Methods for Reducing Monitoring Overhead in Runtime Verification

by

Chun Wah Wallace Wu

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, 2013

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.

Abstract

Runtime verification is a lightweight technique that serves to complement existing approaches, such as formal methods and testing, to ensure system correctness. In runtime verification, monitors are synthesized to check a system at run time against a set of properties the system is expected to satisfy. Runtime verification may be used to determine software faults before and after system deployment. The monitor(s) can be synthesized to notify, steer and/or perform system recovery from detected software faults at run time.

The research and proposed methods presented in this thesis aim to reduce the monitoring overhead of runtime verification in terms of memory and execution time by leveraging *time-triggered* techniques for monitoring system events. Traditionally, runtime verification frameworks employ *event-triggered* monitors, where the invocation of the monitor occurs after every system event. Because systems events can be sporadic or bursty in nature, event-triggered monitoring behaviour is difficult to predict. Time-triggered monitors, on the other hand, periodically preempt and process system events, making monitoring behaviour predictable. However, software system state reconstruction is not guaranteed (i.e., missed state changes/events between samples).

The first part of this thesis analyzes three heuristics that efficiently solve the NP-complete problem of minimizing the amount of memory required to store system state changes to guarantee accurate state reconstruction. The experimental results demonstrate that adopting near-optimal algorithms do not greatly change the memory consumption and execution time of monitored programs; hence, NP-completeness is likely not an obstacle for time-triggered runtime verification. The second part of this thesis introduces a novel runtime verification technique called *hybrid runtime verification*. Hybrid runtime verification enables the monitor to toggle between event- and time-triggered modes of operation. The aim of this approach is to reduce the overall runtime monitoring overhead with respect to execution time. Minimizing the execution time overhead by employing hybrid runtime verification is not in NP. An integer linear programming heuristic is formulated to determine near-optimal hybrid monitoring schemes. Experimental results show that the heuristic typically selects monitoring schemes that are equal to or better than naïvely selecting exclusively one operation mode for monitoring.

Acknowledgements

I would like extend my greatest appreciation and gratitude to my family. Thank you for continually supporting my dreams and aspirations, for your words of advice and wisdom, and for your love. I am truly blessed by God to have such a great and loving family.

Words cannot describe how grateful I am for the love of my life, Noreen Wong, for being in my life. Thank you for always being there for me throughout all of the joys and challenges that I experienced throughout my graduate studies. I cannot thank you enough for your presence, comforting words, and love.

My graduate studies would not have been possible without the great mentorship, guidance, and support of Dr. Sebastian Fischmeister, my supervisor, and Dr. Borzoo Bonakdarpour. I sincerely thank you both for the opportunity and privilege of learning and conducting research with both of you. I would also like to extend my thanks to Dr. Derek Rayside for providing me an exciting opportunity to collaborate and develop new lab content for the undergraduate course in E&CE 351 (Compilers) and for his mentorship.

I would like to thank all of the members of the Real-Time Embedded Software Group for making the past two years on campus memorable. It was a pleasure to work on research projects with Shay Berkovich, Deepak Kumar, and Samaneh Navabpour. Hany Kashif and Johnson Thomas, thanks for being such great officemates.

I am also truly grateful for all of my friends' support. Thank you Amanda Cheung, Kathryn Cheung, Irina Choi, Jasmine Choi, Nathan Chung, Esther Lee, Heidi Wu, and Joanna Wu, for many years of incredible and supportive friendship. Yuki Cheung, Kenneth Lam, Henry Pang, Alice Tsang, Ronald Wan, Abraham Wong, Lily Wong, Osman Wong, and Jeffrey Woo, thank you for bringing lots of laughter and priceless memories throughout my undergraduate and graduate studies. Thank you Pastor Timothy Wai for your continual prayer, support, and mentorship, in my spiritual walk with God. Thank you all for everything.

My research and graduate studies were funded by scholarships from Natural Sciences and Engineering Research Council of Canada (NSERC) and Ontario Student Assistance Program (OSAP). I greatly appreciate the financial support NSERC and OSAP provided me over the past two years.

With kind permission of Springer Science and Business Media, Chapters 3 and 4 of this thesis were adopted from a publication that I co-authored: "*Efficient Techniques for Near-Optimal Instrumentation in Time-Triggered Runtime Verification*," in Runtime Verification, ser. Lecture Notes in Computer Science, vol. 7186, pp. 208 – 222.

Dedication

This thesis is dedicated to my family: Matthew, Wendy, and Joshua Wu, and to the love of my life, Noreen Wong, for their endless love, care, support, and encouragement throughout my studies.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software is ubiquitous in many applications and the versatility of software has promoted a rapid growth of complex software systems. Verifying software systems' correctness poses a significant and important challenge to address. In a relatively recent National Institute of Standards and Technology (NIST) report, it is estimated that $59.6 billion are lost every year from software errors [1]. In a more recent article, Charette highlights some incidents that attribute huge monetary losses due to software failures [2]. For example, Inland Revenue (from the United Kingdom) experienced a $3.45 billion tax-credit overpayment that were attributed to software errors in 2004-2005. Software failures are not limited to monetary consequences. There are millions of lines of code in modern aircrafts and vehicles. The F-35 Joint Strike Fighter and Boeing's 787 Dreamliner require the use of 5.7 and 6.5 million lines of code, respectively; premium-class automobiles also require software systems to that scale and the complexity of such systems are growing extremely fast [3]. Ensuring software correctness is of utmost importance in these applications because software failures may endanger the safety of the people operating and using them.

In computing systems, *correctness* refers to the assertion that the system satisfies its specification; verification is a way to check for such an assertion [4]. There are traditionally three main verification techniques that are applied in the software domain: theorem proving, model checking, and software testing. All three types of techniques have strengths and weaknesses when they are applied to software systems design and development.

Theorem proving generally involves finding a proof from a set of axioms after express-

ing the system and its properties in some form of mathematical logic. State-of-the-art theorem provers can provide some automation, but typically requires a significant amount of manual effort/interaction to prove the correctness of the system model with respect to its properties [5, 6]. Theorem proving is able to cope with an infinite search space.

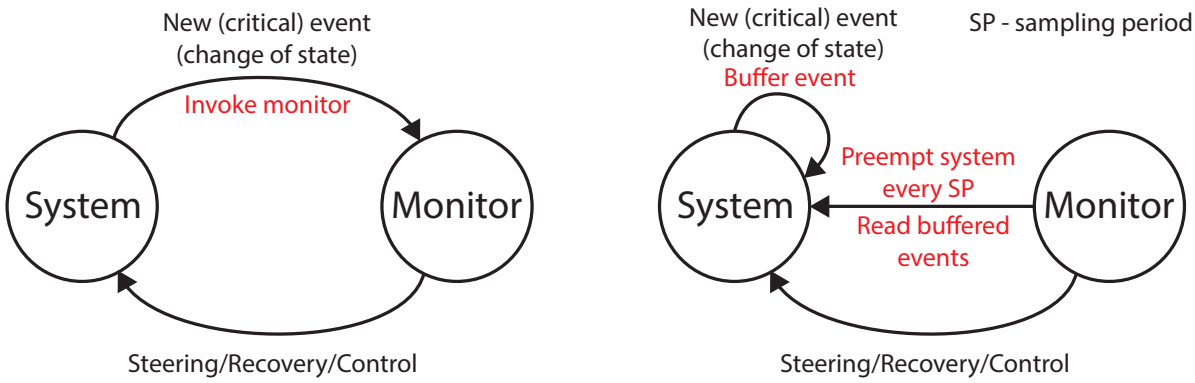
Model checking [7] on the other hand automates the process of proving the satisfiability of system properties by conducting an exhaustive search on an abstract model that is representative of the system. This search is computationally expensive; it suffers from the state-explosion problem [8]. Binary decision diagrams [9], reduced ordered binary decision diagrams [10] and bounded model checking [11] are some techniques that were developed and applied to make model checking more efficient. However, these techniques are still constrained by the state-explosion problem and become intractable for larger models.

Software testing covers a wide range of diverse methods that are generally incomplete and mainly test for the presence of bugs while verification techniques tests for the absence of bugs [6]. Software testing does not usually provide very high confidence about the correctness of a system, as it only checks for the presence of defects under specific conditions.

Runtime verification [6, 12–17] is an emerging *lightweight* verification technique, where a *monitor* dynamically checks a system under inspection at run time with respect to the system's specification. Runtime verification complements exhaustive verification methods, such as model checking and theorem proving, as well as incomplete solutions, such as testing and debugging. Runtime verification adopts a high degree of rigour like model checking and theorem proving, but trades off some of the rigour and completeness for practicality and efficiency that is inherent in most online software testing methods and approaches.

Most techniques in the literature of runtime verification employ monitors that are *event-triggered*. In event-triggered runtime verification, the system under scrutiny is modified to invoke the monitor every time a *critical event* occurs at run time. A *critical event* refers to a change in the program state that may influence the verdict (i.e., the truthfulness) of one or more properties that the system must/should satisfy. Figure 1.1(a) illustrates how a system and an event-triggered monitor would interact.

In recent years, there has been an increasing level of interest in exploring *time-triggered* approaches for runtime verification [18–22]. In time-triggered runtime verification, the



(a) Event-triggered Approach.

(b) Time-triggered Approach.

Figure 1.1: Overview of established runtime verification approaches.

monitor typically preempts the system under scrutiny periodically at run time to evaluate the properties of interest. Once the monitor is finished checking the set of system properties, it relinquishes control to the system. The monitoring correctness is a primary concern in time-triggered runtime verification because the monitor may miss critical events between two consecutive samples. One approach to tackle this issue is to buffer critical events into auxiliary memory and have the monitor read the event buffer when it samples the program [18]; other approaches are discussed in Chapter 2. The behaviour of the system and the monitor is illustrated in Figure 1.1(b).

Determining the periodicity of the monitor is dependent on the available computational resources and the maximum tolerable latency allowed for the system to detect and report any system property violation. The time from which a property violation occurs to the time the monitor detects the violation is known as the *detection latency*.

Time-triggered runtime verification is advantageous in certain aspects. First, time-triggered monitors have predictable behaviour and incur predictable overhead. Secondly, time-triggered monitors can potentially reduce the amount of incurred runtime monitoring overhead when the sampling period is increased (or the sampling frequency is decreased).

The deployment of runtime monitors for software systems requires an in-depth understanding of the inherent trade-off between execution time and memory overhead. From the observations made in [18, 19], the monitoring overhead with respect to execution time

can decrease if the monitor and the system can increase the amount of memory used at run time (or even after, if offline verification is considered). The reverse also holds; if the available memory resources decrease, then the monitor will likely incur greater overhead with respect to execution time.

The work that is presented in this thesis tackles memory overhead and execution time overhead separately in two different ways. The first part of the thesis (see Chapter 4) focuses on making time-triggered runtime verification applicable to larger scale systems by tackling the NP-complete problem of determining the optimal instrumentation scheme that would yield in the lowest amount of auxiliary memory required to preserve correctness in monitoring. The second part of the thesis (see Chapter 5) presents a novel approach, known as *hybrid runtime verification*, that aims to minimize the amount of time spent in executing verification procedures at run time by combining the advantages of both event- and time-triggered monitoring; hybrid runtime verification is particularly effective when the system encounters both sporadic and bursty critical events at run time.

The approach that one should take highly depends on the nature of the system, the available resources, and the set of properties it should satisfy. For hard real-time systems, predictable monitoring is very important. This is because bursty critical events may cause extremely high monitoring overhead and make the monitor's behaviour hard to predict. As a result, the monitor may cause the system to violate a timing constraint. This would be difficult to debug if the unmonitored system initially satisfies the detected timing violation. Thus, hard-to-predict monitoring behaviour is undesirable. Another benefit of having predictable monitoring for such systems is that the monitors activity can be included in offline schedule computation and verification; this enables designers to check before deployment that timing guarantees will likely not be violated from the added checking at run time. For systems that do not have hard timing deadlines and do not impose tight memory constraints, increasing the efficiency in executing the monitored system with respect to time is beneficial.

1.1 Summary of Contributions

The main contributions that are presented in thesis are twofold:

1. Minimizing the number of locations where events must be buffered in time-triggered runtime verification given some target sampling period is NP-complete [18]. Solving for the optimal solution is intractable for programs with a large number of critical events, even with state-of-the-art solvers. The first part of this thesis introduces three polynomial-time heuristics that aim to scale the applicability of time-triggered runtime verification and shows that the heuristics provide reasonable sub-optimal instrumentation schemes. This work resulted in a publication to the International Conference on Runtime Verification in 2011 [19].

2. Time-triggered runtime verification may reduce the monitoring overhead as the sampling period is increased; however, the monitor can sometimes sample without doing any meaningful work. The second part of this thesis introduces hybrid runtime verification, which aims to exploit the benefits of both event- and time-triggered runtime monitoring to reduce the overall monitoring overhead with respect to execution time. This involves solving an optimization problem that is already difficult to solve for a linear execution trace. To cope with the complexity of the optimization problem, a heuristic is presented that reasonably models the optimization problem. The experimental results demonstrate that the heuristic generally produces monitoring schemes that are equal or better than the most efficient way of monitoring a program using an event- or time-triggered monitor.

1.2 Outline

The rest of the thesis is organized as follows. Chapter 2 presents related work in the literature on methods of improving the efficiency and feasibility of runtime verification. Chapter 3 formally defines and describes all common terminology and concepts that are used in Chapters 4 and 5. Then, Chapter 4 presents three polynomial- time and space heuristics that trade-off optimality for scalability of time-triggered runtime verification.

Some of the secondary observations made in Chapter 4 motivate the work presented in Chapter 5. Chapter 5 explores a novel approach, known as hybrid runtime verification, which attempts to leverage the benefits of both event- and time-triggered monitoring techniques to reduce the runtime monitoring overhead. Chapter 6 summarizes the key findings of the work presented in Chapters 4 and 5. Finally, Chapter 7 describes some future research directions and ideas that may extend the work presented in this thesis.

Chapter 2

Related Work

Existing work and literature on runtime verification and monitoring is presented and discussed in this chapter. Most of the work cited in this chapter pertain to methods and techniques that are used to make monitors for runtime verification more efficient.

In classic runtime verification [13, 23, 24], a system is composed with an external observer, called the monitor. Monitors in runtime verification are typically automatically synthesized from one or more properties that the system should satisfy [14]; the monitor routinely checks the system against these properties and is responsible for detecting property violations at run time. Most work in the literature of runtime verification adopts and uses *event-triggered* monitors [25], where every critical change in the state of the system causes a direct invocation of the monitor for analysis.

Runtime verification has mostly been studied in the context of *linear temporal logic* (LTL) properties [17, 24, 26–28] and, in particular, safety properties [29, 30]. Other languages and frameworks have also been developed for facilitating specification of temporal properties [31–33]. [34] considered runtime verification of ω-languages. In [35], the authors address runtime verification of safety-progress [36, 37] properties.

Organization. The rest of this chapter describes some recently published work that address the challenges in making runtime verification applicable and feasible to more software systems. Section 2.1 presents recent literature that address the issue of reducing the overhead cost of monitoring in runtime verification. Section 2.2 focuses on surveying literature where authors employ time-triggered techniques for runtime verification.

2.1 Reducing Runtime Verification Overhead

Dwyer et al. [38] aim to dynamically adjust the runtime monitoring scheme by incorporating the semantics of the system properties of interest and the program state to reduce the number of monitor invocations at run time. The decision to skip or execute instrumentation code is done at run time. Their framework, known as *Adaptive Online Program Analysis* (AOPA), guarantees that the monitor will correctly detect property violations and empirically observe that the monitoring overhead is reduced because of the optimizations performed on both monitor synthesis and program instrumentation. AOPA runs within Java; the dynamic behaviour of the monitored system is controlled within a Java virtual machine. The authors choose to have the monitors verify the properties online to further reduce the overhead costs of post-processing potentially very large buffers offline.

Barringer et al. [39] demonstrate that a large set of logics used to express safety and liveliness system properties may be expressed using their unifying logic known as EAGLE. They further show that the monitors that are synthesized from properties expressed in EAGLE logic do not require storing the execution trace, which reduces monitoring overhead with respect to memory.

Bodden et al. [40] describes an approach that reduces the monitoring overhead by statically analyzing the program under scrutiny along with its corresponding *tracematches* to eliminate critical events that need not be monitored. A tracematch defines a runtime monitor using a regular expression over an alphabet of critical events in the program. The monitoring technique that they base their method on is aspect-oriented programming (AspectJ). The potential reduction of tracematches occurs in three stages, so that the user has control over the precision of the reduction; each subsequent stage in their procedure increases in computational complexity.

Bodden et al. [41] present two partitioning schemes that alleviate runtime monitoring overhead of widely used and deployed large scale systems. The partitioning schemes distribute the monitoring workload across all of the deployed systems; each instance either will monitor regions of the program or will temporally toggle the monitor so that the overhead is not noticeable (approx. 5% overhead). In this work, the schemes that the authors present ensures that no false positives are reported (but the monitors may report false negatives), thereby making the debugging information easier to use.

Huang et al. [42] introduced a rigorous non-linear feedback controller for the monitor and the monitored program so that the monitor does not cause overload situations. Overload situations are caused by bursts of critical instructions that are monitored over a short period of time. *Software monitoring with controllable overhead* (SMCO) strives to maintain the monitoring overhead below the design threshold (or reference). The feedback controller controls whether the program will be monitored or not. When the monitor approaches or exceeds the target overhead, the monitor is effectively switched off by executing the original of the program. When the monitoring overhead is less than the target overhead, the monitor is invoked by the program executing the instrumented version.

The authors aim to reduce the overhead (with respect to time) incurred from toggling on and off the monitor by making a copy of the the original source code and instrumenting the copy. The monitored program inserts guards at the beginning of function calls to determine whether the un-instrumented or instrumented copy of the program will execute.

While SMCO effectively controls the monitoring overhead, it does not guarantee that all properties are correctly monitored because in overload situations, the monitor is turned off and does not process any critical events that are generated from the program being scrutinized. Since the monitoring accuracy is not 100%, this approach is unsuitable for applications that require all violations to be detected. Furthermore, the authors assume 'infinite' application memory space and double the size of the source code of the program being monitored. In memory-constrained systems, this approach may be infeasible.

Zhu et al. [43] present a hard real-time runtime monitoring solution. They bound the latency of error detection by performing schedulability analysis on the monitor's execution time requirements (i.e. monitoring budget) and its overhead with the real-time schedule of the system that is being scrutinized. Although real-time tasks are assumed to be periodic, the release of critical events (also known as PVEs in [43]) are aperiodic. The main goal of this work is on time-aware instrumentation.

2.2 Time-triggered Runtime Verification

Pike et al. [21] proposed an embedded domain-specific language in Haskell, known as *Copilot*, that compiles into small constant-time and constant-space C monitors that may

be used in time-sensitive software systems (i.e. hard real-time). The monitors that are generated using Copilot periodically sample the program state to verify system properties. The authors of [21] do not consider missing state changes that result from sampling because within a hard real-time context, the monitor and program are assumed to share a global sense of time and a static periodic schedule. The monitors synthesized using Copilot are conservative in that they will report false positives (of system property violations), true positives and true negatives, but not false negatives.

While the work in [21] is suitable specifically for hard real-time applications, it is difficult to apply this work to soft-RT or non-RT applications. The rigour and predictability of real-time systems allowed the authors to synthesize correctly functioning monitors. However, the periodicity and predictability of real-time systems does not hold for other types of systems, so the monitor's correctness is not guaranteed in other applications. Furthermore, Copilot synthesized monitors are only capable of monitoring global variables, which may encourage poorly encapsulated (and difficult to manage) code. This may cause issues in software maintainability for large-scale software systems.

Stoller et al. [22] present a technique called *runtime verification with state estimation* (RVSE), which aims to cope with the pitfalls of time-triggered runtime verification with respect to program correctness by utilizing a hidden Markov model of the monitored program to estimate missing program states incurred from sampling-induced gaps. The state estimation of the hidden Markov model determines the probability of a state sequence given an observed sequence, which can then be used to infer the probability that a system property is satisfied by an execution of the program.

The monitor's correctness is not guaranteed because this method utilizes state estimation. This technique may be useful in determining potential system property violation (useful bug reporting), but cannot be deployed in systems that require the monitor to report all violations confidently.

Bartocci et al. [44] extends the work of [22] and [42] by combining the two approaches together to form a more complete monitoring framework with controllable overhead. Specifically, the authors of [44] leverage RVSE to infer the likelihood of property satisfaction/violation whenever the monitor is temporarily disabled for overhead control purposes. They also extend the theory behind RVSE and introduce techniques that can predict the *criticality level* of a system property, which is a function of the expected distance to the violation

of the property. The criticality levels of system properties of interest can then determine the allocation of the available monitoring resources. While [44] improves the usability and usefulness of [42], this runtime verification framework does not guarantee correct state reconstruction.

Another approach to time-triggered runtime verification was presented by Bonakdarpour et al. [18], where the correctness of the monitor and the verification process is achieved by determining the longest possible sampling period of the monitor (without instrumentation). The authors consider buffering critical events into auxiliary memory as a viable way to increase the sampling period while preserving correct program state reconstruction. Extending the longest sampling period involves solving an NP-complete optimization problem that aims at minimizing the size of auxiliary memory required for the monitor to correctly reconstruct all program state sequences at run time. The majority of the work presented in the remaining chapters of this thesis extends this particular approach.

Navabpour et al. [20] propose methods of time-triggered runtime verification that utilize *symbolic execution* [45] to predict the set of all feasible paths of the system prior to run time. From the set of predicted paths, the longest sampling period is computed. At run time, the monitor will adjust its sampling period based on the path that is taken by the program. The authors observe that some execution paths contain unevenly distributed critical events; to further refine this monitoring technique, they consider determining code regions within each path and assign the monitor to vary the sampling period depending on the code region. This path-aware approach for time-triggered runtime verification aims to reduce the amount of runtime monitoring overhead incurred from sampling by enabling the monitor to adapt to local sampling periods rather than a global sampling period. Global sampling periods do not efficiently use the available monitoring resources.

Chapter 3

Preliminaries

In this chapter, the common terms and concepts used throughout the subsequent chapters are defined and explained. Section 3.1 introduces terms and symbols that are used in the definitions presented in the rest of the chapter. Section 3.2 explains some basic concepts in control-flow analysis that is required for determining correct monitoring behaviour in time-triggered runtime verification. Section 3.3 formally describes how control-flow analysis is used to determine a 'safe' sampling period for runtime verification and how the sampling period can be effectively increased to reduce the monitoring activity.

3.1 Checking System Properties at Run Time

Runtime verification consists of a monitor and a program/system under inspection. The monitor is typically synthesized from one or more properties that the system should satisfy and runs in parallel with the system at run time. The program invokes the monitor whenever it executes an event that may change the logical result of one or more properties in event-triggered runtime verification. In time-triggered runtime verification, the monitor interrupts the program execution at regular time intervals to observe the state(s) of the program and check the set of properties the system is expected to satisfy.

The state of the program is determined by evaluating the value of a set of variables being monitored. Formally, let P be a program and Π be a logical property (e.g., in LTL),

where P is expected to satisfy Π. Let $\mathcal{V}_\Pi$ denote the set of variables that participate in Π. Let a *critical event* be a change of any variable in $\mathcal{V}_\Pi$.

In *event-triggered runtime verification*, the instrumented version of P will call/invoke the monitor to evaluate Π whenever P encounters a critical event. Invoking the monitor every time that some variable in $\mathcal{V}_\Pi$ changes guarantees correct program state reconstruction at run time.

In *time-triggered runtime verification*, a monitor reads the value of variables in $\mathcal{V}_\Pi$ at a fixed sampling frequency and evaluates Π. Accurate program state reconstruction of P between two consecutive samples is the main challenge in using this mechanism; e.g., if more than one critical event occurs between two consecutive samples, then the monitor may fail to detect violations of Π. The monitor must sample at a 'safe' rate to facilitate correct program state reconstruction of P, typically determined by applying control-flow analysis [18–20].

3.2 Control-flow Analysis

In control-flow analysis, *control-flow graphs* are used to represent the program P. Definition 1 formally defines what a control-flow is and what it represents:

Definition 1 *The* control-flow graph *(CFG) 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.*
- *v^0: is the* initial vertex *with in-degree 0, which represents the initial basic block of P.*
- *A: is a set of* arcs *(u, v), where $u, v \in V$. An arc (u, v) exists in A if and only if the execution of basic block u immediately leads 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 represents the* best-case execution time *(BCET) of the source basic block. The best-case execution times are expressed in terms of the number of clock cycles required to execute the basic blocks.*

```
1  scanf("%d", &a);
2  if ( a % 2  == 0 ) {
3      printf("%d is even", a);
4  } else {
5      b = a / 2;
6      c = a / 2 + 1;
7      printf("%d is odd", a);
8  }
9   d = b + c;
10 end program
```

Figure 3.1: A simple C program.

A *basic block* in control-flow analysis represents a linear sequence of instructions in a program P without any jumps or jump targets (i.e., goto statements in C). In other words, only the last instruction in a basic block can be a branching statement. A *critical basic block* is a basic block that contains one or more critical events/instructions.

Control-flow graphs are used in time-triggered runtime verification to determine the maximum sampling period that the monitor can operate with while preserving correct program state reconstruction for checking the set of system properties. The next subsection describes the operations and procedures used to determine the sampling period.

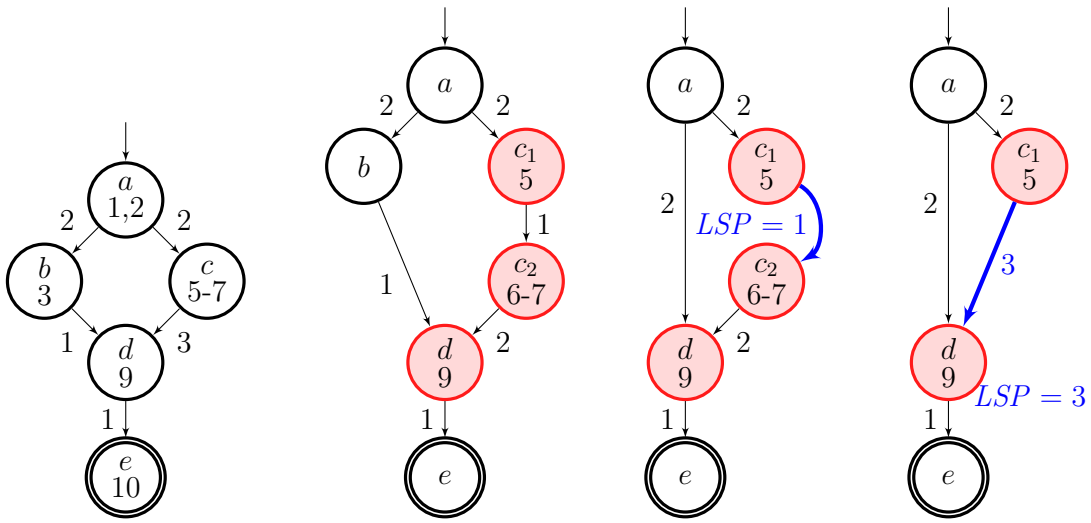
Consider the C program in Figure 3.1. Figure 3.2(a) shows the resulting control-flow graph of the C program. The arc weights in the control-flow graph are computed based on the assumption that the BCET of each line of code is one time unit. Vertices of the graph in Figure 3.2(a) list the corresponding line numbers of the C program in Figure 3.1.

3.3 Time-triggered Runtime Verification

3.3.1 Transforming Control-flow Graphs

The control-flow graph of P must go through several transformations to identify the maximum possible period that a monitor may sample P with while guaranteeing accurate program state reconstruction.

Let $CFG_P = \langle V, v^0, A, w \rangle$ be a control-flow graph represent to the program P. The first procedure to perform on CFG_P is to determine the *critical vertices*. A *critical vertex* is a



(a) CFG (Section 3.2)

(b) Step 1 (Section 3.3.1)

(c) Step 2 (Sections 3.3.1 and 3.3.2)

(d) Step 3 (Section 3.3.3)

Figure 3.2: Steps for obtaining optimized instrumentation and longest sampling period.

vertex $v \in V$ that represents a critical basic block (i.e., the basic block contains instructions that change one or more variables in $\mathcal{V}_\Pi$). For a critical basic block/vertex that contain more than one critical instruction, the vertex must be split into multiple vertices, where each vertex represents a critical basic block with exactly one critical event/instruction. Revisiting the program shown in Figure 3.1, if variables `b`, `c`, and `d` are in $\mathcal{V}_\Pi$, then lines 5, 6 and 9 are critical instructions. Since instructions in lines 5 and 6 are critical and they both reside in basic block c, c will be split into c_1 and c_2 as shown in Figure 3.2(b); the highlighted vertices in the figure denote the critical basic blocks.

After ensuring that each critical vertices in the control-flow graph contain exactly one critical instruction, the graph is transformed into a *critical control-flow graph*. A critical control-flow graph has the following characteristics:

- The initial vertex is non-critical.
- The graph may possible contain a non-critical vertex with out-degree zero (i.e., if the program terminates).

- All other vertices in the graph are critical vertices. Figure 3.2(c) shows the corresponding critical control-flow graph of Figure 3.2(b).

The function $T(CFG, v)$ is defined to facilitate the transformation of a control-flow graph into a critical control-flow graph. The inputs of T are the current control-flow graph, CFG, and a non-critical vertex $v \in V$ that should be removed from CFG, where V is the set of vertices in CFG. $T(CFG, v)$ is only applicable when $v \in V \backslash \{v^0\}$ and the out-degree of v is positive. The output of $T(CFG, v)$ is a modified control-flow graph, $CFG' = \langle V', v^0, A', w' \rangle$, and is obtained by the following ordered steps:

1. Let A'' be the set $A \cup \{(u_1, u_2) | (u_1, v), (v, u_2) \in A\}$. Observe that if an arc $(u_1, u_2) \in A$, then A'' will contain parallel arcs (such arcs can be distinguished by a simple indexing or renaming scheme). Parallel arcs are eliminated in Step 3.

2. For each arc $(u_1, u_2) \in A''$,

$$w'(u_1, u_2) = \begin{cases} w(u_1, u_2) & \text{if } (u_1, u_2) \in A \\ w(u_1, v) + w(v, u_2) & \text{if } (u_1, u_2) \in A'' \backslash A \end{cases} \tag{3.1}$$

3. If there exist parallel arcs from vertex u_1 to u_2, only include the arc with minimum weight in A''.

4. Finally, the set of arcs and vertices are updated:

$$A' = A'' \backslash \{(u_1, v), (v, u_2) \mid u_1, u_2 \in V\} \tag{3.2}$$

$$V' = V \backslash \{v\} \tag{3.3}$$

Special Case: If u and v are two non-critical vertices and $(u, v), (v, u) \in A$, then removing one of the vertices, e.g., u, results in the self-loop (v, v). This self-loop may safely be removed. A loop that does not contain critical instructions does not affect the sampling period.

Applying $T(CFG, v)$ on all non-critical vertices $V \backslash \{v^0\}$ with positive out-degrees in CFG_P results in the critical control-flow graph of P. For the example program in Figure 3.1, the corresponding critical control-flow graph obtained by first ensuring that all critical vertices contain exactly one critical instruction followed by applying the transform $T(CFG, v)$ is shown in Figure 3.2(c).

3.3.2 Determining the Longest Sampling Period (LSP)

The *longest sampling period* (LSP) is the maximum sampling period the monitor can sample with and preserve correct program state reconstruction. In other words, the longest sampling period is the minimum timespan between two successive changes of any two variables in $\mathcal{V}_\Pi$ (i.e., the minimum distance between all pairs of critical vertices). The longest sampling period may be computed using either the control-flow graph or critical control-flow graph of a program P. Definition 2 formally defines the longest sampling period using both types of control-flow graphs.

Definition 2 *Let $CFG_P = \langle V, v^0, A, w\rangle$ be the* control-flow graph *of a program P such that each critical vertex contains exactly one critical event/instruction. Let $V_c \subseteq V$ be the set of vertices that correspond to critical basic blocks of CFG_P; and Π_c be the set of paths $\langle v_h \to v_{h+1} \to \cdots \to v_{k-1} \to v_k\rangle$ in CFG such that $v_h, v_k \in V_c$ and $v_{h+1}, \ldots, v_{k-1} \in V \setminus V_c$. The* longest sampling period (LSP) *for CFG_P is*

$$LSP_{CFG_P} = \min_{\pi \in \Pi_c} \left\{ \sum_{\substack{(v_i, v_j) \in A \\ v_i, v_j \in \pi}} w(v_i, v_j) \right\} \tag{3.4}$$

Alternatively, apply the transformation function $T(CFG, v)$ to CFG_P on the applicable non-critical vertices to create the critical control-flow graph *of P, $CFG'_P = \langle V, v^0, A, w\rangle$. Again, let $V_c \subseteq V$ be the set of critical vertices in CFG'_P. The longest sampling period (LSP) for CFG'_P is*

$$LSP_{CFG'_P} = \min\{w(v_1, v_2) | (v_1, v_2) \in A \wedge v_1 \in V_c\} \tag{3.5}$$

Recall that the critical control-flow graph of the program in Figure 3.1 is shown in Figure 3.2(c). The arc weights $w(c_1, c_2) = 1$ and $w(c_2, d) = 2$ are the possible values of the longest sampling period for this program. Therefore, by Definition 2, the longest sampling period for the program is 1. In other words, if the monitor samples this program with a periodicity of 1, all potential property violations can be detected.

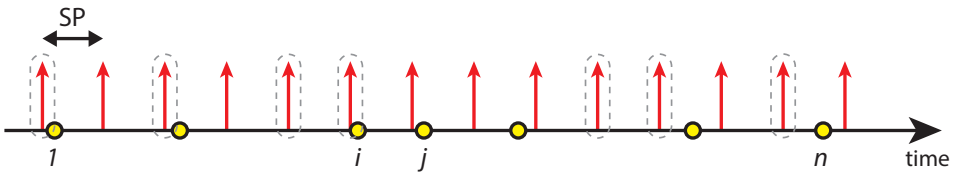


Figure 3.3: Illustrating redundant samples in time-triggered runtime verification.

3.3.3 Increasing the Longest Sampling Period

While sampling at the longest sampling period computed from the critical control-flow graph guarantees correct program state reconstruction, it may impose a significant amount of monitoring overhead. Consider the execution trace shown in Figure 3.3, where the critical events are marked on the timeline as circles. Assume that the timespan between events i and j ultimately determines the longest sampling period. The monitor must sample with this sampling period (or smaller) to preserve correct program state reconstruction. To minimize the monitoring overhead, the monitor should sample with the longest sampling period (i.e., to minimize total number of samples at run time). Figure 3.3 illustrates when the monitor would sample the program with the longest sampling periodThe dotted ovals around some of the arrows (i.e. samples taken by the monitor) are samples where the monitor does not do any meaningful work because no critical events occurred during those sampling periods. Such samples are called *redundant samples*.

To reduce the number of redundant samples, the sampling period must be increased. To preserve the correctness in monitoring and program state reconstruction while increasing the sampling period, another graph transformation function is defined. This function is known as the *instrumenting transformation*. Let $IT(CFG, v)$ be the instrumenting transform function. This function may be applied to critical vertices in the critical control-flow graph, CFG'_P. $IT(CFG, v)$ consists of the following ordered steps:

1. Let $(u, v) \in A$, where v is a critical vertex. Apply transformation $T(CFG'_P, v)$.
2. Append an instruction $i' : a' \rightarrow a$ to the sequence of instructions corresponding to basic block u, where a' is an auxiliary memory location. The instructions of basic block u is now $inst_u = inst_u \langle i, i' \rangle$.

Note that adding the extra instruction i' does not affect the calculation of the sampling period. This is because adding instrumentation only increases the best case execution time of a basic block. By maintaining the calculated sampling period, no critical instruction is overlooked.

Unlike non-critical vertices, the issue of loops involving critical vertices need to be handled differently. Suppose that u and v are critical vertices and $(u, v), (v, u) \in A$. Consider removing u. This results in a self-loop (v, v), where $w(v, v) = w(u, v) + w(v, u)$. The loop iterates an unknown number of times at run time, so it is difficult to determine the upper bound on the size of auxiliary memory required to collapse vertex v. To ensure correctness, the transformation IT is forbidden for critical vertices that have self-loops.

Figure 3.2(d) illustrates this instrumenting transformation on critical vertex c_2. Applying $T(CFG, v)$ with $v = c_2$, the resulting graph returned removes c_2 and an arc is added directly between vertices c_1 and d, with the sum of weights of arcs (c_1, c_2) and (c_2, d) in the previous graph (see Figure 3.2(c)). The second step of the transformation inserts an instruction to save the critical event that is in c_2, thereby effectively increasing the longest sampling period to 3.

The maximum *violation detection latency* (i.e., the time elapsed between the occurrence of a property violation and the detection of the violation) of Π, the availability of auxiliary memory and other system constraints limit the number of times $IT(CFG, v)$ can be applied to increase the longest sampling period.

Chapter 4

Heuristics for Time-triggered Runtime Monitoring

4.1 Introduction

In the literature, deploying monitors for runtime verification involves instrumenting the program under inspection, so that upon occurrence of events (e.g., change in a variable's value) that may change the truthfulness of a property, the monitor is called to (re-)evaluate the property; this method is known as *event-triggered* runtime verification, because each change prompts a re-evaluation. Event-triggered runtime verification suffers from two drawbacks: (1) *unpredictable* overhead, and (2) possible *bursts* of events at run time. These defects can lead to undesirable transient overload situations in time-sensitive systems such as real-time embedded safety-critical systems. To address these issues, Bonakdarpour et al. introduced a notion of *time-triggered* runtime verification [18], where a monitor runs in parallel with the program and samples the program state periodically to evaluate a set of system properties.

The main challenge in time-triggered runtime verification is to guarantee accurate program state reconstruction when the monitor samples the program. [18] introduced an optimization problem where the objective is to find the minimum number of critical events that need to be buffered for a given sampling period. Consequently, the time-triggered

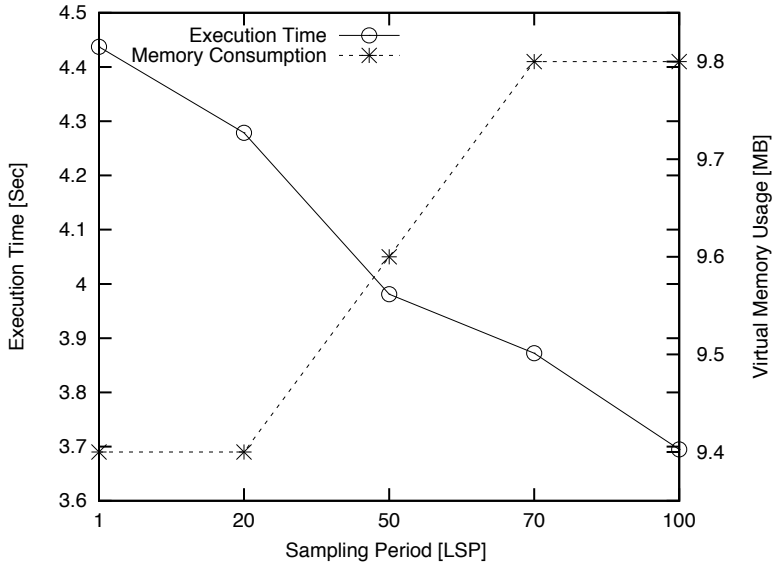


Figure 4.1: Memory usage vs. sampling period [18].

monitor can successfully reconstruct the state of the program between two successive samples using the buffering scheme returned by solving the optimization problem. [18] proved that this optimization problem is NP-complete and proposed a transformation of this problem to an instance of the *integer linear programming* (ILP) problem, which is described in Section 4.2. Transforming this problem enables the capability of employing powerful ILP-solvers to identify the minimum buffer size and instrumentation instructions for state reconstruction. It is possible to solve the corresponding ILP model for some applications, but for larger applications, the complexity of the problem poses a serious stumbling block.

The intractability of the optimization problem prompted an investigation of applying efficient (i.e., polynomial-time) heuristics to identify near-optimal solutions for the optimization problem. In [18], the authors observed that the impact of increasing the monitor's sampling period significantly reduced the overall execution time of the monitored program with a very small (almost negligible) increase in runtime memory usage of the monitored program. Figure 4.1, taken from [18], shows the total execution time and memory usage of blowfish, a benchmark program from MiBench [46], for target sampling periods of 1, 20, 50, 70, and 100 times the LSP. When the target sampling period is increased to $100 \times LSP$, the monitored program's memory usage only increased by 4%. Other experimental data

in [18] show similar patterns in the results. This observation suggests that nearly optimal solutions to the optimization problem are likely sufficiently effective.

With this motivation, three polynomial-time heuristics were developed to find near-optimal solutions to the optimization problem defined in [18]. All three heuristics are over-approximations and, hence, sound (they do not cause overlooking of events to be monitored). The first heuristic is a *greedy* algorithm that aims at instrumenting variables that participate in many execution branches. The second heuristic is based on a 2-approximation algorithm for solving the *minimum vertex cover problem*. Intuitively, this heuristic instruments variables that are likely to cover all cases where variable updates occur within time intervals less than the target sampling period. The third heuristic is a genetic algorithm, an evolutionary heuristic where the evolution of the population aims to minimize the number of variables that need to be instrumented and buffered.

The collected experimental data show that these three heuristics are significantly faster than the ILP-based solution described in [18]. More importantly, the solutions returned by all three algorithms lead to a negligible increase in instrumentation overhead and total memory usage at run time as well as negligible increase in the total execution time of the monitored program. Also, the experimental data show that extra instrumentation instructions are evenly distributed between samples. Moreover, the genetic algorithm generally produces instrumentation schemes closest to the optimal solution as compared to the other heuristics. The experimental results empirically suggest that the NP-completeness of the optimization problem is not an obstacle when applying time-triggered runtime verification in practice.

Organization. The rest of this chapter is organized as follows. Section 4.2 summarizes the integer linear programming model used by Bonakdarpour et al. in [18]. Section 4.3 describes and illustrates the three heuristics, which include the greedy and minimum vertex-cover heuristics, as well as the genetic algorithm. Experimental results are presented and analyzed in Section 4.4. Section 4.5 finishes the chapter with some concluding remarks.

4.2 Optimizing Memory Overhead in Time-triggered Runtime Verification

Given a critical control-flow graph, the goal of [18] is to optimize two factors through a set of $IT(CFG, v)$ transformations: (1) minimizing auxiliary memory, and (2) maximizing sampling period. Bonakdarpour et al. showed that this optimization problem is NP-complete [18]. The remainder of this section is organized as follows. Section 4.2.1 briefly states the integer linear programming problem in its general form. After that, Section 4.2.2 describes the integer linear programming model used by the authors in [18] to solve the optimization problem. Chapter 4 extends this work by exploring more efficient algorithms to compute near-optimal solutions for this problem.

4.2.1 Integer Linear Programming

The integer linear programming (ILP) problem is of the form:

$$\begin{cases} \text{Minimize} & c.\mathbf{z} \\ \\ \text{Subject to} & A.\mathbf{z} \geq \mathbf{b} \end{cases} \tag{4.1}$$

where A (a rational $m \times n$ matrix), c (a rational n-vector) and $\mathbf{b}$ (a rational m-vector) are given, and $\mathbf{z}$ is an n-vector of integers to be determined. In other words, solving this problem involves finding the minimum of a linear function over a feasible set defined by a finite number of linear constraints. It can be shown that a problem with linear equalities and inequalities can always be put in the above form, implying that this formulation is more general than it might look.

4.2.2 Integer Linear Programming Model

Let $CFG'_P = \langle V, v^0, A, w \rangle$ be the critical control-flow graph of a program P. In the ILP mapping of the optimization problem, the following sets of integer variables are defined:

- $\mathbf{x} = \{x_v | v \in V\}$, where x_v is a binary integer variable. $x_v \in \{0, 1\}$. If $x_v = 1$, then v is removed from V and the critical event in the basic block corresponding to v is buffered as history in auxiliary memory. If $x_v = 0$, then v remains in V and no instrumentation is required to save the critical event in the basic block corresponding to v in auxiliary memory.
- $\mathbf{a} = \{a_v | v \in V\}$, where a_v are integer variables that represent the weight of arcs originating from vertex v.
- $\mathbf{y} = \{y_v, y'_v | v \in V\}$ are choice variables, where y_v and y'_v are integer variables. The use of choice variables are described later.

4.2.2.1 Objective Function

The *objective function* of this ILP model is:

$$\min \sum_{v \in V} x_v \tag{4.2}$$

Minimizing Equation 4.2 will minimize the set of vertices removed from CFG'_P. The set of vertices removed indicate the critical events in the program P that must be instrumented to save the event into auxiliary memory to increase the effective longest sampling period while preserving the correctness of program state reconstruction.

4.2.2.2 Initial Basic Block Constraints

The initial basic block has several unique constraints. The vertex v^0 corresponds to the initial basic block in the program P. The following constraints are imposed on v^0.

$$x_{v^0} = 0 \tag{4.3}$$

$$a_{v^0} = w(v^0) \tag{4.4}$$

These two constraints ensure that the initial basic block is never included in the set of removed (i.e., instrumented) vertices because the monitor should sample the at the beginning of the program to extract the initial values of the variables in $\mathcal{V}_\Pi$.

4.2.2.3 Constraints on Arc Weights and Internal Vertices

After performing the necessary instrumenting transformations (i.e., $IT(CFG, v)$) on CFG'_P, the effective longest sampling period should be equal or larger that the target sampling period that the user specifies. Let SP be the target longest sampling period. Then for every arc $(u, v) \in A$,

$$a_u + SP \cdot x_v \geq SP \tag{4.5}$$

In other words, the solution should ensure that all arc weights become at least SP. If any instance of this constraint cannot be met, then no solution exists such that P can be sampled at SP while preserving correctness in program state reconstruction.

Whenever a vertex is removed by applying the transformation $IT(CFG, v)$, the arc weights will change. The arc weight of vertex v, a_v, is subject to two different cases:

Case 1 If $x_v = 0$ then $a_v = w(v)$

Case 2 If $x_v = 1$ then $a_v = w(v) + w(u)$, where $(u, v) \in A$. Even though vertex v is removed upon applying $IT(CFG, v)$, a_v is used to retain the value of the newly created arc for simplicity. Also, outgoing arcs from u automatically satisfy Equation 4.5.

To enforce mutual exclusivity of these two cases in the model and correct arc weights, the set of choice variables, $\mathbf{y}$, are used. The choice variables in this model exhibit the following properties:

Property 1: y_v and y'_v are such that one of them is zero and the other is a_u. This property enforces mutual exclusivity.

Property 2: If $x_v = 1$, then $y_v = a_u$ and $y'_v = 0$. If $x_v = 0$, then $y_v = 0$ and $y'_v = a_u$.

A special data structure, called *Special Ordered Set Type 1* ($sos_1(\ldots)$) [47], is used to enforce the first property. This data structure ensures that at most one variable can take on a non-zero value. The constraints of the two properties may be expressed as:

$$y_v + y'_v = a_u \tag{4.6}$$

$$sos_1(y_v, y'_v) \tag{4.7}$$

$$1 \leq x_v + y'_v \leq a_u \tag{4.8}$$

The following constraints implement Cases 1 and 2, respectively.

$$w(v) + a_u - y'_v = a_v \tag{4.9}$$

$$y_v + w(v) = a_v \tag{4.10}$$

These five constraints are duplicated for each incoming arc to vertex v. Since the depth of nested conditional statements is not normally high, it is unlikely that models of programs would cause an explosion in the number of a-variables in the model.

4.2.2.4 Loop Constraint

To ensure that self-loops are not removed, one more constraint is added to the ILP model. For each cycle $v_1 \to v_2 \to \cdots \to v_n \to v_1$, at most $n-1$ vertices can be removed using $IT(CFG, v)$:

$$\sum_{i=1}^{n} x_{v_i} \leq n - 1 \tag{4.11}$$

During the construction of the (critical) control-flow graph, cycles are already detected, so there is no need to explore the graph again to identify them.

4.3 Heuristics

In order to tackle the intractability of the optimization problem for larger software systems, [18] proposed a mapping of the problem to an instance of ILP. The ILP model enabled the authors to utilize state-of-the-art ILP-solvers to solve the models for relatively small benchmark programs. Some additional experiments were conducted on programs from the same benchmark using this ILP mapping. The results are tabulated in Table 4.1. The ILP solver's performance quickly degrades as the size of the critical control-flow graph increases.

With the observation described in Section 4.1 that trading off small amounts of memory can significantly increase the effective longest sampling period, three heuristics were explored. Each heuristic is described in the remainder of this section. All three heuristics take a control-flow graph G and a desired sampling period SP as input and return a set

Subroutine 4.1 PruneCFG(G,SP)

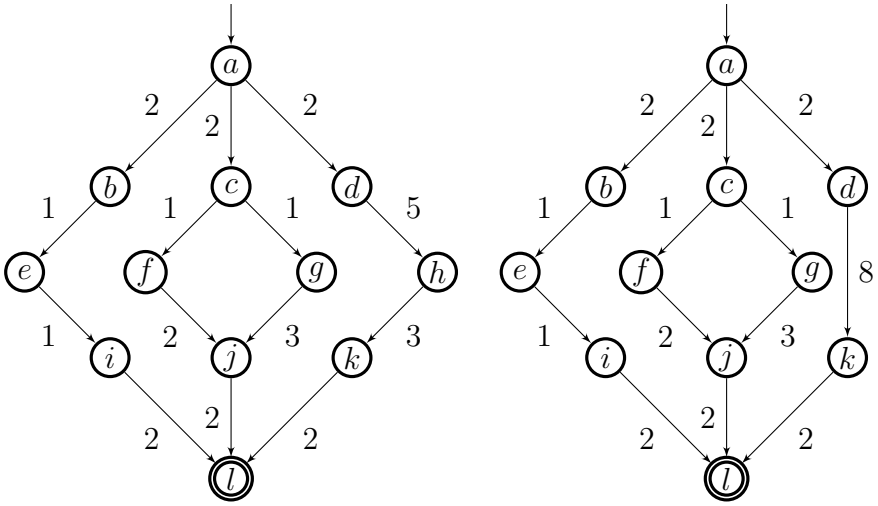
```
1: for v ∈ CFG.V do
2:     if (w(u, v) > SP ∀(u, v) ∈ CFG.A) ∧ (w(v, u) > SP ∀(v, u) ∈ G.A) then
3:         G = CollapseNode(G,v)
4:     end if
5: end for
6: return G
```

U of vertices to be deleted as prescribed by the transformation $IT(CFG, v)$ defined in Section 3.3.3. This set identifies the location where additional instructions should be interleaved in the program under examination to buffer critical events for the time-triggered monitor to process upon the next sample it takes.

In the remainder of this section, the control-flow graph shown in Figure 4.2(a) will be used to illustrate how the various heuristics work. In the greedy and minimum vertex covered based heuristics, both of the heuristics initially go through a procedure referred to as *pruning* the input control-flow graph. Pruning a control-flow graph involves removing all vertices whose weights of all its incoming and outgoing arcs are greater than or equal to SP by applying the transformation $T(CFG, v)$ defined in Section 3.3.1. In the two heuristics that prune the input control-flow graph, such vertices may be safely ignored because the longest sampling period is unaffected by these vertices. The procedure to prune a control-flow graph is shown in Subroutine 4.1. Subroutine 4.1 calls Subroutine 4.2 to 'collapse' (or remove) the selected node from the current control-flow graph. Note that Subroutine 4.2 is equivalent to the function $T(CFG, v)$. For the example shown in Figure 4.2(a), Figure 4.2(b) shows the resulting graph after pruning when $SP = 3$. Vertex h in this case is removed because both the incoming and outgoing arcs are greater than or equal to SP.

Subroutine 4.2 CollapseNode(G,*node*)

```
1: Incoming = {u : (u, node) ∈ G.A}
2: Outgoing = {v : (node, v) ∈ G.A}
3: for u ∈ Incoming do
4:     for v ∈ Outgoing do
5:         if (u, v) ∈ G.E then
6:             w(u, v) = min{w(u, v), w(u, node) + w(node, v)}
7:         else
8:             w(u, v) = w(u, node) + w(node, v)
9:         end if
10:    end for
11: end for
12: Destroy all incoming and outgoing arcs to node
13: Remove node from G
14: return G
```



(a) Example. (b) Prune CFG for $SP = 3$.

Figure 4.2: CFG used for illustrating heuristics.

Heuristic 4.3 Greedy

Input: A critical control-flow graph $G = \langle V, v^0, A, w \rangle$ and target sampling period SP.
Output: A set U of vertices to be deleted from G.

```
1: U := {};
2: G := PruneCFG(G, SP);

3: while (MW(G) < SP ∧ U ≠ V) do
4:     v := GreedySearch(G);
5:     G := CollapseVertex(G, v);
6:     U := U ∪ {v};
7: end while

8: if (U = V) then declare failure;
9: return U;
```

4.3.1 Heuristic 1: Greedy Heuristic

The first heuristic presented is a simple greedy algorithm (see Heuristic 4.3). The explanation of the heuristic is illustrated by using the example control-flow graph shown in Figure 4.2(a) and $SP = 3$:

- First, it prunes the input control-flow graph G (Line 2). After pruning the graph in Figure 4.2(a), the graph shown in Figure 4.2(b) is obtained.
- Next, it explores G to find the vertex, v, incident to the maximum number of incoming and outgoing arcs whose weights are strictly less than SP (Line 4). The intuition behind the selection of this vertex to delete/collapse is that such a vertex results in removing a high number of arcs whose weights are less than SP; this should have a greater (if not the greatest) impact on increasing the longest sampling period of the graph. From the pruned graph shown in Figure 4.2(b), vertex c would be the first selected candidate to greedily collapse because it has a total of 3 incoming and outgoing arcs combined that are less than SP (see Figure 4.3(a)). The other

Subroutine 4.4 GreedySearch(G,SP)

1: bestcount = 0
2: node = $\emptyset$
3: **for** $v \in G.V$ **do**
4: **if** $|\{a : (u, v) \in V.A, u \in G.V \wedge w(u, v) = SP\} \cup \{a : (v, u) \in V.A, u \in G.V \wedge w(v, u) = SP\}| >$ bestcount **then**
5: $bestcount = |\{a : (u, v) \in V.A, u \in G.V \wedge w(u, v) = msp\} \cup \{a : (v, u) \in V.A, u \in G.V \wedge w(v, u) = msp\}|$
6: $node = v$
7: **end if**
8: **end for**
9: **return** *node*

candidates for collapsing have fewer than 3 incoming and outgoing arcs whose weights are less than SP.

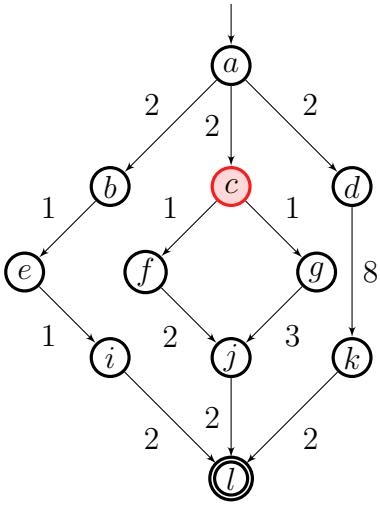
- Then, it collapses vertex v identified on Line 4. This operation (Line 5) results in merging incoming arcs to v with outgoing arcs from v in the fashion described in the transformation function $T(CFG, v)$. Applying this transformation on vertex c in the example results in the graph shown in Figure 4.3(b).

- Obviously, the basic block corresponding to vertex v contains a critical instruction that requires buffering (i.e., added instrumentation to save the event into memory). Thus, v is added to U (Line 6). Note that Lines 5 to 6 is performs actions that are very similary to the transformation $IT(CFG, v)$.

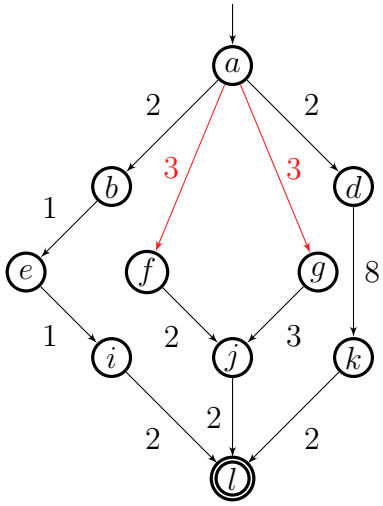
- Lines 3-7 are repeated until the longest sampling period of G is greater than or equal to SP (i.e., the minimum arc weight currently in the graph after applying the transformation). The terminating condition is expressed within the while-loop condition on Line 3. In the running example, this heuristic terminates after the set of vertices $\{b, c, d, e, i, j, k\}$ are removed from G. The effective control-flow graph is shown in Figure 4.3(c).

- If the graph cannot be further transformed, (i.e., only the source and sink vertices remain in the graph), then the graph's structure will not permit increasing the sampling

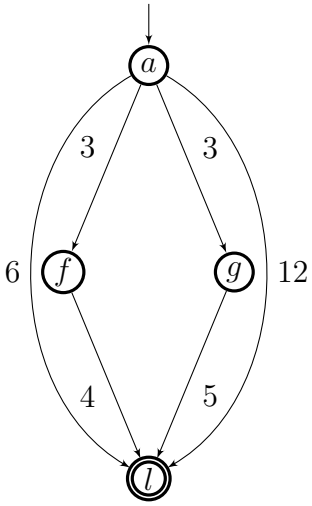
period to SP and the algorithm declares failure.



(a) Greedily select.



(b) Collapse node.



(c) Solution.

Figure 4.3: Illustrations of Heuristic 1.

Subroutine 4.5 Approximate-Vertex-Cover(G)

```
cover = ∅
arcs = G.A
while arcs ≠ ∅ do
    (u, v) = randomly select arc to remove from arcs, u, v ∈ G.V
    cover = cover ∪{u, v}
    remove all incoming and outgoing arcs from u, v
end while
return cover
```

4.3.2 Heuristic 2: Minimum Vertex Cover Heuristic

The second heuristic that was explored is an algorithm based on a solution to the *minimum vertex cover* problem. The minimum vertex cover problem is defined as follows:

Given a (directed or undirected) graph $G = \langle V, E \rangle$, the goal is to find the minimum set $U \subseteq V$, such that each edge in E is incident to at least one vertex in U.

The minimum vertex cover problem is NP-complete, but there exists several approximation algorithms that find nearly optimal solutions. [48] describes a 2-approximation algorithm for the minimum vertex cover problem. This heuristic employs the 2-approximation algorithm in [48] to determine an approximate minimum vertex cover for the control-flow graph under examination. The pseudocode for this approximation algorithm is shown in Subroutine 4.5.

Heuristic 4.6 presents the minimum vertex cover based heuristic as pseudocode. This algorithm works as follows and is illustrated using the example control-flow graph shown in Figure 4.2(a).

- First, it prunes G (Line 2). Figure 4.2(b) shows the resulting graph after pruning the graph in Figure 4.2(a).
- Next, an approximate minimum vertex cover of graph G is computed (Line 4), denoted as *vc*. The graph that is used to determine the minimum vertex cover consists of arcs whose weights are strictly less than *SP* are considered. The subgraph that

Heuristic 4.6 Vertex Cover Based

Input: A critical control-flow graph $G = \langle V, v^0, A, w \rangle$ and desired sampling period SP.

Output: A set U of vertices to be deleted from G.

```
1: U := {};
2: G := PruneCFG(G, SP);

3: while (MW(G) < SP ∧ U ≠ V) do
4:    vc := Approximate-Vertex-Cover(G);
5:    for each vertex v ∈ vc do
6:       G := CollapseNode(G, v);
7:       U := U ∪ {v};
8:    end for
9: end while

10: if (U = V) then declare failure;
11: return U;
```

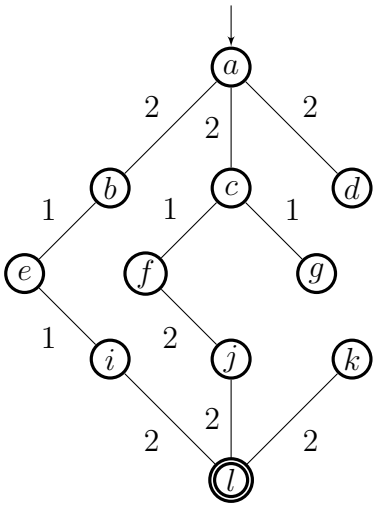
is generated from the pruned graph in Figure 4.2(b) is shown in Figure 4.4(a). The intuition behind determining the minimum vertex cover is that collapsing all vertices in vc may result in removing all arcs whose weights are strictly less than SP because the graph is pruned and the vertex cover vc covers all arcs of the graph. The approximation minimum vertex cover algorithm adopted from [48] is a non-deterministic randomized algorithm and may produce different covers for the same input graph. To improve the solution, Line 4 is invoked multiple times (this parameter may be changed by the user) and of the generated approximate minimum vertex covers, the heuristic selects the smallest vertex cover. This is abstracted away from the pseudo-code. Figure 4.4(b) shows an approximate minimum vertex cover from the subgraph in Figure 4.4(a).

- Then, similar to Heuristic 4.3, vertices $v \in vc$ are collapsed (Lines 5-7). The graph transform operation (Lines 5-7) results in merging incoming arcs to v with outgoing arcs from v in the fashion described by transformation function $T(CFG, v)$. The

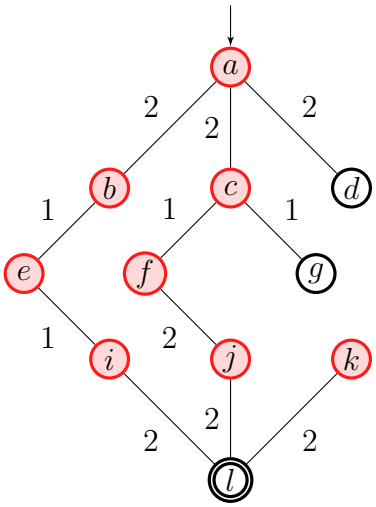
basic block corresponding to vertex v contains a critical instruction that needs to be buffered in memory through additional instrumentation. Thus, v is added to U (Line 7). Figure 4.4(c) shows the graph that results upon collapsing all vertices in the vertex cover shown in Figure 4.4(b).

- Lines 3-8 are repeated until the minimum arc weight(s) of G are greater than or equal to SP. In other words, the heuristic will repeat until the effective graph's longest sampling period is at least SP. This termination condition is expressed in the while-loop condition on Line 3. One possible solution that this heuristic may return after processing the graph in Figure 4.2(a) is shown in Figure 4.4(d).

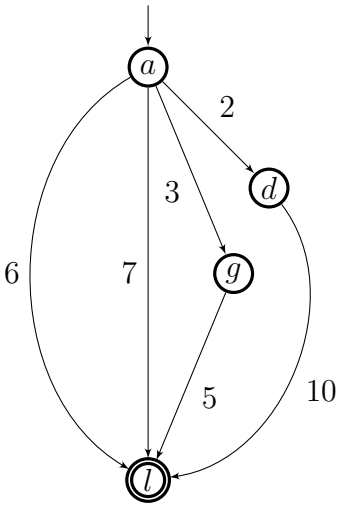
- If the graph cannot be collapsed further (i.e., all vertices are collapsed), then the graph's structure will not permit increasing the sampling period to SP and the algorithm declares failure.



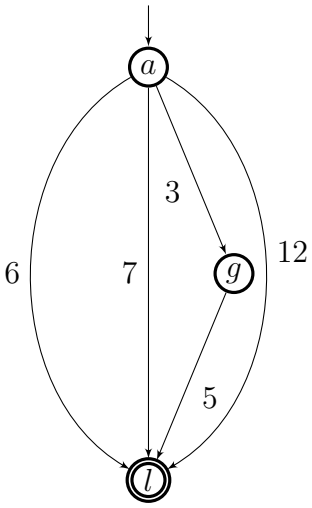
(a) Generate vertex cover subgraph.



(b) Approximate minimum vertex cover.



(c) Collapse all nodes in vertex cover.



(d) Solution.

Figure 4.4: Illustrations of Heuristic 2.

4.3.3 Heuristic 3: Genetic Algorithm

The final heuristic that was considered for approximating a (near-)optimal instrumentation scheme is a genetic algorithm (GA). Genetic algorithms are search heuristics that adopt a process that is evolutionary-like. In this context, the heuristic aims at collapsing the fewest number of vertices in a critical control-flow graph G such that the longest sampling period in the resulting graph G' (after collapsing the selected vertices) is equal to or greater than the target (i.e., desired) sampling period SP.

The optimization problem is mapped to the following necessary facets of a genetic algorithm; the subsequent subsections will describe these facets in greater detail:

Chromosomes: A chromosome represents the list of vertices in a critical control-flow graph, G. Each bit in a chromosome maps to a vertex in G. When a bit's value is set to *true*, it represents the condition where the corresponding vertex is selected to be collapsed in G.

Fitness Function: The fitness function of a chromosome is the number of collapsed vertices represented by the chromosome. The fittest chromosome is one that has the fewest number of bits set to *true* for all chromosomes in the population.

Reproduction: Both crossover and mutation are used to generate new generations of chromosomes.

Termination: The genetic algorithm terminates when the upper limit on the number of generations is reached.

4.3.3.1 Chromosomes

Let $G = \langle V, v^0, A, w \rangle$ be a critical control-flow graph. Each chromosome in the genetic model has $|V|$ entries. Each entry is a tuple $\langle \text{vertex id}, \text{SP}_{min}, \text{value} \rangle$ that represents a vertex in G. *Vertex id* is the vertex identifier, *min-SP* is the minimum weight of the incoming and outgoing arcs of the vertex and *value* is a boolean value that indicates if the vertex is collapsed in G. If $value = true$ for a vertex v, then v is collapsed and auxiliary memory is used to temporarily store the event until the monitor flushes the history buffer.

The chromosome's longest sampling period is defined as the longest sampling period of the control-flow graph obtained by collapsing the vertices in the chromosome. In this genetic model, each chromosome's longest sampling period must be always at least SP.

4.3.3.2 Initialization (Seeding)

The number of chromosomes created in each generation is chosen by the user. Let $|\mathcal{G}|$ denote the size of a generation. In the initialization step of this genetic algorithm, $|\mathcal{G}|$ chromosomes are randomly created. To generate these random chromosomes, a set of vertices are arbitrarily collapsed in G so that the chromosomes' longest sampling period is at least SP. The generation of an arbitrary initial chromosome follows these three steps:

1. Find the set of vertices, $U \subseteq V$ where $u_{min\text{-}SP} < SP,\ \forall u \in U$.
2. Randomly choose a vertex $u \in U$ to collapse in G to produce a new control-flow graph $G' = T(G, v)$.
3. Calculate the longest sampling period of G', $LSP_{G'}$. If $LSP_{G'} < SP$, return to step one and operate on G'.

4.3.3.3 Selection/Fitness Function

Since the goal of solving this optimization problem is to transform a graph such that $LSP \geq SP$ with as few collapsed vertices as possible, the chromosome's fitness is characterized by the number of *value* in the chromosome's tuple are set to *true*. Hence, the fitness function of a chromosome *chr* is defined as:

$$\mathcal{F}_{chr} = \sum_{i=1}^{|V|} chr.\text{value} \tag{4.12}$$

$\mathcal{F}_{chr}$ represents the number of nodes collapsed in chr. Consequently, if $\mathcal{F}_{chr}$ is smaller, then the chromosome is more fit.

4.3.3.4 Reproduction

Both genetic operators, *crossover* and *mutation*, are used to evolve the current generation into the next generation. Reproduction/evolution in this genetic algorithm first modifies the chromosomes by crossover. The resulting chromosomes are then mutated to form the next generation of chromosomes as required.

Crossover. New chromosomes are formed by applying *one-point* crossover. Two parents are randomly chosen for crossover. In the crossover, the two parents are split into halves; the two children are produced by swapping one of the two halves between the parents. For each child, if the child chromosome's longest sampling period is at least SP, the child will be added to the set of chromosomes of the next generation; if this condition is not satisfied, the child will be mutated.

Mutation. The mutation process takes the children passed over by the crossover process and mainpulates each child by the following steps:

1. Find the set of vertices, $U \subseteq V$ where $u_{min\text{-}SP} < SP \; \forall u \in U$.
2. Randomly select a vertex $u \in U$ to collapse by $T(G, u)$.
3. Find the set of collapsed vertices S in the chromosome for vertices where $s_{min\text{-}SP} > SP, \; S \subseteq U$.
4. Randomly select a vertex $s \in S$ to expand, meaning that s is restored (as a vertex) to the control-flow graph represented by the child chromosome.
5. Check if the chromosome's longest sampling period, LSP_{chr}, is at least SP. If $LSP_{chr} < SP$, return to step 1 and continue until the chromosome's longest sampling period is at least SP or when the maximum number of mutations allowable is reached.
6. If the resulting chromosome's longest sampling period is at least SP when it reaches step 5, it is added to the next generation.

Crossover and Mutation Limitations. Sometimes the crossover and mutation processes fails to create $|\mathcal{G}|$ chromosomes to populate the next generation. When this occurs, it

means that fewer than $|\mathcal{G}|$ modified chromosomes satisfy the sampling period restriction for chromosomes. In this case, the genetic algorithm chooses the most fit chromosomes from the current generation and adds them to the next generation to create a population of $|\mathcal{G}|$ chromosomes. In the case that duplicates chromosomes appear in this process, it discards the duplicate and randomly creates new chromosomes as described in Section 4.3.3.1.

4.3.3.5 Termination

For this genetic algorithm, termination can occur when one of two conditions are met:

- The highest ranking solution's level of fitness does not change over a fixed number of generations; the number of generations is defined by the user.
- The maximum number of permitted generations is reached. This number is also user-defined. In this case, the chromosome across all generations with the best fitness value is returned.

4.4 Experimental Results

Experiments were conducted to compare the effectiveness of the heuristics to the optimal method of solving the optimization problem. The toolchain that was developed consists of the following:

- The tool CIL [49] was first used to generate the control-flow graph of a given C program.
- Next, tools were developed to transform the control-flow graph into the critical control-flow graph corresponding to the set of critical variables that the monitor needs to observe at run time.
- For optimally solving the problem, a component in the toolchain generated the ILP model (using the method described in [18]) corresponding to the critical control-flow graph. Then, this model is solved using lp_solve [47].

- For solving an instance of the optimization problem using the heuristics, the critical control-flow graph is passed to the respective modules.
- The result of solving the problem using any of the above methods returns the set of instructions and variables in the program that need to be instrumented to store the corresponding critical events in auxiliary memory.
- The program is then instrumented using the returned instrumentation scheme.
- To simulate a time-triggered software monitor, gdb's [50] breakpoint mechanism was used to pause the program's execution at run time while the monitor extracts the necessary information from auxiliary memory and program state; gdb is controlled by a Python script.

Using this toolchain, experiments were conducted on case studies from the embedded software benchmark suite, MiBench [46]. The target sampling period used for all of the case studies presented is $40 \times LSP$, where LSP is the longest sampling period of the program (see Definition 2). All experiments in this section are conducted on a personal computer with a 2.26 GHz Intel Core 2 Duo processor and 6 GB of main memory.

4.4.1 Performance of Heuristics

Table 4.1 compares the performance of the ILP-based solution [18] with the heuristics presented in Section 4.3 for various programs from MiBench. The first column in the table shows the size of the critical control-flow graph of programs in terms of the number of vertices. With each approach, the time spent to solve the optimization problem (in seconds) was logged. The performance of the heuristics are characterized by the suboptimal factor (SOF). SOF is defined as $\frac{sol}{opt}$, where sol and opt are the number of vertices requiring instrumentation returned by a heuristic and the ILP-based solution (i.e., the optimal solution), respectively.

Clearly from Table 4.1, all three heuristic algorithms perform substantially faster than solving for the exact optimal solution. On average, Heuristic 1, Heuristic 2, and the genetic algorithm yield in speedups of 200 000, 7 000, and 9, respectively, where the speedup is defined as the ratio between the execution time required to solve the ILP problem and

Table 4.1: Performance of different optimization techniques.

	CFG	*ILP*		*Heuristic 1 (Greedy)*		*Heuristic 2 (VC)*		*Genetic Algorithm*	
	Size($\|V\|$)	**time (s)**	**SOF**	**time (s)**	**SOF**	**time (s)**	**SOF**	**time (s)**	**SOF**
Blowfish	177	5316	–	0.0363	7.8	0.8875	8	383	2.5
CRC	13	0.35	–	0.0002	3.5	0.0852	3	0.254	1.5
Dijkstra	48	1808	–	0.0064	1.2	0.1400	1.2	116	1.7
FFT	47	269	–	0.0042	1.7	0.1737	1.8	74	1.1
Patricia	49	2084	–	0.0054	1.4	0.1369	1.6	140	1.5
Rijndael	70	3096	–	0.0060	1.6	0.2557	2.1	370	1.9
SHA	40	124	–	0.0039	2.2	0.1545	2.2	46	1.3
Susan	20 259	∞	–	3 181	N/A	26 211	N/A	923	N/A

the time required to generate an approximate solution using one of the heuristics. The execution times of Heuristic 2 are based on running Approximate-Vertex-Cover 500 times to cope with the randomized vertex cover algorithm (see Line 4 in Heuristic 4.6). Table 4.1 shows that for large programs, such as Susan, solving for the optimal solution becomes infeasible because the size of the problem is too large to cope with. All three heuristics, however, are able to generate some approximate solution that can be used to instrument the program for time-triggered runtime verification.

In general, the genetic algorithm produces results that are closer to the optimal solution than Heuristic 1 and Heuristic 2. The spread of the SOFs for the conducted experiments is much smaller for the genetic algorithm. For the conducted experiments, the worst SOF for the genetic algorithm is 2.5 (i.e., for Blowfish), which indicates that this solution will instrument at 2.5 times more locations in the program than the optimal solution. With the exception of Blowfish, Heuristic 1 and Heuristic 2 also perform well; the SOF in Table 4.1 ranges from 1.2 to 3.5. Based on the conducted experiments, it cannot be concluded that the performance of Heuristic 1 and Heuristic 2 suffers as the size of the problem increases. The SOFs for Susan were not reported in Table 4.1, but the results from the three heuristics were recorded. Heuristic 1 and Heuristic 2 indicate that the target sampling period may be satisfied by collapsing 104 and 180 vertices, respectively, while the genetic algorithm produced a solution that requires 222 vertices to be collapsed. The SOFs for Dijkstra also indicate an anomaly in the overall trend perceived in Table 4.1. Therefore, the performance of the heuristics likely depends on the structure of the critical control-flow graph. For Susan,

the number of vertices being collapsed is approximately 0.5% to 1% of $|V|$, which indicates that the instrumentation overhead should be small.

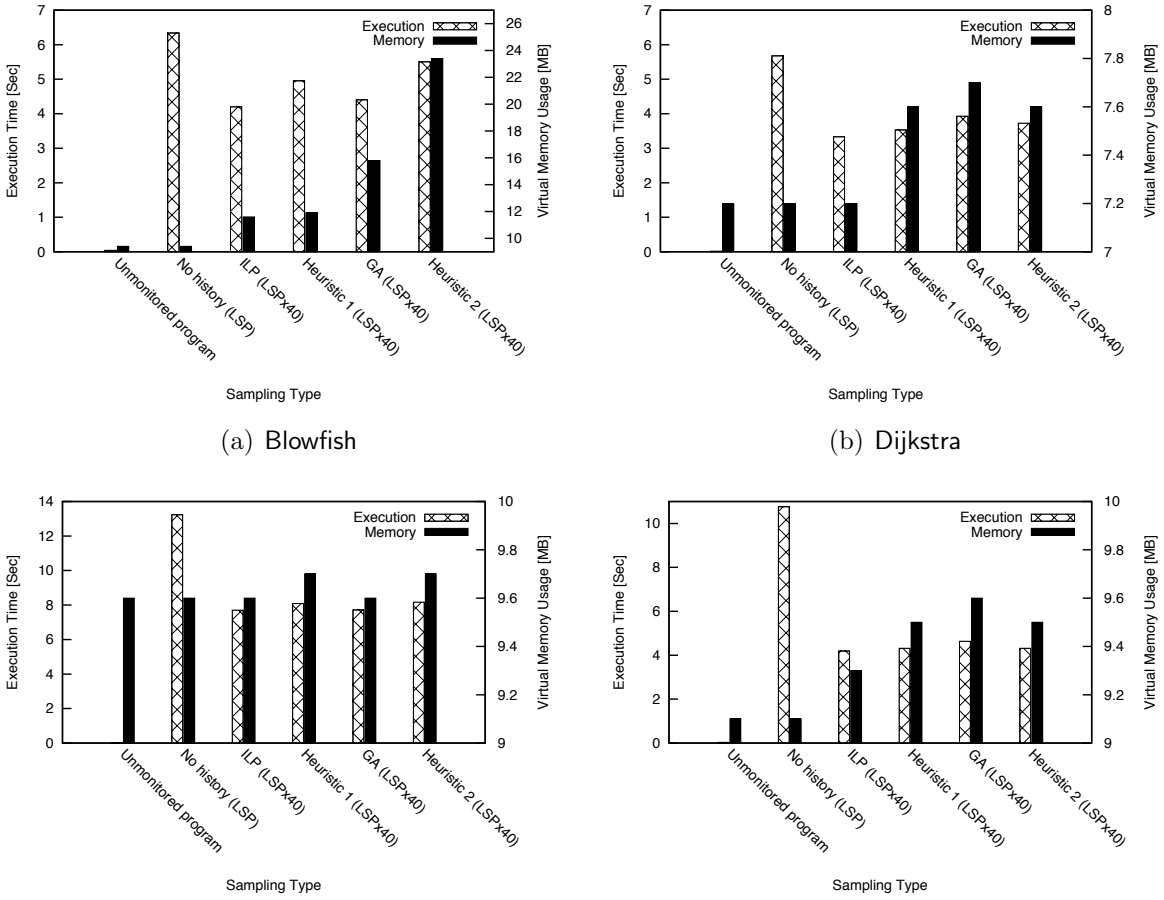
4.4.2 Analysis of Instrumentation Overhead

The execution times and memory usage of the instrumented benchmark programs were also collected during experimentation. Figure 4.5 shows the execution times and memory usage of four of the eight benchmark programs used in the experiments. Each plot in Figure 4.5 contains the total execution times and memory usage for the unmonitored program, the program monitored with a sampling period of LSP, and the program monitored at $40 \times LSP$ with the inserted instrumentation points indicated by the optimal and heuristic solutions. The benchmark program results not shown in Figure 4.5 exhibit similar trends as Figure 4.5(c).

All instrumented benchmark programs with no history always run slower than the instrumented programs that support a target sampling period of $40 \times LSP$ as illustrated in Figure 4.5. This is expected because time-triggered runtime monitoring without history requires the monitor to sample at higher frequencies to preserve monitoring correctness. Monitor invocations are much more expensive than buffering events in auxiliary memory.

Figure 4.5 also shows that the variation of the execution times of the instrumented benchmark programs based on the optimal and heuristic solutions (i.e., optimal (ILP), Heuristic 1, Heuristic 2 and genetic algorithm) are negligible. Therefore, using suboptimal instrumentation schemes does not significantly impact the execution time of the program in comparison to the optimally instrumented program.

As shown in Figure 4.5, utilizing the instrumentation schemes returned by solving the ILP or running the heuristics result in an increase in the memory usage during program execution in comparison to both the un-monitored program and when the program is monitored in a time-triggered fashion without the use of auxiliary memory to buffer events. This is expected because to increase the sampling period of the monitor, some critical events must be retained in auxiliary memory to ensure that the program can be correctly verified at run time. With the exception of Blowfish, the memory usage increase is negligible for the benchmark programs.



(a) Blowfish (b) Dijkstra

(c) FFT (d) Rijndael

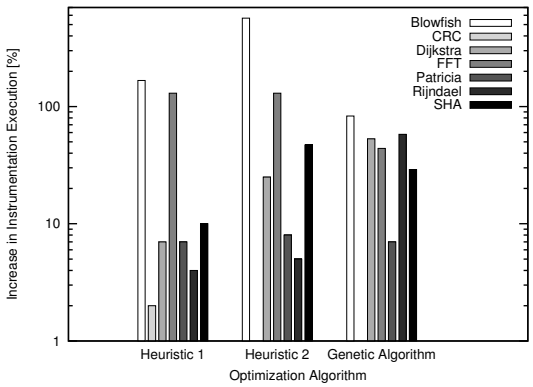
Figure 4.5: The impact of different instrumentation schemes on memory usage and total execution time.

Using the instrumentation schemes generated by the heuristics, the increase in memory usage is negligible during program execution with respect to the optimally instrumented program, except for Blowfish. The variation of memory usage for all benchmark programs except for Blowfish generally spans from 0 MB to 0.1 MB. Even though the memory usage of Blowfish instrumented with the schemes produced by Heuristic 2 and the genetic algorithm is relatively larger than the optimal scheme, an increase of 15 MB of virtual memory is still negligible to the amount of memory that is generally available machines used to verify

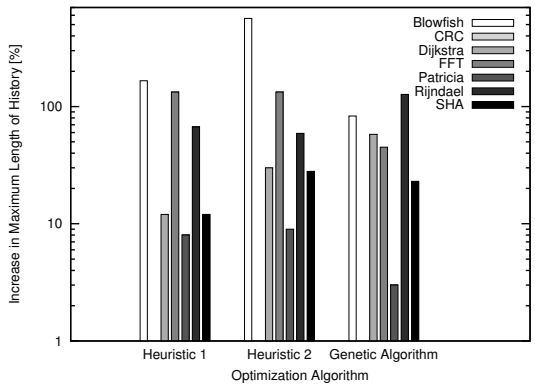
software programs. From the experimental data collected for the three heuristics, no generalizations can be made. In other words, none of the heuristics generally yield the best instrumentation scheme. The best sub-optimal instrumentation scheme depends on both the input control-flow graph and the heuristic that is used to produce an approximate solution.

Figure 4.6 shows the percentage increase in the number of instrumentation instructions executed and the percentage increase in the maximum size of history between two consecutive samples with respect to the optimally instrumented benchmark programs. Note that logarithmic scales are used in the charts in Figure 4.6. Observe that Susan is not shown in the figure because the ILP model for the program could not be solved using the machine the experiments were conducted on. Blowfish performed the poorest with respect to the two measures when the instrumentation schemes generated by Heuristic 1 and Heuristic 2 were used. In most cases, the percentage increase in the number of instrumentation instructions that are executed and the maximize size of history are below 50% if the two largest percentages are removed from each set. If a few more outliers are removed, then most of the percentage increases for both measures will be below 20%. In addition, observe that the percentage increase in the number of instrumentation instructions executed is proportional to the increase in the maximum size of the history between two consecutive samples. This implies that the extra instrumentation instructions (compared to the optimal solution) are evenly distributed among sampling points.

Recall that the collapsed vertices during the transformation $IT(CFG, v)$ (see Section 3.3.3) determine the instrumentation instructions added to the program under inspection. These instructions in turn store changes in critical variables to the history. Although one may argue that auxiliary memory usage at run time must be directly related to the number of collapsed vertices (i.e., instrumentation instructions), this is not necessarily true. This is because the number of added instrumentation instructions differs in different execution paths. In the extreme case, one execution path may include no instrumentation instructions and another path may include all such instructions returned by solving the problem instance. In this case, the first path will build no history and the second will consume the maximum possible auxiliary memory. This observation also holds in the conducted analyses on other types of overhead and the total execution time. This is why the genetic algorithm produced the best instrumentation scheme for the Blowfish



(a) Increase in the number of execution of instrumentation instructions.



(b) Increase in the maximum size of history between two samples.

Figure 4.6: The impact of sub-optimal solutions on execution of instructions to build history and its maximum size.

benchmark (see Table 4.1), but the benchmark used substantially more memory than the greedy heuristic at run time (see Figure 4.5(a)). This is also explains why the amount of auxiliary memory used by a monitored program is not proportional to the number of instrumented critical instructions (see Figure 4.6).

> *The experimental results empirically demonstrate that the NP-completeness of the optimization problem is likely not an obstacle when applying time-triggered runtime verification in practice.*

4.5 Concluding Remarks

In this chapter, three efficient (polynomial-time and space) algorithms were presented that address the NP-complete problem of optimizing the instrumentation of programs in the context of time-triggered runtime verification [18]. The need for instrumentation is required to record events between two consecutive samples at run time so that monitoring correctness is preserved. The presented algorithms were inspired by different techniques for determining near-optimal instrumentation schemes, which include using a greedy approach, determining the minimum vertex cover, and biological evolution. A total of eight programs from MiBench [46] were used to rigorously benchmark the proposed heuristics. The results show that the solutions returned by all three algorithms led to negligible increase in instrumentation runtime overhead, total runtime memory usage, and total execution time of monitored program. Moreover, the genetic algorithm yielded in more consistent results compared to the other two heuristics. In summary, the empirical results illustrate that the NP-completeness of the optimization problem is likely not an obstacle when applying time-triggered runtime verification in practice.

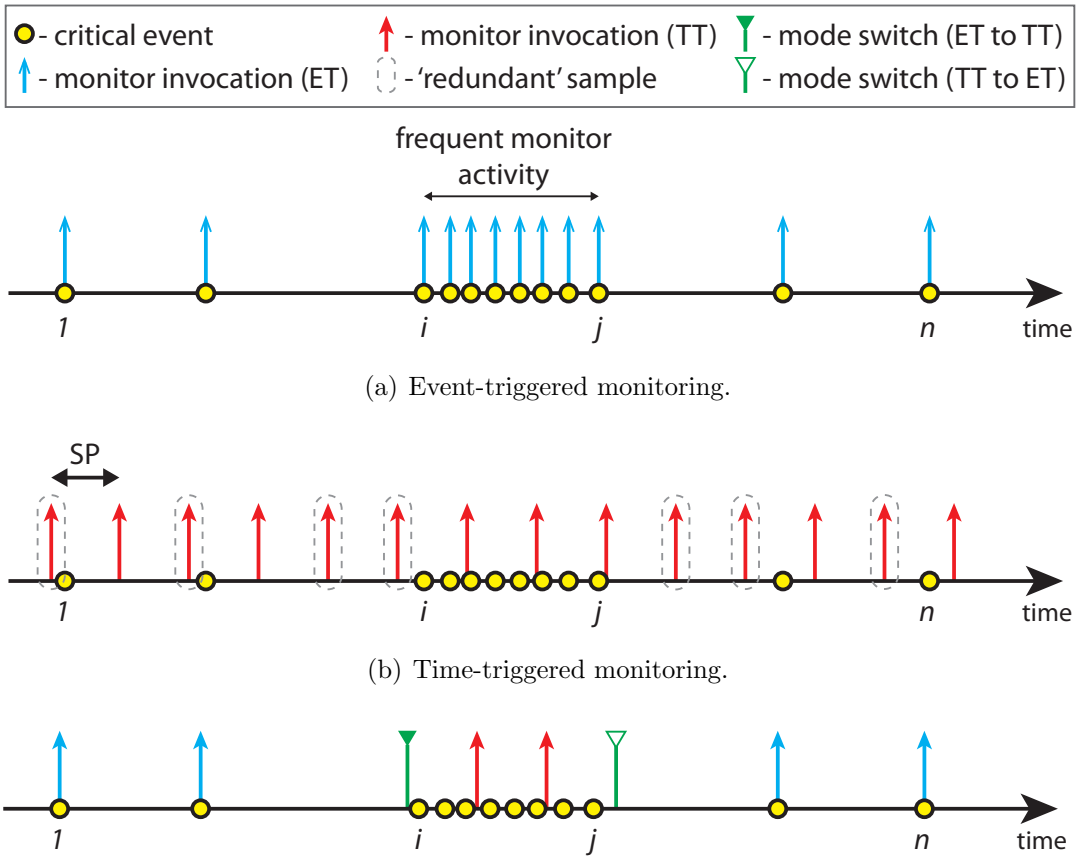
Chapter 5

Hybrid Runtime Monitoring

5.1 Introduction

The main challenge in augmenting a system with runtime verification is to contain its runtime *overhead*. Most monitoring approaches in the literature are *event-triggered* (ET), where the occurrence of a new critical event (e.g., change of value of a variable) triggers the monitor to verify a set of logical properties. Consider the timing diagrams in Figure 5.1(a), where the dots 1 through n along the timeline represent the critical events that occur for an execution trace of the program under scrutiny at run time. The calls to the monitor are added as instrumentation instructions in the program. As shown in the figure, there is a burst of events in this execution trace from event i to event j. The burst of critical events that occur from i to j leads to frequent monitor invocations and activity, which causes high execution overhead and unpredictability of the program's timing behaviour.

Navabpour et al. [18] introduced an approach that uses *time-triggered* (TT) monitoring for runtime verification. Time-triggered runtime verification makes the runtime monitoring overhead controllable and predictable, and makes monitoring tasks schedulable. In this method, a monitor samples the program at periodic time intervals. This time interval, known as the *sampling period* (SP), should guarantee that the monitor is capable of observing all critical events. Time-triggered monitoring is especially desirable for designing real-time systems, where time predictability and scheduling plays a central role in system



(a) Event-triggered monitoring.

(b) Time-triggered monitoring.

(c) Hybrid monitoring.

Figure 5.1: Comparing different methods of monitoring.

correctness. Figure 5.1(b) shows the interactions that occur between the program and a TT monitor. Navabpour et al. [18] present a technique to decrease the overhead of time-triggered monitor, which involves buffering a selected subset of critical events in auxiliary memory. With buffering enabled, the monitor can effectively sample at a lower sampling frequency, thereby, reduce the overall monitoring overhead. The monitor preserves correctness because it is configured to read the buffered critical events when it samples the program in addition to the current program state so that property evaluation occurs for all critical program state changes [18]. From Figure 5.1(b), it is evident that the monitoring activity between events i and j is significantly less than what an event-triggered

monitor would require. Navabpour et al. [18] observed that in some cases, time-triggered runtime verification (TTRV) may also reduce the cumulative runtime overhead effectively. However, time-triggered runtime verification does not guarantee a reduction in monitoring overhead. Consider the example shown in Figure 5.1 and the target sampling period adopted for Figure 5.1(b): there are some 'redundant' samples that the monitor takes. A *'redundant' sample* is an invocation of the monitor, where the monitor does not have any critical events to process. In other words, the monitor does not do any meaningful work. The dashed ovals in Figure 5.1(b) mark the redundant samples in this example. In execution traces where the critical events are sparse and sporadic, time-triggered monitoring incurs a lot of unnecessary overhead.

From Figures 5.1(a) and 5.1(b), it is evident that both event- and time-triggered monitoring techniques have both advantages and disadvantages with respect to the monitor's execution overhead. Event-triggered monitoring tends to be advantageous in situations where critical events occur sparsely because the monitor is active only when the program encounters a critical event; time-triggered monitoring tends to be better when there are many critical events to process within a short time frame.

With this motivation, this chapter proposes a novel technique called *hybrid* runtime verification (HyRV) that exploits the benefits of both ETRV and TTRV to reduce the runtime overhead. The goal of this technique is to supply a program under scrutiny with a more efficient monitor that supports both ET and TT modes of operation. This 'hybrid' monitor may switch from one mode to another at run time depending upon the current execution path. HyRV automatically obtains the locations to *switch modes* in the program by solving a doubly exponential optimization problem; this method accounts for all monitoring and switching costs in terms of execution time overhead. The main challenge in formulating the optimization problem is threefold:

1. Determining the precise timing behaviour of the program under inspection,
2. Identifying the overhead of all required activities for implementing an ET or TT monitor (e.g., cost of monitoring mode switching, sampling, monitor invocation),
3. Identifying the execution subpaths that are likely to be suitable for ET and TT monitoring modes.

The solution to the problem is an instrumentation scheme for a program that may switch monitoring modes at runtime. For instance, in Figure 5.1(c), the reduction in monitoring activity will likely reduce the overall monitoring execution overhead. Obviously, using hybrid monitoring will incur overhead costs in performing mode switches. In this example, a mode switch occurs right before i and right after j to switch from ET to TT and TT to ET monitoring modes, respectively.

A fully implemented toolchain of this technique leverages static analysis techniques and integer linear programming (ILP) to solve the optimization problem. The inputs to the toolchain are a C program and a set of variables to monitor. The toolchain outputs the program source code augmented with the instrumentation scheme that may toggle the monitoring mode at runtime to reduce the monitoring overhead. The experiments conducted on a benchmark suite for real-time embedded programs strongly validate the effectiveness of this technique.

Organization. The rest of the chapter is organized as follows. Section 5.2 introduces the HyRV optimization problem. Experimental results and analyses are presented in Section 5.3. Finally, in Section 5.4 offers some concluding remarks.

5.2 Hybrid Runtime Verification

The goal of hybrid monitoring in runtime verification is to select the monitoring scheme that minimizes the expected total overhead incurred from executing the monitor. Given an execution path in advance, the optimal solution with minimum overhead is already a complex problem. Therefore, for any general control-flow graph, the problem of finding the optimal solution is is even more difficult, especially when loops are present. In the case that the loops are unbounded, this problem is unsolvable. If all lower and upper loop bounds are given, the problem is still likely not in NP with respect to the size of the graph.

The optimization problem is likely not in NP because the verification of a certificate that includes an instrumentation scheme and an integer denoting maximum allowable monitoring overhead requires enumerating all execution paths in the worst case, which is exponential in the size of the problem's input. Hence, the complexity of this optimization

problem is likely to be at least doubly exponential (one for execution path explosion and one for solving an integer program).

In order to tackle the high computational complexity of the problem, a heuristic is introduced that aims to return a monitoring scheme where the monitoring overhead is equal or better (i.e. lower) than monitoring exclusively in either ET or TT mode. This heuristic involves solving an instance of the *integer linear programming* problem. The sub-optimality stems from how the program is subdivided into subpaths to estimate the monitoring cost incurred by sampling. The rest of this section is organized as follows. First, in Section 5.2.1, the monitoring overhead cost incurred at run time is divided into different types/categories. Then, Section 5.2.2 presents a transformation of the optimization problem (for reducing the runtime overhead of monitoring by integrating event- and time-triggered techniques) to an ILP model.

5.2.1 Overhead Runtime Costs

HyRV classifies the *overhead costs* incurred from monitoring as follows:

- C_{event}: the cost incurred to handle critical events (i.e., in TT mode, this includes the costs of writing and retrieving the history, and the property evaluation; in ET mode, this includes calling the monitor and the property evaluation),
- C_{switch}: the cost incurred from switching between ET and TT modes and vice versa, and
- C_{sample}: the cost incurred from sampling (i.e. preempting and resuming the program under scrutiny) in TT mode.

The cost estimates are derived in terms of the best-case execution time of the corresponding instructions. In particular, these costs are calculated in the same fashion as determining the arc weights of a control-flow graph (see Definition 1). The objective function with respect to these costs is:

$$\min \quad (C_{event} + C_{switch} + C_{sample}) \tag{5.1}$$

For the rest of this chapter, let $CFG_P = \langle V, v^0, A, w, \mathcal{F} \rangle$ be a control-flow graph corresponding to a program P. Each vertex corresponds to a basic block containing one and only one critical instruction. The definitions of V, v^0, A, and w correspond to the ones described in Defintion 1 (see Figure 3.2(b) for an example). $\mathcal{F}$ is a function $\mathcal{F} : (u, v) \to \mathbb{N}$, $(u, v) \in A$, $u, v \in V$, that defines the expected number of times P will execute the basic block corresponding to v immediately after executing the basic block corresponding to u. Figure 5.2 illustrates a *CFG*, where the critical vertices are highlighted, and the set of numerical values within parentheses defines the function, $\mathcal{F}(u, v)$. The function $\mathcal{F}$ can be evaluated for a program using standard techniques such as profiling and symbolic execution; if these are infeasible, the user may define this function or specify a uniform function over the input domain.

To derive expressions for the overhead costs defined in the objective function, the cost of monitoring is broken down into five *elementary cost* values, which capture the costs incurred from performing specific interactions between the program and the monitor:

- c_{ET}: cost of invoking monitor to check a single critical event in ET mode
- c_{hist}: cost of saving a critical event into the history buffer in TT mode
- c_{TT}: cost of processing the history buffer at a sample in TT mode
- $c_{E \to T}$: cost of a switch from ET mode to TT mode
- $c_{T \to E}$: cost of a switch from TT mode to ET mode

5.2.2 Utilizing Integer Linear Programming as a Heuristic

The ILP problem is of the form:

$$\begin{cases} \text{Minimize} & c.\mathbf{z} \\ \\ \text{Subject to} & A.\mathbf{z} \geq \mathbf{b} \end{cases} \tag{5.2}$$

where A (a rational $m \times n$ matrix), c (a rational n-vector) and $\mathbf{b}$ (a rational m-vector) are given, and $\mathbf{z}$ is an n-vector of integers to be determined. In other words, solving this

problem involves finding the minimum of a linear function over a feasible set defined by a finite number of linear constraints. It can be shown that a problem with linear equalities and inequalities can always be put in the above form, implying that this formulation is more general than it might look.

The remainder of this section describes the mapping of the optimization objective (Equation 5.1) stated in Section 5.2.1 to ILP.

5.2.2.1 ILP Variables

Two binary variables x_v and y_v are defined for each $v \in V$ in CFG_P. If $x_v = 1$, then the monitor will operate in ET mode whenever the corresponding basic block executes, and if $y_v = 1$, the monitor will operate in TT mode whenever the program is executing the basic block. The following constraint expresses the mutual exclusivity of monitoring modes for $v \in V$:

$$x_v + y_v = 1 \tag{5.3}$$

5.2.2.2 Constraint of Handling Critical Events

Equation 5.4 expresses the cost incurred at each critical event in P:

$$C_{event} = \sum_{v \in V_c} \sum_{\substack{(u,v) \in A \\ u \in V}} [\mathcal{F}(u,v) \cdot (c_{ET} \cdot x_v + c_{hist} \cdot y_v)] \tag{5.4}$$

where $V_c \subseteq V$ is the set of nodes that correspond to the critical basic blocks in CFG_P. The number of times that P is expected to transit from the set of nodes u to v, where $(u, v) \in A$, determines the expected number of times that the basic block corresponding to v will execute. Equation 5.3 and Equation 5.4 guarantee that the cost incurred for the critical event in v is exclusively c_{ET} or c_{TT} if the monitor is operating in ET or TT mode at that point in the program, respectively.

5.2.2.3 Constraints of Switching Monitoring Mode

The following equation expresses the cost of switching between ET and TT modes:

$$C_{switch} = \sum_{\substack{(v_1,v_2)\in A \\ v_1,v_2\in V}} [\mathcal{F}(v_1,v_2)\cdot(c_{E\to T}\cdot x_{v_1}\cdot y_{v_2} + c_{T\to E}\cdot y_{v_1}\cdot x_{v_2})] \tag{5.5}$$

There exists a mode switch between basic blocks v_1 and v_2 when $x_{v_1} = y_{v_2} = 1$ or $y_{v_1} = x_{v_2} = 1$. The former case implies that the monitor switches from ET mode to TT mode and the latter case implies that the monitor switches from TT mode to ET mode. Note that a switch may occur between any two connected vertices; solving the optimization problem ensures that the switches are optimal. Equation 5.5 is non-linear; to linearize this expression, let p_{v_1,v_2}, q_{v_1,v_2}, r_{v_1,v_2}, and s_{v_1,v_2} be all binary variables and rewrite Equation 5.5 as:

$$C_{switch} = \sum_{\substack{(v_1,v_2)\in A \\ v_1,v_2\in V}} [\mathcal{F}(v_1,v_2)\cdot(c_{E\to T}\cdot p_{v_1,v_2} + c_{T\to E}\cdot q_{v_1,v_2})] \tag{5.6}$$

subject to:

$$x_{v_1} + y_{v_2} + 2r_{v_1,v_2} \geq 2 \tag{5.7}$$

$$p_{v_1,v_2} + r_{v_1,v_2} = 1 \tag{5.8}$$

$$x_{v_1} + y_{v_2} - 2(1 - r_{v_1,v_2}) < 2 \tag{5.9}$$

$$y_{v_1} + x_{v_2} + 2s_{v_1,v_2} \geq 2 \tag{5.10}$$

$$q_{v_1,v_2} + s_{v_1,v_2} = 1 \tag{5.11}$$

$$y_{v_1} + x_{v_2} - 2(1 - s_{v_1,v_2}) < 2 \tag{5.12}$$

Equations 5.7 through 5.9 ensure that if $x_{v_1} = y_{v_2} = 1$, then $p_{v_1,v_2} = 1$, i.e., a switch from ET to TT mode occurs between v_1 and v_2 and incurs the cost, $c_{E\to T}$. Similarly, the constraints reflected in Equations 5.10 through 5.12 ensure that if there exists a switch from TT to ET mode, then $q_{v_1,v_2} = 1$ and the switch will add the cost, $c_{T\to E}$.

5.2.2.4 Constraints of Sampling Cost in TT Mode

Finally, Equation 5.13 captures the cost incurred from the sampling the monitor does in TT mode:

$$C_{sample} = \sum_{\pi \in \Pi'(CFG_P)} (c_{TT} \cdot \mathcal{F}_\pi \cdot N_{samp_\pi}) \tag{5.13}$$

where $\Pi'(CFG_P)$ denotes the set of all subpaths in CFG_P that satisfy the following five conditions if $\pi = v_1 \to v_2 \to \cdots \to v_k$, $\pi \in \Pi'(CFG_P)$:

1. $k \geq 2$
2. $indegree(v_i) = outdegree(v_i) = 1,\ 2 \leq i \leq k-1$
3. $indegree(v_1) \neq 1 \vee outdegree(v_1) \neq 1$
4. $indegree(v_k) \neq 1 \vee outdegree(v_k) \neq 1$
5. for each $(v_i, v_j) \in A$, (v_i, v_j) appears in exactly one $\pi \in \Pi'(CFG)$

In other words, $\Pi'(CFG_P)$ returns the set of longest simple linear subpaths within CFG_P. For example, $\Pi'(CFG_P)$ of the control-flow graph shown in Figure 5.2 is:

$$\begin{aligned} \Pi'(CFG) = \{&\langle a \to b \to c \to d \rangle, \\ &\langle d \to e \to f \rangle, \\ &\langle f \to d \rangle, \\ &\langle d \to g \to h \to f \rangle, \\ &\langle f \to i \to j \rangle\} \end{aligned} \tag{5.14}$$

Moreover, in Equation 5.13, $\mathcal{F}_\pi$ is the expected number of times that π will execute at run time. $\mathcal{F}_\pi = \mathcal{F}(v_i, v_j)$, where (v_i, v_j) is any arc on path π. N_{samp_π} expresses the number of samples that the monitor takes when P executes π once:

$$N_{samp_\pi} = \sum_{\substack{\gamma = \langle v_i \to \ldots \to v_j \rangle, \\ \gamma \in \Gamma_\pi}} \left[\frac{W(\gamma) + c_{hist} \cdot \sum_{m=i}^{j} y_{v_m}}{SP} \cdot x_{v_{i-1}} \cdot x_{v_{j+1}} \cdot \prod_{l=i}^{j} y_{v_l} \right] \tag{5.15}$$

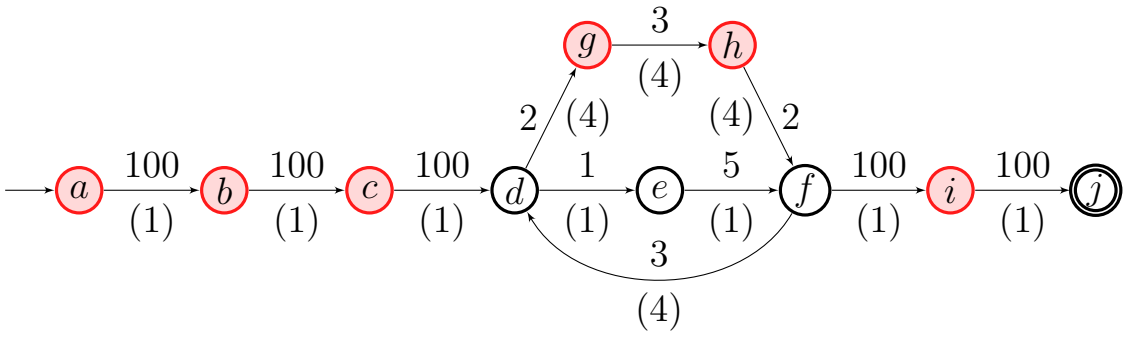


Figure 5.2: CFG used for illustrating ILP model.

where $W(\gamma)$ returns the sum of weights of all arcs on the path $\gamma \in \Gamma_\pi$; v_{i-1} and v_{j+1} denote the immediate predecessor and successor of $v_i, v_j \in V$, respectively; and SP is the target sampling period of the monitor when it is operating in TT mode. If v_{i-1} does not exist in π, $x_{v_{i-1}} = 1$. Similarly, $x_{v_{j+1}} = 1$ if v_{j+1} does not exist in π. Γ_π is the set of enumerated paths in $\pi \in \Pi'(CFG)$ of length 2 or greater. Note that $|\Gamma_\pi| = \Theta\left(|\pi|^2\right)$.

Consider $\Pi'(CFG_P)$ for the control-flow graph shown in Figure 5.2. Then, Γ_π for the subpath $\pi = \langle d \to g \to h \to f \rangle$ is:

$$\begin{aligned} \Gamma_\pi = \{&\langle d \to g \to h \to f \rangle, \\ &\langle d \to g \to h \rangle, \\ &\langle g \to h \to f \rangle, \\ &\langle d \to g \rangle, \\ &\langle g \to h \rangle, \\ &\langle h \to f \rangle\} \end{aligned} \tag{5.16}$$

Considering the example where $\pi = \langle d \to g \to h \to f \rangle$; if $\gamma \in \Gamma_\pi$ starts with d or ends with f, then the terms $x_{v_{i-1}}$ and $x_{v_{i+1}}$ are ignored by substituting them with the value of 1, respectively. N_{samp_π} is linearized by the linearization technique employed for C_{switch} (see Equations 5.7 through 5.12).

5.3 Implementation and Experimental Results

The proposed hybrid monitoring approach was empirically tested and verified by applying this technique on a subset of programs from the SNU Real-time benchmark suite [51] on an embedded development platform. Section 5.3.1 describes the experimental setup and the toolchain. Then, Section 5.3.2 presents and analyzes the results collected from the experiments.

5.3.1 Experimental Setup

Figure 5.3 depicts the constructed toolchain used to generate instrumentation schemes from the model described in Section 5.2. The toolchain generates the program's control-flow graph with estimated execution times of basic blocks by statically analyzing the program's source code with clang and llvm [52]. A custom CodeSurfer [53] plugin was written to determine the location of the critical events the monitor should track at run time based on the set of user-defined critical variables. The model generator takes this information along with the estimated monitoring costs to produce the corresponding model for the program. The toolchain then uses Yices [54], an SMT solver, to identify an approximate solution (i.e., an instrumentation scheme) to the optimization problem described in Section 5.2. Yices is not an out-of-the-box optimization solver, but it is wrapped with additional code that performs a binary search to derive the optimal value of a model; the solutions converged significantly quicker using this particular implementation than the ILP solver, lp_solve. A custom-written clang tool then takes the instrumentation scheme and instruments the program source with the necessary instructions required to monitor the program accordingly.

The monitor and programs were compiled and executed on the Keil uVision simulator that emulates the behaviour of the MCB1700 development platform, which houses an ARM Cortex-M3 processor. Note that the observed execution time across multiple runs of the experiment remains constant because the hardware platform provides accurate timing behaviour of instructions, and in each experiment, the only tasks running were the program under inspection and the monitor. Therefore, the results are presented here without reporting statistical measures.

The performance analysis was conducted on a subset of programs in the SNU-RT [51]

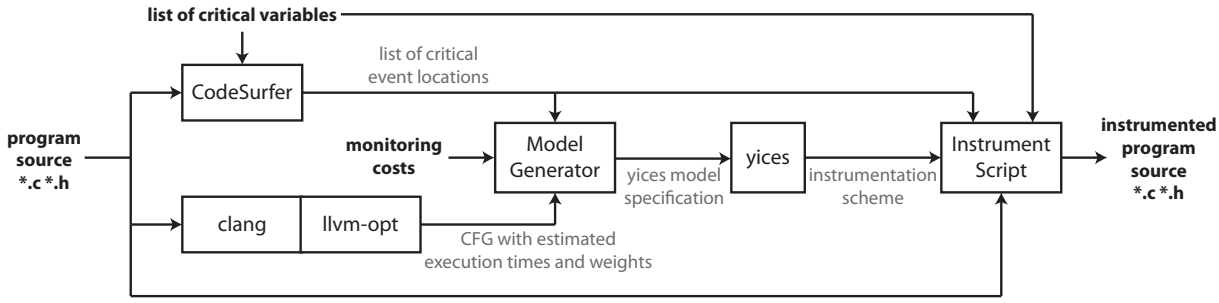


Figure 5.3: HyRV instrumentation toolchain for C applications.

benchmark suite. Six programs were selected from the suite with different sizes: bs, fibcall, insertsort, fir, crc, and matmult. The largest program has 250 lines of code, and the smallest has 20. Two sets of variables were selected for monitoring for each program:

1. A set containing the most frequently changing variables
2. A set containing the program variables that change the most infrequently

Instructions that potentially change the value of these variables form the set of critical instructions monitored in the experiments. For each program, the monitoring overheads were measured using the cost configurations (listed in Table 5.1) and its' associated instrumentation schemes. The cost configurations are dependent on the implementation of the monitor (e.g., running on the same processor, distributed). Table 5.1 summarizes the six cost configurations that were considered for the experiments to demonstrate that the instrumentation schemes may change as a result of the relative differences in the elementary monitoring costs.

5.3.2 Experimental Results

The experimental results are classified based on the generated instrumentation scheme and runtime overhead:

1. The first class consists of cases where the heuristic suggests a hybrid monitor and the monitor indeed significantly outperforms an ET or TT monitor in practice (see Figure 5.4).

Configuration	c_{hist}	c_{ET}	c_{TT}	$c_{E \to T}$	$c_{T \to E}$
1	50	100	100	100	100
2	50	100	100	150	150
3	50	150	150	100	100
4	50	150	150	150	150
5	50	250	250	100	100
6	50	250	250	150	150

Table 5.1: Monitor cost configurations [clock cycles].

2. The second class consists of cases where the heuristic suggests either an ET or TT monitor and the suggested solution indeed outperforms other monitoring modes (see Figure 5.5).

3. The third class consists of cases where the returned solution either exhibits slight improvements over other monitoring modes or slightly underperforms in practice (see Figure 5.6).

For the rest of this section, the results of one program from each of the three classes are used to discuss the experimental results. The three other programs that are not discussed in great depth in the following text exhibit results that fall within one of the three classes.

5.3.2.1 Hybrid Monitor with Significant Improvement

The program representing this class (i.e., crc with CFG of the size 65 vertices and 82 arcs) has two characteristics: it has (1) two tight loops, each containing one critical instruction, and (2) a relatively large initialization function that contains only non-critical instructions. Intuitively, if the program is monitored by an ET monitor, then the tight loops in the program will cause monitor invocations for each iteration. This is an instance where a burst of events creates a large overhead over a short period of time (similar to the timeline in Figure 5.1). In such cases, an ET monitor suffers.

On the contrary, the large initialization function does not contain critical events; hence, a TT monitor would suffer from redundant sampling overhead. With these observations,

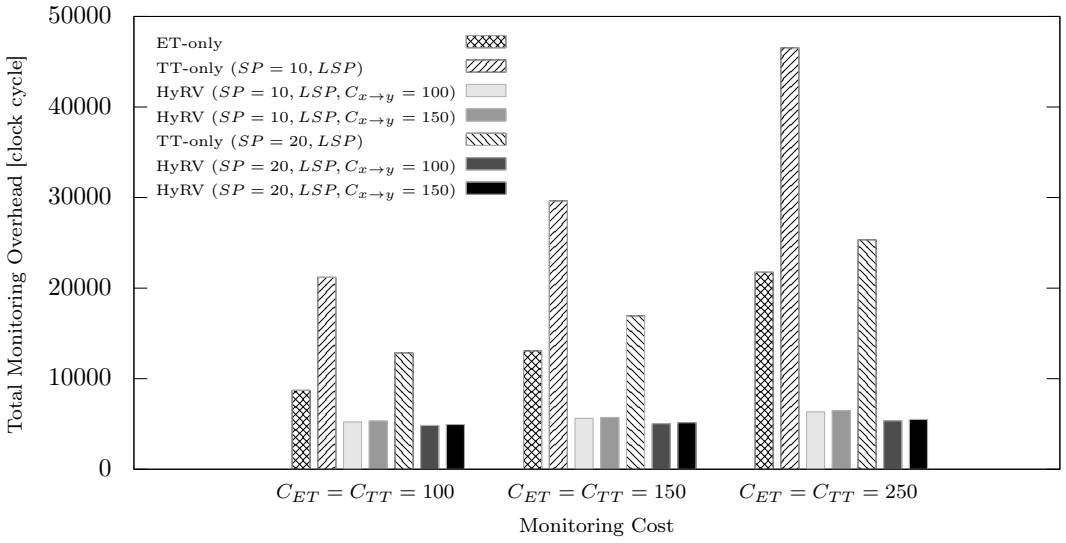


Figure 5.4: Monitoring overhead of crc for three monitoring modes under all cost configurations.

the combination of these two monitoring modes should be able to exploit the benefits of employing a hybrid monitor. The experimental results shown in the plot in Figure 5.4 validates the motivation behind introducing hybrid runtime verification. As can be seen, in all cost configurations, the hybrid monitor incurs significantly less overhead than both the ET monitor and TT monitor operating with the same sampling period. Another interesting observation is that increasing the cost of ET and TT monitor invocations does not greatly increase the overhead of the hybrid monitor. This is because the hybrid monitor only samples when the program reaches its tight loop, which reduces the cost of monitoring frequently occurring critical events by buffering them into memory before sampling; additionally, the monitoring scheme reduces the number of redundant samples by letting the monitor run in ET mode when critical events are infrequent. In such cases, the behaviour of hybrid monitoring is quite robust in this case when the cost of monitor invocation increases.

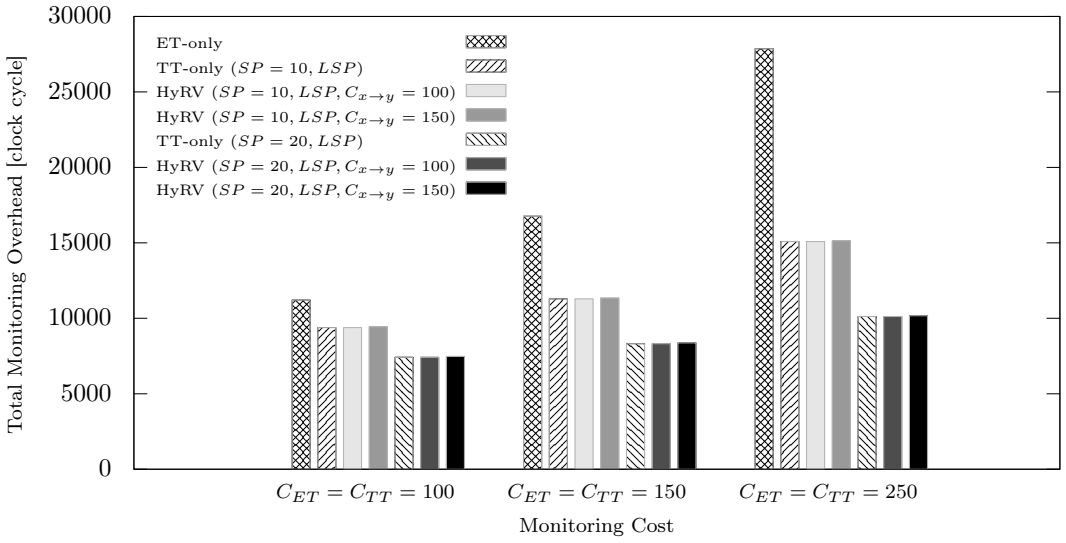


Figure 5.5: Monitoring overhead of insertsort for three monitoring modes under all cost configurations.

5.3.2.2 Time-triggered Monitor with Significant Improvement

The common characteristic of the member programs of this class (i.e., bs, fibcall, insertsort, and matmult) is that the programs have dense and evenly distributed critical instructions throughout its' entirety. This makes the use of TT mode a suitable choice to monitor this class of programs. Figure 5.5 shows the overhead of monitoring insertsort with three monitoring modes (ET-only, TT-only, and hybrid) for all cost configurations. The rest of the programs in this class also exhibit similar monitoring overhead patterns. From Figure 5.5, one can observe that the corresponding solution returned by the heuristic correctly detects the even distribution of events and suggests that the monitor should exclusively operate in TT mode for all cost configurations. Another observation in these experiments is that the number of redundant samples for these programs is either zero or close to zero. The low number of redundant samples again validates the choice of monitoring these programs by having the the monitor operate in time-triggered mode.

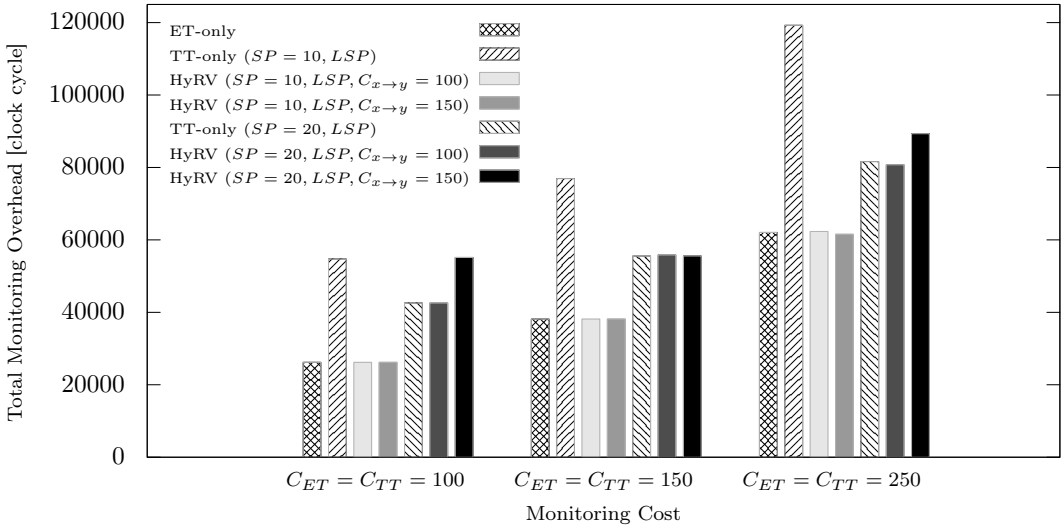


Figure 5.6: Monitoring overhead of fir for three monitoring modes under all cost configurations.

5.3.2.3 Hybrid Monitor with Mixed Behaviour

The program representing this class (i.e. fir with CFG of the size 24 vertices and 27 arcs) does not clearly belong to the previous two classes. The number of redundant samples for this program reduces by a factor of six as the sampling period increases from $10 \times LSP$ to $20 \times LSP$. This brings the overheads of ET and TT modes to a comparable level and makes the ILP model outcome highly sensitive to the elementary monitoring costs. Figure 5.6 shows the monitoring overhead of fir under the three modes of monitoring for different cost configurations. One can observe that when the sampling period is $10 \times LSP$, the model correctly chooses ET mode for the monitoring schemes. However, if we set the sampling period to $20 \times LSP$, then the ILP model provides a hybrid solution for all three cost configurations. The proposed hybrid solutions have slightly higher overheads in comparison to ET mode, but perform as good as TT mode except for two cases in practice. The reason for this discrepancy lies in the fact that our approach is a heuristic algorithm and, hence, finds suboptimal solutions in some cases. Note, however, that this discrepancy does not dramatically affect the usefulness of our approach.

5.4 Concluding Remarks

This chapter presented an approach that combines two techniques in the literature of runtime verification to reduce the overhead: (1) the traditional event-triggered (ET) approach, and (2) the time-triggered (TT) method for real-time systems. Hybrid runtime verification is a technique that can effectively exploit the advantages of both approaches to reduce the overhead of runtime monitoring. To this end, an optimization problem that takes into account the cost of different monitoring interactions (i.e., monitor invocation in ET, sampling and building history in TT, and mode switching) was formulated. In particular, the objective of the problem is to minimize the cumulative overhead in all execution paths using the aforementioned costs.

Since solving the general problem can be computationally unsolvable (e.g., due to the existence of unbounded loops) or expensive (i.e., not in NP), an integer linear programming heuristic was proposed to find suboptimal but effective solutions to the problem by transforming it into an instance of the integer linear programming problem. The experimental results on a subset of the SNU-RT benchmark suite showed that hybrid monitoring can effectively reduce the runtime monitoring overhead; in cases where hybrid schemes are not beneficial, the heuristic can determine, with relatively good accuracy, whether exclusively an ET or TT monitor would yield in lower overhead given a target sampling period.

Chapter 6

Conclusions

Chapters 4 and 5 presented new techniques that may be applied to the field of runtime verification. In Chapter 4, three different heuristics were developed to address the intractability of solving the problem of minimizing the number of events that require buffering in time-triggered runtime verification to preserve correct program state reconstruction [18]. The three heuristics (greedy, vertex-cover based, and genetic) preserve correctness in program state reconstruction and significantly improve the efficiency of generating a feasible instrumentation scheme for time-triggered runtime verification. This efficiency comes with at the cost of sub-optimally instrumenting the program; however, experimental results and analyses show that sacrificing optimality in instrumentation still results in similar run time performance with respect to the optimal scheme.

Chapter 5 introduced the concept of hybrid runtime verification, where the combination of event- and time-triggered monitoring techniques are used to reduce the cost of runtime monitoring. Event-triggered monitoring is advantageous when critical events are sporadic and time-triggered monitoring is more efficient when critical events are bursty. The technique presented in Chapter 5 statically analyzes a program and aims to determine a monitoring scheme that yields in near-optimal overhead. Experimental results and analyses in Section 5.3 show that hybrid runtime verification is feasible and in some cases can dramatically reduce the observed monitoring overhead at run time.

Chapter 7

Future Work

The runtime monitoring techniques presented in this thesis share one common limitation: the heuristics/algorithms that solve the respective optimization problems heavily rely on static analysis. Particularly, the presented algorithms are very sensitive to the input execution time estimates and the frequency distribution of the paths the program under scrutiny takes at run time. Future work in these areas of runtime verification should explore more sophisticated mechanisms and algorithms to address this issue.

In the experiments that were conducted for reducing the memory utilization of the monitor for time-triggered runtime verification (see Section 4.4), it was evident that some of the solutions did not strongly correlate to the measured memory utilization of the monitor because no weights were applied to the control-flow graph to give more importance to basic blocks that are more likely to execute at run time. The heuristics and ILP model indirectly assumes that the execution of the basic blocks are equally likely at run time. In hybrid runtime monitoring, the formulation introduced the notion of frequencies/weights to give frequently executing basic blocks higher priority with respect to optimizing the monitor's execution overhead with respect to time. However, this requires extensive program profiling for good accuracy, which may be infeasible for larger programs or a large range of system input and disturbances.

In a recent paper published by Navabpour et al. [20], they proposed a method that uses symbolic execution [45, 55] to alter the monitor's sampling period at run time by evaluating the program state on-the-fly. For the SNU-RT benchmark suite [51] that they

used in their experiments, they discovered that the maximum number of feasible paths in the programs was 8, which is a number that is significantly less than the number of all paths in a control-flow graph. Furthermore, this method will enable the monitor to dynamically switch 'modes' (or sampling periods) so the monitor may operate at different frequencies at the same program point depending on the state of the program. The two problems that were explored in this thesis could benefit from leveraging the idea of using symbolic execution:

- Reduction in the amount of time required to solve problem instances; given the results of [20], it could make hybrid runtime verification more scalable.
- Improved solution accuracy. The accuracy comes from utilizing the program state at run time as opposed to fixing the instrumentation and monitoring schemes statically (i.e., before run time). Dynamically selecting the instrumentation and monitoring schemes would likely reduce monitoring overhead by making 'better' decisions, provided that the cost of conditionally instrumentation program points are kept relatively low in comparison to other monitoring costs.

Symbolic execution occurs prior to run time and may still present a barrier to time-triggered and hybrid runtime verification because of the resources that are required to compute feasible and (near-)optimal instrumentation schemes. With that said, other dynamic solutions for time-triggered and hybrid runtime verification may scale better for larger programs under inspection. Leveraging existing work on learning algorithms, probabilistic models, and control theory (i.e., discrete-event systems) may alleviate some of the cost required to generate effective and efficient time-triggered and/or hybrid runtime verification solutions.

References

[1] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology, RTI Project*, no. 7007.011, 2002.

[2] R. Charette, "Why software fails [software failure]," *Spectrum, IEEE*, vol. 42, no. 9, pp. 42–49, 2005.

[3] ——, "This car runs on code," *IEEE Spectrum*, vol. 46, no. 3, p. 3, 2009.

[4] B. Bonakdarpour and S. Fischmeister, "Runtime monitoring of time-sensitive systems – tutorial supplement," in *Proc. of the 2nd International Conference on Runtime Verification (RV)*, San Francisco, USA, 2011.

[5] E. M. Clarke and J. M. Wing, "Formal methods: state of the art and future directions," *ACM Comput. Surv.*, vol. 28, no. 4, pp. 626–643, dec 1996. [Online]. Available: http://doi.acm.org/10.1145/242223.242257

[6] M. Leucker and C. Schallhart, "A brief account of runtime verification," *Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1567832608000775

[7] E. Clarke and E. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Logics of Programs*, ser. Lecture Notes in Computer Science, D. Kozen, Ed. Springer Berlin / Heidelberg, 1982, vol. 131, pp. 52–71, 10.1007/BFb0025774. [Online]. Available: http://dx.doi.org/10.1007/BFb0025774

[8] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded Model Checking," ser. Advances in Computers. Elsevier, 2003, vol. 58, pp. 117–148. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0065245803580032

[9] J. Burch, E. Clarke, K. McMillan, and D. Dill, "Sequential circuit verification using symbolic model checking," in *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, jun 1990, pp. 46–51.

[10] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *Computers, IEEE Transactions on*, vol. C-35, no. 8, pp. 677–691, aug 1986.

[11] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, W. Cleaveland, Ed. Springer Berlin / Heidelberg, 1999, vol. 1579, pp. 193–207, 10.1007/3-540-49059-0_14. [Online]. Available: http://dx.doi.org/10.1007/3-540-49059-0_14

[12] S. Colin and L. Mariani, "Run-Time Verification," in *Model-Based Testing of Reactive Systems*, ser. Lecture Notes in Computer Science, M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds. Springer Berlin / Heidelberg, 2005, vol. 3472, pp. 525–555. [Online]. Available: http://dx.doi.org/10.1007/11498490_24

[13] A. Pnueli and A. Zaks, "PSL model checking and run-time verification via testers," in *Proceedings of the 14th international conference on Formal Methods*, ser. FM'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 573–586. [Online]. Available: http://dx.doi.org/10.1007/11813040_38

[14] A. Bauer, M. Leucker, and C. Schallhart, "Runtime Verification for LTL and TLTL," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, pp. 14:1–14:64, Sep. 2011. [Online]. Available: http://doi.acm.org/10.1145/2000799.2000800

[15] ——, "Comparing LTL Semantics for Runtime Verification," *J. Log. and Comput.*, vol. 20, no. 3, pp. 651–674, Jun. 2010. [Online]. Available: http://dx.doi.org/10.1093/logcom/exn075

[16] K. Havelund and A. Goldberg, "Verify Your Runs," in *Verified Software: Theories, Tools, Experiments*, B. Meyer and J. Woodcock, Eds. Berlin,

Heidelberg: Springer-Verlag, 2008, pp. 374–383. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69149-5_40

[17] D. Giannakopoulou and K. Havelund, "Automata-based verification of temporal properties on running programs," in *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*, Nov. 2001, pp. 412–416.

[18] B. Bonakdarpour, S. Navabpour, and S. Fischmeister, "Sampling-based runtime verification," in *Proceedings of the 17th international conference on Formal methods*, ser. FM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 88–102. [Online]. Available: http://dl.acm.org/citation.cfm?id=2021296.2021308

[19] S. Navabpour, C. W. W. Wu, B. Bonakdarpour, and S. Fischmeister, "Efficient Techniques for Near-Optimal Instrumentation in Time-Triggered Runtime Verification," in *Runtime Verification*, ser. Lecture Notes in Computer Science, S. Khurshid and K. Sen, Eds. Springer Berlin Heidelberg, 2012, vol. 7186, pp. 208–222. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-29860-8_16

[20] S. Navabpour, B. Bonakdarpour, and S. Fischmeister, "Path-aware Time-triggered Runtime Verification," in *Runtime Verification*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, to appear in 2013.

[21] L. Pike, A. Goodloe, R. Morisset, and S. Niller, "Copilot: A Hard Real-Time Runtime Monitor," in *Runtime Verification*, ser. Lecture Notes in Computer Science, H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky, and N. Tillmann, Eds. Springer Berlin / Heidelberg, 2010, vol. 6418, pp. 345–359. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16612-9_26

[22] S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok, "Runtime verification with state estimation," in *Proceedings of the Second international conference on Runtime verification*, ser. RV'11. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 193–207. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-29860-8_15

[23] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan, "Java-MaC: A Run-time Assurance Tool for Java Programs," *Electronic Notes in Theoretical*

Computer Science, vol. 55, no. 2, pp. 218–235, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066104002543

[24] K. Havelund and G. Roşu, "Monitoring Java Programs with Java PathExplorer," *Electronic Notes in Theoretical Computer Science*, vol. 55, no. 2, pp. 200–217, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066104002531

[25] O. Kupferman and M. Y. Vardi, "Model Checking of Safety Properties," *Form. Methods Syst. Des.*, vol. 19, no. 3, pp. 291–314, oct 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1011254632723

[26] K. Havelund and G. Roşu, "Monitoring Programs Using Rewriting," in *Proceedings of the 16th IEEE international conference on Automated software engineering*, ser. ASE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 135–. [Online]. Available: http://dl.acm.org/citation.cfm?id=872023.872572

[27] ——, "Synthesizing Monitors for Safety Properties," in *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS '02. London, UK, UK: Springer-Verlag, 2002, pp. 342–356. [Online]. Available: http://dl.acm.org/citation.cfm?id=646486.694486

[28] V. Stolz and E. Bodden, "Temporal Assertions using AspectJ," *Electron. Notes Theor. Comput. Sci.*, vol. 144, no. 4, pp. 109–124, May 2006. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2006.02.007

[29] G. Roşu, F. Chen, and T. Ball, "Synthesizing Monitors for Safety Properties: This Time with Calls and Returns," in *Runtime Verification*, ser. Lecture Notes in Computer Science, M. Leucker, Ed., vol. 5289. Springer Berlin Heidelberg, 2008, pp. 51–68. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89247-2_4

[30] K. Havelund and G. Roşu, "Efficient Monitoring of Safety Properties," *Int. J. Softw. Tools Technol. Transf.*, vol. 6, no. 2, pp. 158–173, Aug. 2004. [Online]. Available: http://dx.doi.org/10.1007/s10009-003-0117-6

[31] W. Zhou, O. Sokolsky, B. Loo, and I. Lee, "Dmac: Distributed monitoring and checking," in *Runtime Verification*, ser. Lecture Notes in Computer Science,

S. Bensalem and D. Peled, Eds. Springer Berlin Heidelberg, 2009, vol. 5779, pp. 184–201. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04694-0_13

[32] "Monitoring, Checking, and Steering of Real-Time Systems," *Electronic Notes in Theoretical Computer Science*, vol. 70, no. 4.

[33] 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*, vol. 24, pp. 129–155, 2004. [Online]. Available: http://dx.doi.org/10.1023/B%3AFORM.0000017719.43755.7c

[34] M. d'Amorim and G. Roşu, "Efficient Monitoring of ω-Languages," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, K. Etessami and S. Rajamani, Eds. Springer Berlin / Heidelberg, 2005, vol. 3576, pp. 311–318, 10.1007/11513988_36. [Online]. Available: http://dx.doi.org/10.1007/11513988_36

[35] Y. Falcone, J.-C. Fernandez, and L. Mounier, "Runtime Verification of Safety-Progress Properties," in *Runtime Verification*, ser. Lecture Notes in Computer Science, S. Bensalem and D. Peled, Eds. Springer Berlin Heidelberg, 2009, vol. 5779, pp. 40–59. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04694-0_4

[36] Z. Manna and A. Pnueli, "A hierarchy of temporal properties (invited paper, 1989)," in *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, ser. PODC '90. New York, NY, USA: ACM, 1990, pp. 377–410. [Online]. Available: http://doi.acm.org/10.1145/93385.93442

[37] E. Chang, Z. Manna, and A. Pnueli, "Characterization of temporal property classes," in *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, W. Kuich, Ed. Springer Berlin / Heidelberg, 1992, vol. 623, pp. 474–486, 10.1007/3-540-55719-9_97. [Online]. Available: http://dx.doi.org/10.1007/3-540-55719-9_97

[38] M. B. Dwyer, A. Kinneer, and S. Elbaum, "Adaptive Online Program Analysis," in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 220–229. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.12

[39] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-Based Runtime Verification," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, B. Steffen and G. Levi, Eds. Springer Berlin / Heidelberg, 2004, vol. 2937, pp. 277–306. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24622-0_5

[40] E. Bodden, L. Hendren, and O. Lhoták, "A Staged Static Program Analysis to Improve the Performance of Runtime Monitoring," in *ECOOP 2007 Object-Oriented Programming*, ser. Lecture Notes in Computer Science, E. Ernst, Ed. Springer Berlin / Heidelberg, 2007, vol. 4609, pp. 525–549. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-73589-2_25

[41] E. Bodden, L. Hendren, P. Lam, O. Lhoták, and N. A. Naeem, "Collaborative runtime verification with tracematches," in *Proceedings of the 7th international conference on Runtime verification*, ser. RV'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 22–37. [Online]. Available: http://dl.acm.org/citation.cfm?id=1785141.1785146

[42] X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok, "Software monitoring with controllable overhead," *Int. J. Softw. Tools Technol. Transf.*, vol. 14, no. 3, pp. 327–347, Jun. 2012. [Online]. Available: http://dx.doi.org/10.1007/s10009-010-0184-4

[43] H. Zhu, M. Dwyer, and S. Goddard, "Predictable Runtime Monitoring," in *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, Jul. 2009, pp. 173–183.

[44] E. Bartocci, R. Grosu, A. Karmarkar, S. A. Smolka, S. D. Stoller, E. Zadok, and J. Seyster, "Adaptive runtime verification," in *Proc. 3rd International Conference on Runtime Verification (RV'12)*, Istanbul, Turkey, sep 2012.

[45] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[46] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE*

International Workshop, ser. WWC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14. [Online]. Available: http://dx.doi.org/10.1109/WWC.2001.15

[47] "ILP solver lp_solve," http://lpsolve.sourceforge.net/5.5/.

[48] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[49] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of c programs," Proceedings of Conference on Compilier Construction, 2002.

[50] "GNU debugger," http://www.gnu.org/software/gdb/.

[51] "SNU Real-Time Benchmarks," http://www.cprover.org/goto-cc/examples/snu.html.

[52] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in *International Symposium on Code Generation and Optimization: Feedback Directed and Runtime Optimization*, 2004, p. 75.

[53] GrammaTech Inc., "CodeSurfer®," http://www.grammatech.com/products/codesurfer/.

[54] SRI, "Yices: An SMT Solver (1.0.34)," http://yices.csl.sri.com/index.shtml.

[55] C. Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756