other approaches for ensuring correctness, such as exhaustive verification methods, model checking and theorem proving, as well as incomplete solutions, such as testing and debugging. This is because exhaustive verification often requires developing a rigorous, abstract model of the system and suffers from the state-explosion problem. Testing and debugging, on the other hand, provide us with under-approximated confidence about the correctness of a system, as these methods only check for the presence of defects under specific scenarios.

Most monitoring approaches in runtime verification are *event-triggered*. In these approaches, the occurrence of new events (e.g., change of value of a variable) triggers the monitor. This constant invocation of the monitor leads to *unpredictable overhead* and potentially *bursts* of monitoring intervention at run time. These defects can cause serious issues especially in real-time embedded safety/mission-critical systems. To tackle these drawbacks, in [14, 27], the authors propose *time-triggered* runtime verification and execution monitoring, where a time-triggered monitor runs as a separate process in parallel with an application program under scrutiny and samples the program's state periodically to evaluate a set of properties. Applying this technique in a computing system results in obtaining bounded and predictable overhead. Gaining such characteristics for overhead is highly desirable for designing and engineering time-critical applications, such as safety-critical embedded systems.

Although time-triggered monitoring results in obtaining a monitor with predictable overhead and probe effects, it introduces certain complexities as well. For example, the monitor needs to run as a separate process or thread. The first drawback of such a structure is the high cost of context switching and synchronization. Moreover, synchronization data structures require the underlying operating system to provide kernel-level system call primitives and inter-process communication features. In fact, some of the widely used embedded environments (e.g., TinyOS) lack such multitasking features. Moreover, if the program under scrutiny is blocked (e.g., for I/O), the monitor continues trying to sam-

ple the program periodically. This will waste system resources. Furthermore, a monitor process coupled with a program creates a tight dependency between them at run time. For instance, if the monitor crashes while evaluating properties, it may never resume the program's normal operation. Finally, since the monitor cannot directly read the state of the program, it will keep taking samples from the program even if no new events have occurred between two samples.

To address the aforementioned problems, in this chapter, we introduce a new concept called *time-triggered self-monitoring*. Our idea is to instrument the program under scrutiny with instructions in such a way that it *self-samples* (i.e., records the program's state) itself periodically without maintaining an internal timer. In other words, the time-triggered monitor is weaved into the program. The main challenges in instrumenting the program for enabling self-monitoring are the following:

1. (Correctness) How should the program be instrumented such that the time interval between two successive self-sampling points does not exceed the desired sampling period? We assume that the sampling period is provided by the system designer based on the structure of the program under inspection and properties of interest (e.g., determined by the automated methods in [14, 27, 64, 52]).

2. (Instrumentation optimality) How can we minimize the overhead of instrumentation at run time while enabling time-triggered self-monitoring that respects the correctness condition?

3. (Minimum deviation) How should the program be instrumented such that execution of sampling instructions are as close as possible to the given sampling period in all execution paths?

Our approach works as follows for sequential programs. We formally define the concept of time-triggered self-monitoring in terms of the correctness and instrumentation optimality constraints mentioned above. In order to ensure correctness, we construct the program's control-flow graph (CFG), where the weight of a vertex is its best-case execution time (BCET). Computing the sampling period of a CFG based on BCET of basic blocks is quite

realistic, as (1) all hardware vendors publish the BCET of their instruction set in terms of clock cycles, and (2) BCET is a conservative approximation and no execution occurs faster than that. Using vertex weights, one can design simple algorithms that identify vertices where self-sampling instructions should be added. These instructions simply read the state of the program (e.g., variable values, contents of stacks, register values, etc) and pass the state to a monitor function in the program for evaluating properties.

A naive approach to facilitate programs to *self-sample* itself is by instrumenting all the vertices in the control-flow graph of the program. Although this is an acceptable solution, it is desirable to have minimal impact of instrumentation on the code size and execution time of the program under scrutiny. To ensure optimal instrumentation, we require that the number of instrumented vertices in the control-flow graph is minimum. It is shown that the corresponding optimization decision procedure is NP-complete in the size of the program's control-flow graph. To remedy the exponential complexity, we propose two approaches. First, we propose a mapping from our optimization problem to the Boolean satisfiability problem. This mapping enables us to utilize powerful SAT-solvers to solve our optimization problem. Secondly, we propose a heuristic that finds nearly optimal solutions to the problem.

We emphasize that we do not address the third constraint introduced above (i.e., minimum deviation) in this work. Also, our current approach works only for sequential programs. This is because enabling self-monitoring with optimal instrumentation requires analysis of the causal order of occurrence of events in a concurrent program for identifying optimal instrumentation points. We consider this to be part of the future work related to self-monitoring. Our method is fully implemented in a tool chain. The tool takes a C program as input and computes the instrumentation locations. We have conducted a set of experiments to compare the behavior of self-monitoring programs with their counterparts monitored by an external process. The experimental results show that self-monitored programs perform significantly faster than externally monitored programs. This is simply due to the elimination of the cost of synchronization and context switching for programs monitored by an external process.

It can be noted that instrumenting a program to add self-checking instructions is a commonly applied exercise by system designers and developers. Examples include asser-

tion instructions, exception handling, and even simple conditional statements to check the state of the program. The technique proposed in this work ensures that such instructions are executed within a certain time period at run time and that the number of inserted instructions is minimum.

4.2 Time-triggered Self-monitoring with Minimum Instrumentation

4.2.1 Problem Description

Time-triggered runtime monitoring [27, 14] consist of a monitor and an application program under inspection. The monitor runs in parallel with the application program and interrupts the program execution at regular time intervals to observe the state of the program. This state could be formed by some variable values, stack values, register values, etc. The key advantage of this technique is obtaining bounded and predictable overhead incurred on the program execution. This overhead is inversely proportional to the sampling period at which the monitor samples the state of the program.

Although time-triggered monitors with interruptions have widely been used, they suffer from two main drawbacks:

- time-triggered interruptions introduce a large number of context switches to the system,
- if the program under inspection is blocked (e.g., waiting for I/O) the monitor keeps waking up to take samples from the program, and
- incorporating external monitors requires communication between at least two processes and synchronization data structures require the underlying operating system to provide kernel-level system call primitives and inter-process communication features.

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

Figure 4.1: A simple C program

These issues introduce additional but unnecessary overhead to the system. In addition, in the latter case, such primitives may possibly be unreliable in real-time settings. Moreover, some embedded environments such as TinyOS do not provide such primitives at all.

In order to eliminate the aforementioned overheads, we introduce the concept of *time-triggered self-monitoring*. Our idea is to remove the external time-triggered monitor and instrument the program under inspection by augmenting the program with instructions that *self-sample* the state of the program periodically for inspection without using an internal timer. Specifically, these instructions read the state of the program (e.g., a set of variables, registers, path markers, stack contents, etc) and call a function within the program for monitoring purposes. Moreover, the program execution time between each two successive samples must be at most the desired sampling period, given as an input parameter. For instance, for the program in Figure 4.1, where the sampling period is $SP = 2$, the goal is to augment the program with instructions that take a sample from the value of variable `x`, such that the execution time between each two successive samples is *at most* 2 time units.

We emphasize that, we assume that the sampling period be given as part of the inputs to an instrumentation algorithm for self-monitoring. The sampling period is provided by the system designer using automated techniques such markers, history variables, etc as described in related work [14, 27, 64, 52]. Thus, the process of obtaining a sampling period

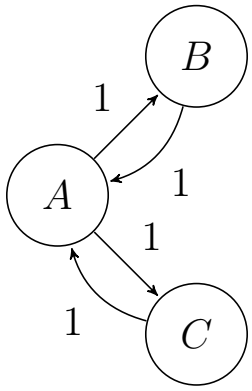

Figure 4.2: Control flow-graph (CFG) for the program in Figure 4.1

is irrelevant to the algorithms that generate instrumentation schemes for enabling self-monitoring for a program. In other words, such algorithms only take a control-flow graph and a desired sampling period as input and return a set of vertices of the control-flow graph that need to be instrumented.

A naive solution to instrument a program for self-monitoring is as follows. One can insert self-sampling instructions at every vertex of the CFG, which take samples within the sampling period. However, in order to minimize the impact of instrumentation, it is desirable to insert the minimum number of self-sampling instructions in the program. Next, we formalize an optimization problem that captures minimum instrumentation that enables self-monitoring in a program for a given sampling period.

4.2.2 Complexity Analysis

Let $G = \langle V, v^0, A, w \rangle$ be the control-flow graph of a program and SP be the sampling period at which the program has to be sampled. We define a *control-flow graph* (CFG) as follows:

Definition 1. *The control-flow graph of a program P is a weighted directed simple graph $CFG_P = \langle V, v^0, A, w \rangle$, where:*

- *V is a set of vertices, each representing a basic block of P. Each basic block consists of a sequence of instructions in P.*
- *v^0 is the initial vertex with in degree 0, which represents the initial basic block of P.*
- *A is a set of arcs of the form (u, v), where $u, v \in V$. An arc (u, v) exists in A, if and only if the execution of basic block u can immediately lead to the execution of basic block v.*
- *w is a function $w : A \to \mathbb{N}$, which defines a weight for each arc in A. The weight of an arc is the best-case execution time (BCET) of the source basic block.*

For example, Figure 4.1 shows a simple C program with three basic blocks labeled A, B, and C. Figure 4.2 shows the control-flow graph of the program.

Let $\Pi_{v,v'}$ denote the set of all simple paths between two vertices v and v' in V and Π be the set of all paths in G. The length of a path π (denoted $Length(\pi)$) is calculated as the sum of the weights of arcs present on π using the function w.

Now, let $\mathcal{V} : \Pi \to 2^V$ be the function that obtains the set of vertices on a path except the source and end vertices. Our goal is to find the minimum set of vertices $V' \subseteq V$, such that for any two vertices $v, v' \in V'$, the length of the longest path from v to v' that does not pass through a vertex $v'' \in V'$, where $v'' \neq v, v'$, is at most SP. The initial vertex v^0 is by default always present in V'. The set V' identifies the basic blocks that need to be instrumented to augment the program with self-sampling instructions. We now show that this minimization problem is NP-complete.

Instance. A control-flow graph $G = \langle V, v^0, A, w \rangle$, a sampling period SP, and a positive integer k, where $k \leq |V|$.

Self-monitoring instrumentation decision problem (SMI). Does there exist a set $V' \subseteq V$ of vertices such that:

- $|V'| \leq k$

- $v^0 \in V'$
- $\forall v, v' \in V' : \forall \pi \in \Pi_{v,v'} :$
$$\nexists v'' \in (\mathcal{V}(\pi) \cap V') :$$
$$Length(\pi) \leq SP.$$

Lemma 1. *SMI is in NP.*

Proof. We need to show that given a solution to the problem, one can verify its correctness in polynomial-time. Given an instance of SMI and a set V' of vertices as a certificate, we verify whether or not V' solves the decision problem as follows. The first two conditions of SMI can be verified trivially. For the third condition, for every vertex $v \in V'$, we construct an SP-depth first tree (SPDFT) as follows. An SP-depth first tree of a vertex v is a spanning tree rooted at v obtained by applying depth-first search exploration on G, such that the length of any path of the tree that starts from v and ends at a leaf is at most SP. Also, all forward and cross edges are preserved when a depth first search exploration is applied on G to obtain the SPDFT of a vertex v.

Now, given a SPDFT rooted at vertex $v \in V'$, we check if there exists a path from v to one of the leaves, such that this path does not include a vertex $v' \in V'$, where $v' \neq v$. If so, it means that after self-sampling in basic block v, there exists an execution path of the program where self-sampling does not occur within the given sampling period SP. Hence, the answer to the verification question is negative. Otherwise, starting from v, in all execution paths the program self-samples within SP time units.

We repeat this procedure for all vertices in V'. If all vertices pass this verification successfully, the answer to the verification problem is affirmative. The complexity of the algorithms is $O(V^2)$ and, hence, SMI is a member of the class NP. □

Lemma 2. *SMI is NP-hard.*

Proof. Now, we show that SMI is NP-hard. To this end, we reduce the *Minimum Vertex Cover Problem* (VC) to SMI. The minimum vertex cover problem is as follows [40]. Given a (directed or undirected) graph $G = \langle V, E \rangle$ and a positive integer K, the problem is to find a set $U \subseteq V$, such that $|U| \leq K$ and each edge in E is incident to at least one vertex

in U.

Mapping. Let directed graph $G_{vc} = \langle V_{vc}, E_{vc} \rangle$ and a positive integer K be an instance of VC. We assume that the graph has a vertex v^0_{vc} with in degree zero. This assumption does not change the complexity of VC. We can obtain an instance of SMI as follows:

- Graph $G_{smi} = \langle V_{smi}, v^0_{smi}, A_{smi}, w \rangle$, where
  - $V_{smi} = V_{vc}$
  - $v^0_{smi} = v^0_{vc}$
  - $A_{smi} = A_{vc}$
  - $w(a) = 1$ for all $a \in A_{smi}$
- Sampling period $SP = 2$, and
- $k = K$.

Reduction. We now prove that a solution to VC exists if and only if a solution to the obtained instance of SMI as prescribed above exists:

- ($\Rightarrow$) Let $U \subseteq V$ be a solution to VC for graph $G_{vc} = \langle V_{vc}, E_{vc} \rangle$, such that $|U| \leq K$. Let V' identical to U be the solution to SMI for graph $G_{smi} = \langle V_{smi}, v^0_{smi}, A_{smi}, w \rangle$ and sampling period $SP = 2$. Thus, self-sampling instructions are added to the end of basic blocks in U. We now show that this solution is valid for SMI. First, we have $|U| \leq k$, as $k = K$ and $|U| \leq K$. Secondly, for any edge $e = (u, v) \in E$, either one of the incident vertices u or v must belong to U or both u and v belong to U. In the former case, the number of edges on a path from either u or v to another vertex $v' \in V'$ will be 2. In the latter case, the number of edges on a path from either u or v to another vertex in V' will be 1 since that would either be v or u, respectively. Likewise, the number of edges between any two vertices in V'

is at most 2 by applying the same analogy to all edges in E. Hence, the set U is an answer to the instance of SMI.

- ($\Leftarrow$) Let V' be a solution to the instance of SMI (i.e., the graph $G_{smi} = \langle V_{smi}, v^0_{smi}, A_{smi}, w\rangle$ and sampling period $SP = 2$). We show that U identical to V' is a solution to VC for the graph $G_{vc} = \langle V_{vc}, E_{vc}\rangle$. First, notice that we have $|V'| \leq K$, as $K = k$ and $|V'| \leq k$. Secondly, for any two vertices $v, v' \in V'$, the number of arcs on a path from v to v' that does not include a third vertex $v'' \in V'$ must be is at most 2. Otherwise, U is a not a valid solution to SMI. For the case of length 2, one of the edges will be incident to v and the other will be incident to v'. Since $v^0_{smi} \in V'$, it can be seen that all the arcs in E_{vc} will be incident to at least one of the vertices in U using the same analogy applied to all vertices in U taken as pairs. Hence, U is a vertex cover for graph G_{vc}.

□

Theorem 3. *SMI is NP-complete.*

Proof. The proof of the theorem trivially follows from Lemmas 1 and 2. □

4.3 Coping with the Exponential Complexity

As we showed in Section 4.2, the problem of identifying minimum set of instrumentation for a program to enable self-monitoring is NP-complete. To remedy the exponential complexity, in this section, we propose a SAT-based solution that finds an optimal solution to our problem and a greedy algorithm. We present these solutions in Subsections 4.3.1 and 4.3.2, respectively.

4.3.1 A SAT-based Solution

In this subsection, we propose a transformation from the optimization problem presented in Section 4.2 (SMI) into the *Boolean satisfiability problem* (SAT); i.e., the problem of

assigning truth values to variables of a given Boolean formula to make the formula evaluate to logical *true*.

Let $G = \langle V, v^0, A, w \rangle$ be a control-flow graph and SP be a desired sampling period. We construct a SAT formula as follows. The set of Boolean variables in our SAT formula is:

$$X = \{x_v \mid v \in V\}.$$

Our intention is that, if $x_v = \mathit{true}$, then basic block v in G will be instrumented with self-sampling instructions. Otherwise, basic block v remains unchanged.

We now identify the constraints of the SAT formula. Let v be a vertex in V with out degree greater than 0. We construct the SP-depth first tree as described in the proof of Lemma 1 (i.e., a spanning tree rooted at v obtained by applying depth first search exploration on graph G while preserving all forward and cross edges, such that the length of any path from v to its leaves is at most SP). Let $Ch : V \to 2^V$ denote a function that computes the set of child vertices of a vertex. The Boolean formula representing vertex v is of the form:

$$\mathcal{F}_v = (x_v \Rightarrow \bigwedge_{u \in Ch(v)} \mathcal{G}_u^{SP-1}) \tag{4.1}$$

Intuitively, $\mathcal{F}_v$ captures the constraint that if a basic block v is instrumented, then in all execution branches, a basic block within $SP - 1$ steps needs to be instrumented, as well. The latter proposition is specified by the conjunction of $\mathcal{G}_u^{SP-1}$ formulas. For example, consider the 3-depth tree rooted at vertex v_1 in Figure 4.3. For this tree, by applying Constraint 4.1, we have $\mathcal{F}_{v_1} = (x_{v_1} \Rightarrow \mathcal{G}_{v_2}^2 \wedge \mathcal{G}_{v_3}^2)$.

Formula $\mathcal{G}$ is recursively defined as follows:

$$\mathcal{G}_u^i = (x_u \vee \bigwedge_{w \in Ch(u)} \mathcal{G}_w^{i-1}) \tag{4.2}$$

i.e., either basic block u is instrumented or in all execution branches starting from u, a basic block within $i - 1$ steps is instrumented. The termination condition of this formula

is $\mathcal{G}_w^0 = x_w$. For example, by applying Constraint 4.2, we have $\mathcal{G}_{v_2}^2 = x_{v_2} \vee [(x_{v_4} \vee (x_{v_7} \wedge x_{v_8})) \wedge (x_{v_5} \vee x_{v_9})]$ and $\mathcal{G}_{v_3}^2 = x_{v_3} \vee x_{v_6}$.

We add identical constraints for each vertex in the control-flow graph. Thus, our complete SAT formula is the following:

$$\mathcal{F} = \bigwedge_{v \in V} \mathcal{F}_v \tag{4.3}$$

Finally, our objective to minimize the number of instrumentations for self-sampling is the following (assuming that logical *true* has an integer value 1 and logical *false* has an integer value 0):

$$\text{minimize} \sum_{v \in V} x_v \tag{4.4}$$

i.e., by finding the minimum number of Boolean variables (respectively, vertices) whose truthfulness (respectively, instrumentation) makes the SAT formula *true* (respectively, enables self-monitoring in the control-flow graph). We note that although Constraint 4.4 is not a normal SAT constraint, one can implement such a constraint in modern solvers for satisfiability modulo theory (SMT) efficiently using a simple binary search algorithm.

Special Case. A special case occurs when the required sampling period is greater than the length of the longest execution path of the program. In such a case, both $SAT-based$ and *Greedy* algorithms instrument only the root and the leaves of the program's CFG. This results in the possibility that an execution path with length greater than the required sampling period exists if the CFG has any cycles or loops. The main reason for such a case to occur is due to the fact that the loop bound is not taken into consideration in the building of the CFG of the program. We solve this special case by instrumenting at least one vertex that lies on the execution path that constitutes the loop. We perform the same procedure for all the loops that exist in the program.

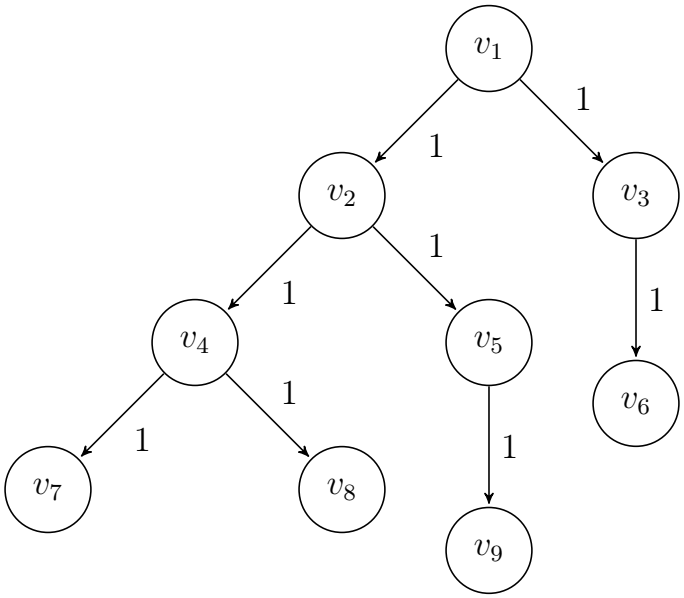

Figure 4.3: Example of a 3-depth first tree.

4.3.2 A Greedy Algorithm

In addition to our SAT-based solution, we also propose a simple greedy algorithm for instrumenting a given control-flow graph $G = \langle V, v^0, A, w \rangle$ with sampling period SP. The algorithm works as follows:

1. Initially, we let $U = \{v^0\}$ and $W = \{v^0\}$.
2. We construct SP-depth first trees (SPDFT) rooted at vertices in U and set $U = \emptyset$.
3. Let T be the set of all vertices and R be the leaves of the tree constructed in Step 2.
4. We instrument the vertices in R and we let $U = U \cup R$ and $W = W \cup T$.
5. We repeat Steps 2 and 3 until $W = V$.

The only case where this algorithm may not instrument correctly is when a control-flow graph has cycles. To deal with cycles, one can find back-arcs and instrument vertices on cycles. The special case discussed in Subsection 4.3.1 also applies for the heuristic presented in this subsection.

4.4 Framework

In order to compare the performance of time-triggered self-monitoring with the widely used external monitored time-triggered monitoring scheme, we have implemented the self-monitoring scheme as a toolchain. The toolchain consists of 3 modules, CFG_Generator, Solution_Generator and Instrumentation_Generator. The inputs to the toolchain include the C program and the sampling period. The output is an instrumented program. The CFG_Generator takes the C program and generates the control-flow graph of the program. The control-flow graph and the sampling period are passed as inputs to the next module Solution_Generator. Depending on the optimality of the solution required, this module transforms the control-flow graph either to a SAT expression using the method described in Subsection 4.3.1 or uses the heuristic described in Subsection 4.3.2 to obtain an optimal or suboptimal solution, respectively. We utilize the SMT-solver Yices [3] to solve the SAT expression. In either case, we obtain the set of vertices of the control-flow graph that need to be instrumented with self-sampling instructions. Finally, we again pass the C program and the set of vertices of the CFG that needs to be instrumented with self-sampling instructions to the Instrumentation_Generator module. This module instruments the required vertices with self-sampling instructions and outputs the final instrumented code. The self-sampling instructions could be any verification task that the designer requires. For the purpose of experimentation, we assign the self-sampling instrumentation instructions to include storing the value of the most frequently used variable to an array to emulate a property verification task. The 3 modules described above are implemented in Java. Bash scripting is used to connect the inputs and outputs of the modules.

4.5 Experimental Methods and Setting

In this section, we present the experimental settings used for experimentation to evaluate the effectiveness and efficiency of time-triggered self-monitoring. In Section 4.6, we present the experimental results and analyze the results.

In order to compare our self-monitoring technique with external time-triggered monitoring, we deploy a time-triggered external monitor as follows. We use shared memory for the program and the monitor to exchange data between them. This data is basically the value of variables that change the truthfulness of a property in the program under inspection. The program is instrumented such that it writes the value of the most frequently used variable to the shared memory whenever the variable gets modified. The monitor periodically reads the shared memory and stores the values in an array, performing the same monitoring task as for the self-monitored program.

Our case studies are drawn from the Mälardalen [33] benchmark suite. All experiments in this section are conducted on a Dual-core ARM Cortex-A9 MPCore with Symmetric Multiprocessing (SMP) at 1 GHz each and 1 GB low-power DDR2 RAM under Ubuntu Linux with the default scheduling policy. Each case study is run in a loop of 1000 iterations for measurements and we ran each experiment 50 times to ensure that the collected data is sound and exhibits reliable confidence intervals.

The parameters used in performing the experimentation were input control-flow graph, required sampling period and the scheme used for solving the self-monitoring optimization problem. The various metrics considered for the experimentation are as follows:

4.5.1 Execution time of the instrumented program

In order to compare the self-monitoring scheme with the external time-triggered monitoring scheme, we compare the execution time of the program instrumented with self-sampling instructions against the execution time of the program monitored using an external monitor. Also, we compare the execution time of the monitored version of the program against the unmonitored version of the program in order to evaluate the overhead of the monitoring.

4.5.2 Context Switches

Since external time-triggered monitoring scheme requires an external monitor to run in parallel with the program, it is interesting the observe the number of context switches that occur as a result of monitoring a program using an external monitor. The context switches of the external time-triggered monitoring scheme are compared against the number of context switches encountered by the time-triggered self-monitoring scheme.

4.5.3 Code Size

We use this metric to compare the impact of instrumentation on the time-triggered self-monitoring scheme. This metric compares the code size of the program instrumented with self-sampling instructions against the code size of the time-triggered monitoring scheme using an external monitor.

4.5.4 Number of vertices picked for Self-Monitoring

We use this metric to compare the effectiveness of the heuristic with the optimal solution produced by the SAT-based solution. This metric compares the number of vertices instrumented with self-sampling instructions using the SAT-based solution described in Subsection 4.3.1 versus the heuristic described in Subsection 4.3.2.

4.6 Experimental Results and Analysis

In this section, we present the experimental results and subsequent analysis of the results applying the experimental methods described in Section 4.5.

4.6.1 Performance of the SAT-based and Greedy Techniques

First, we analyze the performance of our SAT-based (i.e., optimal) and greedy solution in terms of their capability in handling input control-flow graphs. Table 4.1 shows the chosen

Case Study	Sampling Period (ns)	Size of CFG (Vertices)	*SAT-based Solution*		*Greedy Algorithm*	
			No. of Vertices	**% of Vertices**	**No. of Vertices**	**% of Vertices**
CNT	1000	28	5	17.85	7	25
InsertSort	1000	11	4	36.36	4	36.36
MATMULT	1000	29	7	24.13	9	31.03
FIBCALL	25	9	7	77.77	7	77.77
QURT	100	30	8	26.67	20	66.67
ADPCM	1000	158	29	18.35	70	44.3

Table 4.1: Comparing the number and percentage of vertices chosen for instrumentation for SAT-based and greedy techniques.

sampling periods and the size of the input control-flow graph in terms of the number of vertices. We note that the sampling periods are chosen based on two criteria, namely, the internal structure of the case studies and the requirement that the programs should be sampled at least once. It can be observed that on average, the size of the solution set obtained by using the greedy algorithm is nearly double the size of the solution set obtained by using the SAT-based method.

The time spent in obtaining the solution sets using the SAT-based approach and the greedy algorithm (not shown in Table 4.1) are comparable in most cases. The only exception was ADPCM with sampling period 1000ns. We ran the SAT-based method on ADPCM with sampling period 1000ns and it took nearly 4 hours to obtain a solution. This special case highlights the fact that in some cases, obtaining a near-optimal solution is more feasible than the optimal solution. For the cases where the data (number of vertices instrumented) for different sampling periods are equal, the corresponding row is omitted in Table 4.1. Obtaining equal data for different sampling periods is due to the fact that the sampling period is greater than the execution time of the program. In this case, only

the root, leaves, and at least one vertex for each loop that lies on the execution path that constitutes the loop, of the program's CFG would be instrumented. Thus, increasing the sampling period results in obtaining the same size of vertices for instrumentation.

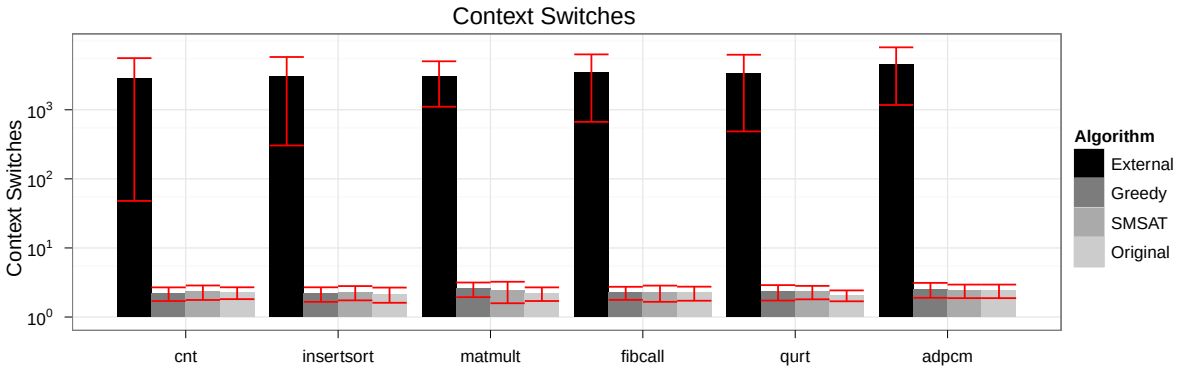

Figure 4.4: Results on reduction of context switching (in log scale).

4.6.2 Analysis of Self-monitoring Overhead

We now analyze the impact of instrumentation on performance in terms of the number of context switches and execution time of programs under inspection. Figure 4.4 compares the number of context switches in execution of our case studies using the external monitor against the self-monitored programs instrumented by the SAT-based approach and the greedy algorithm. Note that the bar chart in Figure 4.4 is in *logarithmic scale*. As can be seen, the number of context switches incurred using external monitoring is higher than self-monitoring in orders of magnitude. This result simply shows that self-monitoring is highly preferred in a real-time setting, where non-determinism is not desirable. Also, the number of context switches incurred in the original unmonitored program and self-monitored programs instrumented by SAT-based and greedy approaches are very close. In other words, self-monitoring can significantly assist in preserving the predictability of the program under inspection. Note that in Figure 4.4 the error bars represent the standard error of mean (SEM).

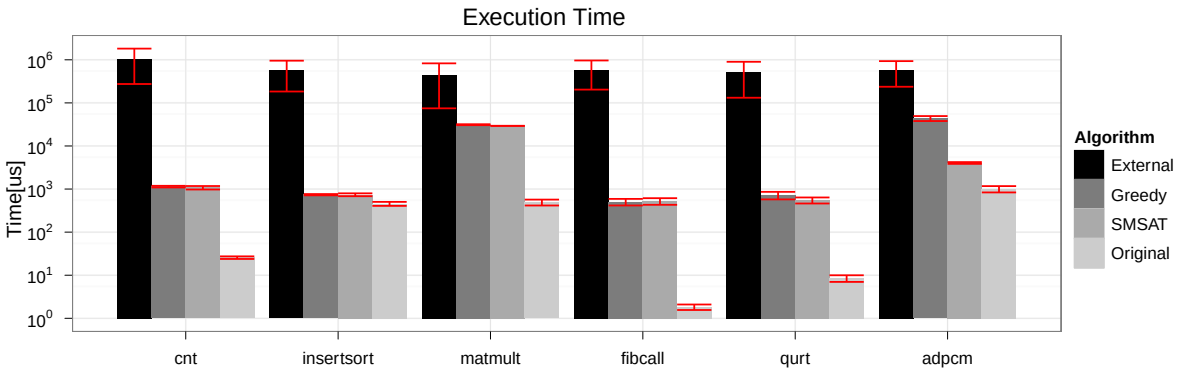

Figure 4.5: Results on reduction of execution time (in log scale).

Figure 4.5 compares the total execution time of our case studies (in microseconds) for the original unmonitored program and three different monitoring techniques: (1) using an external monitor, (2) self-monitoring program instrumented by the SAT-based approach (SMSAT), and (3) self-monitoring program instrumented by our greedy algorithm. Note that the bar chart in Figure 4.5 is also in *logarithmic scale*. As can be seen in Figure 4.5, the total execution time of programs monitored by an external process is significantly higher than the execution of their self-monitored counterparts. More specifically, self-monitored programs instrumented using the SAT-based method are on average 2 times faster than externally monitored programs. And, self-monitored programs instrumented using the greedy approach are on average 1.6 times faster than externally monitored versions. It can also be observed that self-monitored programs instrumented using the SAT-based method run on an average 2 times slower than their unmonitored counterparts. Also, note that in Figure 4.5 the error bars represent the standard error of mean (SEM).

4.7 Inference & Discussion

In this chapter, we proposed a new technique for runtime monitoring called *time-triggered self-monitoring*. This technique aims at reducing the overheads incurred at time-triggered monitoring using an external monitor process. In time-triggered external monitoring, the

monitor runs in parallel with the program and samples the program state periodically to evaluate a set of properties. Incorporating such an external process increases the overhead due to inter-process communication and context switching costs. Moreover, self-monitoring remedies the tight dependency between the program and monitor at run time, making it more resilient to faults and unreliability of kernel-level synchronization system calls in real-time settings. Furthermore, self-monitoring can be deployed in embedded environments, where multi-tasking features are not necessarily assumed (e.g., in TinyOS). Moreover, since time-triggered monitoring provides us with bounded and predictable overhead, it is suitable for time-sensitive platforms, where violation of timing constraints may lead to catastrophic consequences.

Our self-monitoring technique instruments a program under scrutiny with instructions in such a way that the program *self-samples* itself periodically. Moreover, self-sampling instrumentation ensures that (1) the time interval between two successive self-sampling points does not exceed the desired sampling period, and (2) minimum number of self-sampling points is introduced to the program. Our experiments show that self-monitored programs perform significantly faster than their counterparts monitored by an external monitor. Moreover, the binary code size and the number of context switches occurred in self-monitored programs are substantially less than externally monitored programs.

Some of the interesting next steps include self-monitoring in the context of concurrent programs. Our current method cannot handle concurrent programs, as identifying self-sampling points in the program require causal relation analysis of the program's threads and processes. Another direction is to strengthen our optimization problem such that self-sampling instructions use their time budget optimally (see the minimum deviation criterion in Section 4.1); i.e., the instructions execute as close as possible to the intended sampling points. Developing more sophisticated heuristics to tackle the exponential complexity of our optimization problem is also an interesting research problem.

Chapter 5

Conclusion

In software development, debugging is an important phase where developers detect and remove software defects from the program. It is a popular practice to use software instrumentation as the debugging technique in order to perform tracing and monitoring. Recently time-triggered based debugging has been suggested and is useful in performing tracing and runtime verification. Most of the current monitoring approaches used for runtime verification are *event-triggered*. The disadvantage of event-triggered monitoring is the constant invocation of the monitor leading to unpredictable overhead and potentially bursts of monitoring intervention at runtime.

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 the work proposed in Chapter 3, 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 and monotonicity.

Although, time-triggered external monitoring results in bounded overhead incurred on the program under inspection, it suffers several drawbacks. In time-triggered external

monitoring, the monitor runs in parallel with the program and samples the program state periodically to evaluate a set of properties. Incorporating such an external process increases the overhead due to inter-process communication and context switching costs. Moreover, there is a tight dependency between the program and monitor at run time, making it more resilient to faults and unreliability of kernel-level synchronization system calls in real-time settings. Furthermore, time-triggered external monitoring scheme requires the embedded environments to provide multi-tasking features.

To address the above aforementioned problems with time-triggered external monitoring, we proposed a new technique called *time-triggered self-monitoring* in Chapter 4. This technique aims at reducing the overheads incurred at time-triggered monitoring using an external monitor. Our self-monitoring technique instruments a program under scrutiny with instructions in such a way that the program *self-samples* itself periodically without maintaining an internal timer. Moreover, self-sampling instrumentation ensures that (1) the time interval between two successive self-sampling points does not exceed the desired sampling period, and (2) minimum number of self-sampling points is introduced to the program. We showed that solving this minimization problem is NP-complete in the size of the program's control-flow graph. We subsequently proposed a SAT-based method that finds optimal solutions and a polynomial-time greedy algorithm that finds near-optimal solutions. Our experiments show that self-monitored programs perform significantly better than their counterparts monitored by an external monitor. Moreover, the binary code size and the number of context switches occurred in self-monitored programs are substantially less than externally monitored programs.

In general, developers can use any of the time-triggered monitoring schemes discussed in the work presented in this thesis, to debug and monitor their embedded systems for safety-critical as well as other properties of interest.

5.1 Future Work

Future work with respect to time-triggered execution monitoring would include investigating a new technique where different instrumentation schemes are used based on the best

applicability given the current instrumentation problem to be solved. It would be interesting to observe the impact of such an instrumentation scheme on the gain of sampling period and monotonicity.

For future work with respect to time-triggered self-monitoring, we are considering several research directions. An important direction is self-monitoring in the context of concurrent programs. Our current method cannot handle concurrent programs, as identifying self-sampling points in the program require causal relation analysis of the program's threads and processes. Another direction is to strengthen our optimization problem such that self-sampling instructions use their time budget optimally (see the minimum deviation criterion in Section 4.1); i.e., the instructions execute as close as possible to the intended sampling points. Developing more sophisticated heuristics to tackle the exponential complexity of our optimization problem is also an interesting research problem.

‘

References

[1] Dynamorio: Dynamic instrumentation tool platform. http://www.dynamorio.org/.

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

[3] Yices: An SMT Solver. http://yices.csl.sri.com.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] A. Bauer, M. Leucker, and C. Schallhart. Runtime Verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology (TOSEM), 2009. in press.

[12] A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL Semantics for Runtime Verification. Journal of Logic and Computation, 20(3):651–674, 2010.

[13] E. Bertino, E. Ferrari, and G. Guerrini. An approach to model and query event-based temporal data. In Temporal Representation and Reasoning, 1998. Proceedings. Fifth International Workshop on, pages 122 –131, may 1998.

[14] B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Sampling-based runtime verification. In Formal Methods (FM), pages 88–102, 2011.

[15] B. Bonakdarpour, J. J. Thomas, and S. Fischmeister. Time-triggered program self-monitoring. In IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pages 260–269, 2012.

[16] Jim Bowring, Alessandro Orso, and Mary Jean Harrold. Monitoring deployed software using software tomography. In Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, PASTE '02, pages 2–9, New York, NY, USA, 2002. ACM.

[17] 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.

[18] 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.

[19] E. Y. Chang, Z. Manna, and A. Pnueli. Characterization of Temporal Property Classes. In Automata, Languages and Programming (ICALP), pages 474–486, 1992.

[20] 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.

[21] 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.

[22] S. Colin and L. Mariani. Run-Time Verification, chapter 18. Springer-Verlag LNCS 3472, 2005.

[23] M. d'Amorim and G. Rosu. Efficient Monitoring of omega-Languages. In Computer Aided Verification (CAV), pages 364–378, 2005.

[24] 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.

[25] R.P. Dick, D.L. Rhodes, and W. Wolf. Tgff: Task Graphs for Free. In Hardware/Software Codesign (CODES/CASHE), pages 97–101, 1998.

[26] Y. Falcone, J.-C. Fernandez, and L. Mounier. Runtime Verification of Safety-Progress Properties. In Runtime Verification (RV), pages 40–59, 2009.

[27] 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.

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

[29] 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.

[30] 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.

[31] D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In Automated Software Engineering (ASE), pages 412–416, 2001.

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

[33] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. In WCET2010, pages 137–147, 2010.

[34] K. Havelund and A. Goldberg. Verify your Runs. Verified Software: Theories, Tools, Experiments (VSTTE), pages 374–383, 2008.

[35] K. Havelund and G. Rosu. Monitoring Programs Using Rewriting. In Automated Software Engineering (ASE), pages 135–143, 2001.

[36] K. Havelund and G. Rosu. Synthesizing Monitors for Safety Properties. In Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pages 342–356, 2002.

[37] K. Havelund and G. Rosu. Efficient Monitoring of Safety Properties. Software Tools and Technology Transfer (STTT), 6(2):158–173, 2004.

[38] K. Havelund and Gr. Rosu. Monitoring Java Programs with Java PathExplorer. Electronic Notes in Theoretical. Computer Science, 55(2), 2001.

[39] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok. Software monitoring with controllable overhead. Software tools for technology transfer (STTT), 14(3):327–347, 2012.

[40] R. M. Karp. Reducibility Among Combinatorial Problems. In Symposium on Complexity of Computer Computations, pages 85–103, 1972.

[41] M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky. Monitoring, Checking, and Steering of Real-Time Systems. Electronic. Notes in Theoretical Computer Science, 70(4), 2002.

[42] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A Run-Time Assurance Approach for Java Programs. Formal Methods in System Design (FMSD), 24(2):129–155, 2004.

[43] Amir Kishon, Paul Hudak, and Charles Consel. Monitoring semantics: a formal framework for specifying, implementing, and reasoning about execution monitors. In Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, PLDI '91, pages 338–352, New York, NY, USA, 1991. ACM.

[44] 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.

[45] O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. In Computer Aided Verification (CAV), pages 172–183, 1999.

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

[47] 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.

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

[49] 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.

[50] Z. Manna and A. Pnueli. A Hierarchy of Temporal Properties. In Principles of Distributed Computing (PODC), pages 377–410, 1990.

[51] 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.

[52] S. Navabpour, C. W. Wu, B. Bonakdarpour, and S. Fischmeister. Efficient techniques for near-optimal instrumentation in time-triggered runtime verification. In International Conference on Runtime Verification (RV), pages 208–222, 2011.

[53] 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.

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

[55] L. Pike, A. Goodloe, R. Morisset, and S. Niller. Copilot: A Hard Real-Time Runtime Monitor. In Runtime Verification (RV), 2010. 345-359.

[56] L. Pike, S. Niller, and N. Wegmann. Runtime verification for ultra-critical systems. In Runtime Verification (RV), pages 310–324, 2011.

[57] A. Pnueli and A. Zaks. PSL Model Checking and Run-Time Verification via Testers. In Symposium on Formal Methods (FM), pages 573–586, 2006.

[58] G. Rosu, F. Chen, and T. Ball. Synthesizing Monitors for Safety Properties: This Time with Calls and Returns. In Runtime Verification (RV), pages 51–68, 2008.

[59] Mohlalefi Sefika, Aamod Sane, and Roy H. Campbell. Monitoring compliance of a software system with its high-level design models. In Proceedings of the 18th international conference on Software engineering, ICSE '96, pages 387–396, Washington, DC, USA, 1996. IEEE Computer Society.

[60] J. Sosnowski and M. Poleszak. On-line monitoring of computer systems. In Electronic Design, Test and Applications, 2006. DELTA 2006. Third IEEE International Workshop on, page 5 pp., jan. 2006.

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

[62] S. Stoller, E. Bartocci, J Seyster, R. Grosu, K. Havelund, S. Smolka, and E. Zadok. Runtime verification with state estimation. In Runtime Verification (RV), pages 193–207, 2011.

[63] V. Stolz and E. Bodden. Temporal Assertions using Aspectj. Electronic Notes in Theoretical Computer Science, 144(4), 2006.

[64] J. J. Thomas, S. Fischmeister, and D. Kumar. Lowering overhead in sampling-based execution monitoring and tracing. In Languages, compilers, and tools for embedded systems (LCTES), pages 101–110, 2011.

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

[66] 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.

[67] 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.

[68] J.C. Yan. Performance tuning with aims/spl minus/an automated instrumentation and monitoring system for multicomputers. In System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on, volume 2, pages 625 –633, jan. 1994.

[69] 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.

[70] W. Zhou, O. Sokolsky, B. T. Loo, and I. Lee. *MaC*: Distributed Monitoring and Checking. In Runtime Verification (RV), pages 184–201, 2009.

APPENDICES

Appendix A

Copyright Notices

The material contained in this work are subject to the following copyright notices and agreements.

A.1 ACM

J. J. Thomas, S. Fischmeister, and D. Kumar. "Lowering overhead in sampling-based execution monitoring and tracing". In Languages, compilers, and tools for embedded systems (LCTES), pages 101-110, 2011, ©2011 Association for Computing Machinery, Inc. Reprinted by permission. http://doi.acm.org/10.1145/1967677.1967692

"Permission to make digital or hard copies of part or all 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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee."

Note: This work is used from the above mentioned citation is based on the rights retained to the authors under the Copyright Transfer section of the ACM Conference Copyright form and Audio/Video Release form, specifically clause 4 stated as follows *"The right to*

post author-prepared versions of the Work covered by the ACM copyright in a personal collection on their own home page, on a publicly accessible server of their employer and in a repository legally mandated by the agency funding the research on which the Work is based. Such posting is limited to noncommercial access and personal use by others, and must include the following notice both embedded within the full text file and in the accompanying citation display as well".

A.2 IEEE

Citation: B.Bonakdarpour, J.J. Thomas, and S. Fischmeister. Time-triggered Program Self-monitoring. In IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pages 260-269, 2012 [15]

DOI: 10.1109/RTCSA.2012.16

Link on IEEE Xplore:

- http://dx.doi.org/10.1109/RTCSA.2012.16
- http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6300158&isnumber=6300034

right and Consent Form, specifically clause 6 stated as follows *"Personal Servers. Authors and/or employers shall have the right to post the accepted version of IEEE-copyrighted articles on their own personal servers or the servers of their institutions or employers without permission from IEEE, provided that the posted version includes a prominently displayed IEEE copyright notice and, when published, a full citation to the original IEEE publication, including a link to the article abstract in IEEE Xplore. Authors shall not post the final, published versions of their papers."*.