Runtime Verification with Controllable Time Predictability and Memory Utilization

by

Deepak Kumar

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

Abstract

The goal of runtime verification is to inspect the well-being of a system by employing a monitor during its execution. Such monitoring imposes cost in terms of resource utilization. Memory usage and predictability of monitor invocations are the key indicators of the quality of a monitoring solution, especially in the context of embedded systems. In this work, we propose a novel control-theoretic approach for coordinating time predictability and memory utilization in runtime monitoring of real-time embedded systems. In particular, we design a PID controller and four fuzzy controllers with different optimization control objectives. Our approach controls the frequency of monitor invocations by incorporating a bounded memory buffer that stores events which need to be monitored. The controllers attempt to improve time predictability, and maximize memory utilization, while ensuring the soundness of the monitor. Unlike existing approaches based on static analysis, our approach is scalable and well-suited for reactive systems that are required to react to stimuli from the environment in a timely fashion. Our experiments using two case studies (a laser beam stabilizer for aircraft tracking, and a Bluetooth mobile payment system) demonstrate the advantages of using controllers to achieve low variation in the frequency of monitor invocations, while maintaining maximum memory utilization in highly non-linear environments. In addition to this problem, the thesis presents a brief overview of our preceding work on runtime verification.

Acknowledgements

I am grateful to my supervisor, Dr. Sebastian Fischmeister, for providing me this opportunity and his ample support. He not only guided me in the project, but also encouraged me at every step and expressed confidence in my efforts.
I would also like to express my gratitude to Dr. Borzoo Bonakdarpour for guiding me in my reasearch. Despite mutiple projects going on simultaneously under his guidance, he always helped me promtly and made sure that my work did not suffer any delays.
I would also like to extend my thanks to my friend Ramy Medhat and other colleagues at the Real-Time Embedded Software Group, who helped me in completing this work with their valuable comments and suggestions.
I am thankful to my friends in Waterloo for making my stay memorable and a great learning experience.
It would never be sufficient to thank my parents and grandparents; they have always been there for me throughout my life, and it is their success that my life is about to be adorned with a master's degree.

Dedication

This thesis is dedicated to my mother Smt. Kamlesh Devi and my father Shri Umed Singh Rangi.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Program debugging and verification have been one of the most expensive part of the software development cycle and its cost can range from 50 to 75 percent of the total development cost [25]. Software issues create huge managerial and financial impact on the software industry. A recent report of National Institute of Standards and Technology (NIST) suggests that \$59.6 billion are lost to software issues each year [46]. Another report by Charette [13] lists some incidents that incurred great financial losses. For example, a \$3.45 billion tax-credit overpayment was made by Inland Revenue (from the United Kingdom) in 2004–2005 that was attributed to software errors. It is important to realize that the losses are not limited to monetary considerations, but may also include loss of human lives. Modern aircrafts and automobiles run on millions of lines of code. For example, 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. Whereas lines of code executed in automobiles have increased from 50,000 in 1981 to 100M in current generation [47]. The complexity of such systems is increasing rapidly [14] and any failures due to software bug may be fatal for the people operating and using them. Hence, it is of utmost importance to ensure correctness in such critical software applications.

Program verification and testing are the two commonly used practices to ensure program correctness. Program verification includes model checking and theorem proving, that verify the program behavior against a set of properties. Such verification is exhaustive because it aims at analyzing all the execution paths. However, program verification requires developing a correct mathematical model of the system and has limited application due to state-explosion problem. On the other hand, exhaustively covering all parts of large code through testing is infeasible. Therefore, depending on the test case, testing scrutinizes only a subset of behaviors of the system and can miss testing of unanticipated critical part. The limitations of program verification and testing necessitates development of complementary techniques that can check the correctness of the program at runtime. *Runtime verification* [3, 15, 23, 27, 35, 38] (RV) is one such complementary technique, where a *monitor* checks at run time whether or not the execution of a system under inspection satisfies a given correctness property. If the monitor observes that the system is about to violate a property, it can trigger a steering method, so the system is led to a safe behavior. The ability of a monitor to evaluate the system's properties at run time and take all the system dynamics as well as environment stimuli into account has made RV an excellent technique to ensure the well-being of computing systems, especially in the domain of embedded safety/mission-critical systems.

The inherent cost of RV is runtime overhead. In the context of embedded systems, this cost by itself is not the main obstacle in augmenting a system with RV technology. The more significant problem is the fact that if events that would potentially invoke the monitor do not occur in a time-predictable manner (e.g., periodic), monitoring tasks can severely intervene the normal system execution, thereby, causing deadline misses and unscheduled resource utilization. To tackle this problem, there has recently been an emerging trend on designing *time-triggered* monitors. Such a monitor is invoked within fixed time intervals, while ensuring soundness. The existing methods incorporate static analysis techniques to ensure minimum instrumentation [10] and execution path-aware adjustment of monitor invocation [36] to decrease the overhead. However, deep static analysis techniques suffer from two drawbacks: they (1) may not scale, and (2) are completely blind to system dynamics and environment actions at run time, especially in *reactive* systems. For example, in an inverted pendulum, the behavior of the system depends more on forces and their angles rather than execution path of the program that controls the pendulum. Thus, static analysis techniques by themselves are not well-equipped to deal with *reactive* systems.

With this motivation, we focus on designing a RV technique, where the monitor should react to system dynamics and environment actions, while taking resource limitations into account. We, in particular, target reactive embedded real-time systems, where time-predictability plays an important role and memory usage is limited by physical constraints. To this end, we require the following:

1. The monitor is not invoked by occurrence of each event that may change the valuation of properties and rather invoked within time intervals, called the *polling period*. In order to enforce property violation detection latency, the polling period cannot be greater than some value given as a system parameter. The monitor is also required to maintain *minimum jitter* in changing the polling period.

2. The monitor must be *sound*; i.e., false-positives and false-negatives are not acceptable.

3. Since the monitor is invoked within time intervals, multiple events of interest may happen between two monitor invocations. Thus, the program under inspection must be instrumented such that the events of interest between the two invocations are buffered. We assume that the program under inspection provides only a bounded-size buffer. This buffer is required to be filled with *maximum utilization*.

In order to achieve the above requirements and make the polling period resilient to non-uniform environment actions, the monitor must be able to learn, predict, and adapt to the environment stimuli at run time. To design such a monitor, we utilize the rich literature of control theory to enforce the three aforementioned requirements. The controller (see Figure 1.1) executes within the monitor thread. With every invocation of the monitor, the controller determines when the next invocation should occur to satisfy the memory utilization and time predictability objectives. In order to maintain soundness, no events should be dropped from the buffer. Thus, when the buffer is full, the monitor invocation is automatically triggered ahead of its scheduled invocation. We design five controllers:

- A PID feedback controller with variable polling period for systems in which events of interest are expected to occur linearly. This controller aims at maximizing memory utilization.
- Four fuzzy controllers for handling systems where events of interest occur in a non-linear fashion. Moreover, each controller targets achieving a different objective with respect to our problem:
  - The first fuzzy controller attempts to maximize memory utilization, similar to the PID controller.
  - The second fuzzy controller targets both memory utilization and time predictability. The controller attempts to balance between (1) polling periods that would minimize the empty spaces in the buffer, and (2) choosing intervals as close as possible to the mean of all previous intervals.
  - Fuzzy controllers 3 and 4 attempt to maintain an upper bound on the variance of intervals. Their difference is in their internal decision process.

We conduct two thorough case studies. The first case study is on a Bluetooth mobile payment system. This system is highly non-linear and our experiments clearly demonstrate the advantages of using our controllers to achieve low variation in the monitor polling period, while maintaining maximum memory utilization in highly non-linear environments.

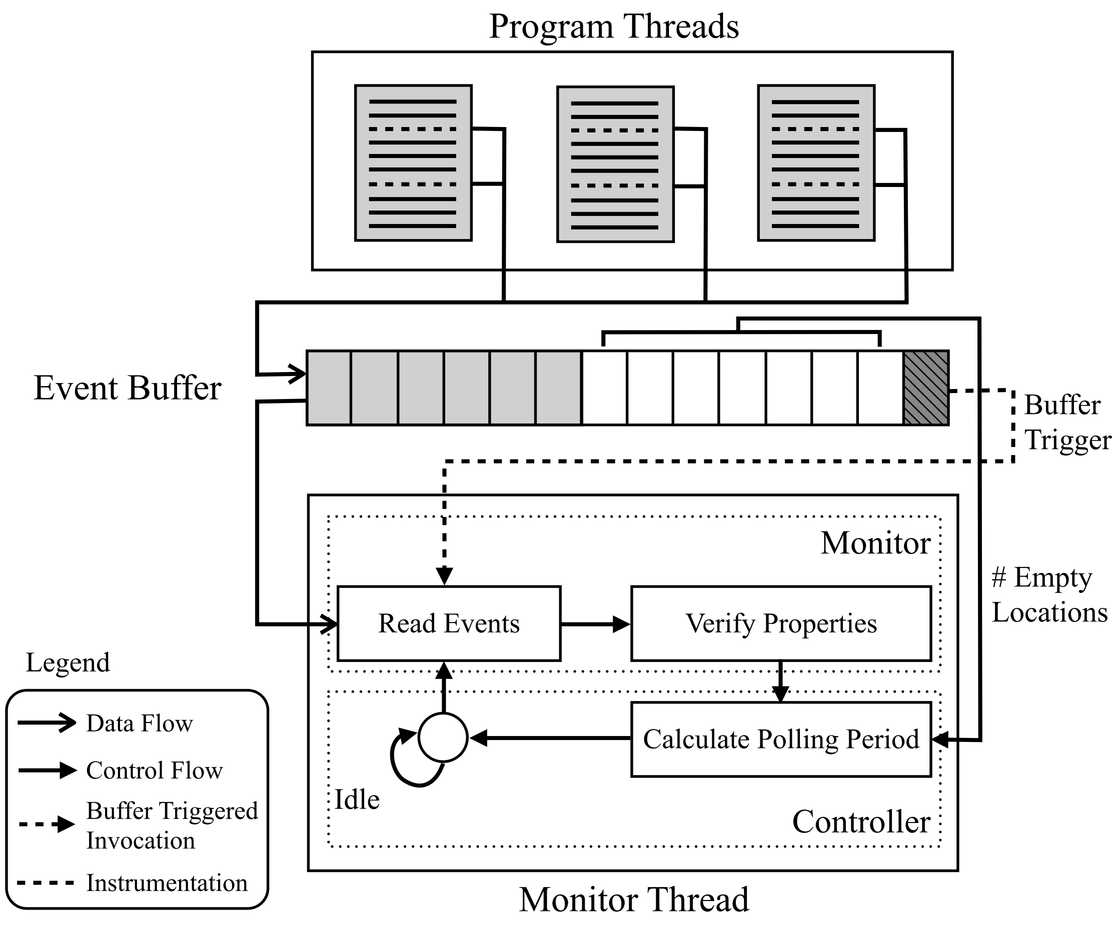


Figure 1.1: Outline of the proposed controller design.

The second case study is on a laser beam stabilizer (LBS) used for aircraft tracking. The RV system for LBS aims at monitoring the location of the laser beam, where aircraft movements are the environment actions. The laser beam control software works periodically and although aircraft movements happen non-linearly, the buffer gets filled up within periods uniformly. This characteristic may unnecessecitate a monitor controller. However, we conduct this experiment to demonstrate that the controlled monitor performs as well as the uncontrolled monitor. That is, our controller introduces negligible disturbance to the system.

1.1 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 presents a summary of related work in the literature on the reduction of resource requirement by runtime verification, techniques designed to make runtime verification predictable, and use of fuzzy logic to solve the problem of buffer management in communication networks which is somewhat similar to our problem. Chapter 3 recaps the basic control theory concepts on PID and fuzzy controllers that are useful to understand the proposed solutions later in Chapter 5. Then, in Chapter 4, we formally state the problem by introducing monitoring objectives. Our controller design choices are explained in Chapter 5. We present our experiment design, and two case studies and the experimental results in Chapter 6 and 7 respectively. In Chapter 8 we present an overview of our preceding work on runtime verification. Finally, we make concluding remarks and discuss future work in Chapter 9 and Chapter 10 respectively.

Chapter 2

Literature Review

This chapter presents a brief overview of the existing work and literature on runtime verification and control theory. Main focus of work discussed in this chapter is related to efficient implementation and invocation of monitors, and use of different type controllers to achieve desired output in non-linear systems.

Compared to traditional methods that ensure the correctness of software before their execution, *Runtime verification* is a relatively new research area which aims to dynamically observe, scrutinize and sometime even guide the execution of a possibly incorrect system. This way runtime verification is more of a complementary technique to testing and model verification than an alternative. Under the paradigm of program checking, Blum and Kannan [4] first presented a rigorous technique to proving the correctness at run time. This was done because, despite the best verification and testing efforts, problems can happen at run time and a dynamic scrutiny of system is required to improve the confidence in the system evolution. Currently, runtime verification is not only being used for system monitoring purposes but also for understanding programs, detecting security or safety violations, performing fault recovery, etc.

Classically, in runtime verification [26, 32, 38] a system comprises an observer module called *monitor*. During the execution program generates execution traces, i.e., sequence of states or events that can affect the correctness properties. Generally traces are generated with the help of efficient instrumentation [6, 16, 31]. The monitor then dynamically checks these traces against formally specified properties (generally formal languages such as *linear temporal logic* [23, 40, 45] or ω-language are used for specifications [17] and verification purposes) to detect whether or not the execution of the system satisfies the given correctness properties. If any violation is detected, the monitor can take actions such as raising an alarm, stopping the execution of system or switching to a secondary mode [42].

Outline. The rest of this chapter outlines some recently published research work to address the challenges in making applicablity and feasiblity of runtime verification more predictable and resource efficients in terms of overhead. Section 2.1 presents the literature survey of work done to reduce overhead cost of monitoring, and hence effectively utilize the computational resources. Section 2.2 discusses work done by authors to make runtime verification predictable with the help of time-triggered monitor invocation. In section 2.3, we present a literature survey of techniques used to manage data buffers in communication networks which is similar to our problem.

2.1 Overhead Reduction

Most of the work proposed to reduce the runtime verification overhead is done by combining static analysis and dynamic runtime checking. In this regard, Eric Bodden et al. [7, 5] propose the use of static analysis of the program to be monitored to identify instrumentation points that can be removed without affecting the correctness of verification. This work was further implemented in the tool Clara [8]. On the similar line of work, authors in [19] aim to reduce monitor invocations. In their framework, *Adaptive Online Program Analysis*, they show that monitor invocation can be lowered by dynamically adjusting the runtime monitoring scheme with the help of semantic analysis of the correctness properties and the program state. The overall overhead is reduced because of efficient synthesis of monitor and intelligently placed instrumentation.

In [2], authors propose that monitoring overhead can be reduced with the help of efficient logic to express the correctness properties. Authors propose a unifying logic, EAGLE, to express safety and liveliness properties, and demonstrate that monitors synthesized from the properties expressed in EAGLE do not require to store program traces. This reduces the memory overhead of runtime verification. Some other works on reducing the monitoring overhead based on similar ideas include improved instrumentation [16], static analysis [5], and efficient monitor generation [17].

In [49] authors propose a novel *hybrid* method of monitor invocation that leverages the benefits of both event- and time-triggered methods to reduce the overall monitoring overhead. This work demonstrates that, in case of frequently occuring critical events event-triggered approach suffers higher overhead as compared to time-triggered approach due to frequent context switching. Whereas time-triggered approach takes unnecessary samples if events are sparsely distributed. Authors proposed that, with the help of static analysis of program one can effectively decide which techniques should be used on different paths to achieve lower overhead.

In some cases, it is possible to compromise the accuracy of verification, and it can be used to lower the overhead of monitoring. Huang, et al. [28] propose a control-theoretic based *Software monitoring with controllable overhead* (SMCO) technique where PID controllers are used to maintain overhead below a user-defined threshold level. Controllers maintain the given target by disabling the monitor whenever an overload is caused by sudden bursts of critical events in a short span. This is the first work that uses controllers to reduce runtime verification overhead. While their technique effectively maintains the overhead below the pre-specified level, it does not ensures correctness. To tackle this problem, Bartocci et al. [44] augment the method presented by Huang, et al. [28] with a hidden Morkov model (HMM) to fill the gaps in event sequences.

Except SMCO, all of the above proposed work for overhead reduction require static analysis and make these technique unscalable like model checking. Although SMCO and extended work by Bartocci et al. [44] do not require static analysis, they are also unscalable because multiple PID controllers require tuning, which is system specific and consumes huge amount of time of the designer.

2.2 Predictable Runtime Verification

The increased use of runtime verification to ensure the correct behaviour of cyber-physical systems required more predictable runtime verification models and researchers started work toward this goal. Initially monitoring was used to verify timing constraints of real-time systems with the help of time stamps and clock-synchronization [30]. Later different approaches such as real-time monitoring, time-triggered monitoring, and networked monitoring were developed to meet timing requirements of real-time embedded systems. Rest of this Section describes some recent approaches related to predictable runtime verification.

Pike et al. [37] developed domain-specific language, Copilot, which can be used for synthesis of time-aware monitor. Monitors synthesized using Copilot periodically sample the system state and adhere to hard real-time deadlines. However, the proposed monitors are not sound as the approach does not consider the effect of change in program state between two consecutive samples. Another limitation of Copilot based monitors is their inability to monitor local variables. Also this work is limited to hard-real time systems because synthesis of monitor requires knowledge of detailed timing behavior of system.

Time-triggered monitoring is another idea proposed by Bonakdarpour et al. [9], where monitor is invoked for verification only after a fixed period called sampling period (equivalent to polling period). The correctness of such monitor is maintained by formally calculating a sampling period using static analysis in such a way that no state change is missed between two consecutive samples. However such sampling period is very conservative as it depends on the best case execution time between the two most densely packed state changes. To this end authors demonstrate that it is possible to increase sampling period and hence reduce monitoring overhead, arising due to high frequency of monitor invocation, with the help auxiliary memory to buffer program states between two consecutive samples. Buffering is done in such a way that monitor can correctly reconstruct the system states between two samples and thus soundness of verification is ensured. This work also explores methods for balancing the trade off between auxiliary memory and sampling period.

Navabpour et al.[36] further extend the work in [9] with the help of *symbolic execution* [33] to predict the set of feasible execution paths before execution. Longest sampling period for each of the predicted path is computed separately. At the runtime, monitor dynamically adjusts its sampling period to longest sampling period of the path which the program is currently executing. This scheme helps in effective utilization of computing resources while scarifying a little on monitoring predictability.

Bonakdarpour et al. [11] propose a technique called time-triggered self-monitoring, where self-sampling instrumentation instructions are inserted into the program. Number of such instrumentation instructions is minimized by formulating an optimization problem and solving it using a SAT solver or a heuristic. This technique eliminates the assistance from external monitor or internal timer, and manages its timing based on program's temporal behavior.

Time-triggered monitoring provides an effective way of ensuring predictability of monitors, and increases the applicability of runtime verification in real-time domain. However these methods require extensive static analysis, knowledge of best case execution time and temporal distribution of critical events for a given program. This severely limits the application of these approaches for complex systems and reactive non-linear systems which is the focus of our work.

2.3 Fuzzy Logic in Telecommunication Network

Complex networks are becoming more and more dynamic while input traffic is increasing rapidly and getting highly uncertain. This results into unexpected overloads and network failures. It has limited the application of accurate analytical modeling to manage important network aspects. Since fuzzy set theory provides robust mathematical model to deal with uncertain real-world problems, researchers have explored use of fuzzy logic for buffer management, routing, load balancing and congestion mitigation problems. Among them, buffer management is closely related to the problem we discuss in this work. This is because, the objective of buffer management policies is to effectively use bounded buffer to minimize transmission delay. Bonde et al. [12] demonstrate the use of fuzzy logic to define soft thresholds in cell-switching networks. They use linguistic system variables to model buffer queues. Fuzzy logic based soft thresholds results in robust and adaptive buffer management, in contrast to traditionally used binary thresholds. They show that buffer queues managed through fuzzy logic exhibit greater resilience and adaptability towards rapidly changing network traffic.

Although, understanding the use of fuzzy logic for buffer management provides insight into possible use of fuzzy theory for our problem. However the solution can not be used as-is. This is because the constraints and objectives of the two problems are different. In buffer management objective is to predict full buffer criteria under non-linear overloads. This is then used by higher level policy to make appropriate decisions once the buffer is full. Whereas our problem has two conflicting objectives: (1) effectively utilizing the buffer, and (2) maintaining time predictability of the monitor. These two objectives have opposite effect on polling rate. Another constraint is that buffer overflow be minimized and no event is dropped.

Chapter 3

Basic Control Theory

Control theory is one of the most researched discipline of engineering that deals with behaviour of dynamic systems. Broadly there are three components of a closed loop control-system:

Plant The target system that needs to be controlled

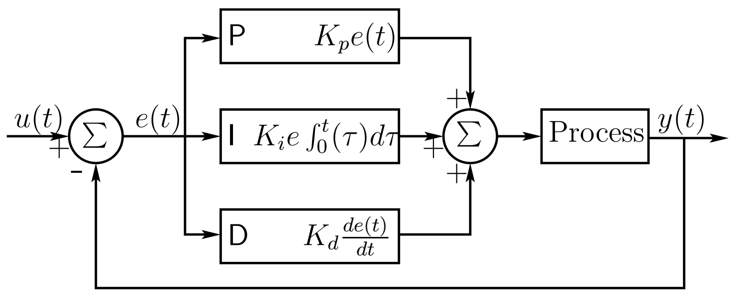
Transducer Measures output of plant and generates an appropriate input to the controller.

Controller Generates an output which is used to steer plant's output toward the reference input. The generated output is a function of the error (the difference between the input received from transducer and the reference input).

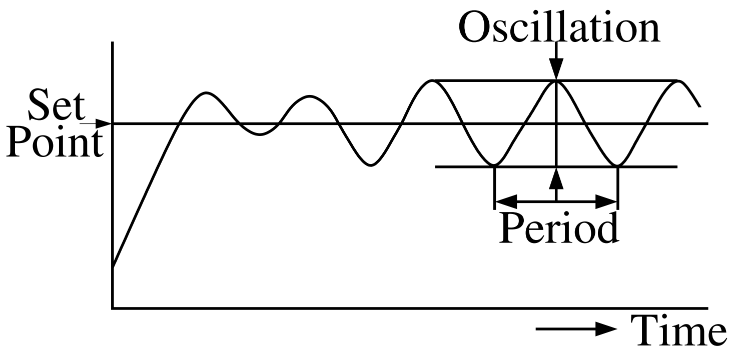
However there are open loop control systems where controller does not receive feedback from plant. There are multiple classifications of control systems. In this work we are concerned with PID and fuzzy controllers, we recap the concepts of these controllers in Section 3.1 and Section 3.2 respectively.

3.1 PID Controller

A PID feedback controller [39] consist of (1) a *proportional*, (2) an *integral*, and (3) a *derivative* component. An *error signal* $e(t)$ is sampled within fixed time intervals called



(a) Structure



(b) Tuning through stable oscillation.

Figure 3.1: PID controller.

the *sampling period*. The three components are then applied collectively to $e(t)$ as follows:

$$u(t) = K_P\, e(t) + K_I \int e(t)dt + K_D \frac{d}{dt} e(t) \tag{3.1}$$

where K_P is the proportional gain, K_I is the integral gain, and K_D is the differential gain. Figure 3.1(a) demonstrates the structure of a PID controller.

PID controllers are often used to control *linear* systems. One approach to using PIDs is to model the system accurately, so as to deduce ideal gains that ensure controllable behavior. Another method is using experience to configure these controllers; often engineers on site can come up with an initial configuration for PID controllers using well known methods. In this work, we use the popular Ziegler-Nichols method [50] to tune K_P, K_I, and K_D. We begin by disabling K_I and K_D, and increasing K_P gradually until oscillation begins with a constant amplitude (see Figure 3.1(b)), where

$$e = \mathit{SetPoint}\ -\ \mathit{Feedback\ Reading}$$

SetPoint is the desirable set point and *Feedback Reading* is output of the plant. The gain at which oscillation begins is called the ultimate gain K_U. Using K_U and the oscillation period T_U, we can determine the values of K_P, K_I, and K_D by substituting in the Ziegler-Nichols rules.

The main drawback of PID controllers is that their performance in non-linear systems is variable, as they are inherently linear. The engineer is, hence, faced with the trade-off of decreasing overshoot versus decreasing settling time.

3.2 Fuzzy Controller

A fuzzy controller is often considered as a real-time expert system that relies in part on the system operator's expertise in the form of situation/action rules [18]. This differs from PID controllers in that fuzzy controllers mainly describe what the system's operator would do in different situations based on a set of *fuzzy* conditions, which resembles our human perception of conditions/actions such as the control we employ while driving. This fundamental basis enables fuzzy controllers to outperform PID controllers in non-linear systems.

Fuzzy Logic

The first function of a fuzzy controller is to transform a discrete measured value (called a *crisp* value) to a *fuzzy* value. We first define *fuzzy sets* as sets, whose elements have degrees of membership to that set. For a universe $\mathcal{U}$, each fuzzy set is associated with a *membership function* (μ), which maps each value $u \in \mathcal{U}$ to a value within the interval $[0, 1]$, indicating the percentage of membership of the value u to the set. That is

$$\mu : \mathcal{U} \rightarrow [0, 1]$$

An *if-then* implication rule is generally of the form "if X is A then Y is B", where X is a fuzzy variable, A is an antecedent fuzzy set, Y is an output fuzzy variable and B is a consequent fuzzy set. In fuzzy logic, there are many methods with which we can perform inference based on this implication. We use *scaled inference*, which has the advantage of preserving the shape of the membership function. In scaled inference, an implication is represented by scaling the consequent membership function with the degree of membership of the crisp value in the antecedent function. Thus, for an if-then rule, scaled inference S is calculated as follows:

$$\mu_S(x, y) = \mu_A(x) \cdot \mu_B(y)$$

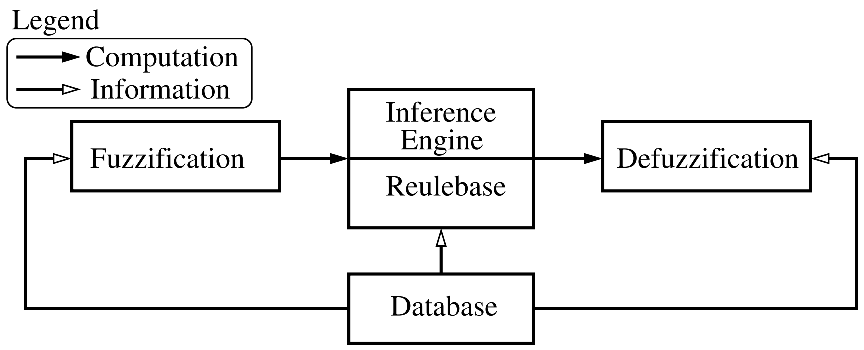


Figure 3.2: Structure of a Fuzzy Controller [18].

where x is the the measured crisp value of the fuzzy variable X and y is the output crisp value of fuzzy variable Y. This process of evaluating the above equation is called *firing*.

Applying scaled inference to support multiple rules is our goal in fuzzy controllers, since we need to control the system using a set of rules that account for the expert's response in different situations. There are two ways to apply scaled inference to multiple rules: (1) composition-based inference, and (2) individual-rule-based inference. The difference between these two methods is that in individual-rule-based inference, each rule is fired individually and then a union is calculated for all rules. Composition-based inference calculates the union first and then fires the resulting set. The output for both methods is the same when using scaled inference. Thus, for a given $u \in \mathcal{U}$, the result of firing the set of rules using individual rule-based inference is obtained by the following equation:

$$\mu_I(u) = \max_k \{\mu_{A^{(k)}}(x) \cdot \mu_{B^{(k)}}(u)\} \tag{3.2}$$

where k is the enumerator over the set of rules, and x is the crisp input.

Structure of a Fuzzy Controller

Figure 3.2 shows the structure of a typical fuzzy controller [18]. A fuzzy controller consists of the following components:

Fuzzification When a fuzzy controller receives a measured value from the system, this value must be *fuzzified*, so that its membership to the associated fuzzy sets could be determined. As mentioned earlier, in this work, we use scaled inference for fuzzification.

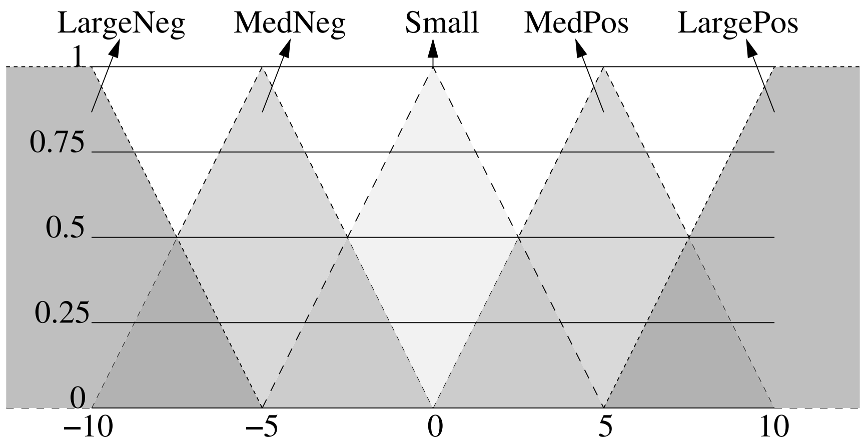


Figure 3.3: Membership functions of input fuzzy sets.

Knowledge base This component consists of a *rulebase* and a *database*. The rulebase contains the set of rules including the antecedents and consequents. The database contains the membership functions of fuzzy sets. In common practice there are five fuzzy sets for each fuzzy variable: LargeNeg, MedNeg, Small, MedPos, and LargePos. The membership functions for these sets are *lambda-type* functions, with the exception of LargeNeg and LargePos, which are Z-type and S-type respectively [41]. An example of these functions is shown in Figure 3.3.

Inference engine The inference engine employs either composition-based inference or individual-rule-based inference, described above. The latter is more widely used in fuzzy control since it is computationally more efficient and uses less memory.

Defuzzification This component transforms the output of the inference engine into one single point-wise value. This value is then applied to the system to complete the control loop. The most widely used method for defuzzification is *gravity defuzzification*, which calculates the center of gravity for $\mu_I(u)$ in Equation 3.2. The output crisp value u^* is calculated as follows:

$$u^* = \frac{\displaystyle\int_{-\infty}^{+\infty} u \cdot \mu_I(u)\, du}{\displaystyle\int_{-\infty}^{+\infty} \mu_I(u)\, du} \tag{3.3}$$

Chapter 4

Problem Description

Expressing logical properties of a system normally involves a set of program variables whose value may change over time. We call such change of value an *event*. Monitoring an event involves invoking a process (called the *monitor*) that evaluates the properties associated with that event at run time. This work is concerned with the problem of runtime verification of reactive systems, where the monitor is required to exhibit the following features simultaneously:

Soundness For verification to be *sound*, all events should be monitored. That is, no event that can potentially change the valuation of a property is overlooked.

Time predictability Since invocation of the monitor interrupts the program execution, we require that these interruptions are predictable with respect to time. This requirement assists in achieving more accurate system-wide scheduling.

Resource utilization The monitor may use bounded memory space to buffer events. We require maximum utilization of this buffer.

We now formulate the above constraints. Let R be a reactive system with limited memory under inspection and Φ be a set of logical properties (e.g., in LTL), where R is expected to satisfy Φ. Since, system R has limited memory, we assume that the number of events that it can buffer for monitoring has an upper bound B.

Let $E = e_1 e_2 \cdots e_n$ be a given finite sequence of events that can change the valuation of Φ and $T_e = t_{e_1} t_{e_2} \cdots t_{e_n}$ be the finite sequence of timestamps of occurrence of the events, where $n \in \mathbb{N}$. Also, let $M = m_1 m_2 \cdots m_k$ be the output finite sequence of monitor invocations and $T_m = t_{m_1} t_{m_2} \cdots t_{m_k}$ be the finite sequence of timestamps of monitor invocations, where $k \in \mathbb{N}$. We note that k is a variable to be controlled, meaning that depending upon the monitoring policy, k may change. We denote the start time of the monitor by m_0. Thus, we extend T_m as $t_{m_0} t_{m_1} t_{m_2} \cdots t_{m_k}$.

Let function *between*(τ_1, τ_2) be a function that returns all the events that occur between time τ_1 and τ_2:

$$between(\tau_1, \tau_2) = \{e_i \mid \tau_1 < t_{e_i} < \tau_2\} \tag{4.1}$$

Based on the above description, we say that the monitor is *sound* iff:

$$\forall i \in \{1 \cdots k\} : \left| between\left(t_{m_i}, t_{m_{i-1}}\right) \right| \leq B \tag{4.2}$$

which implies that at no point in time incoming events will overflow the buffer.

We formalize maximization of *memory utilization* as the following objective:

$$\max \left\{ \frac{1}{k} \sum_{i=1}^{k} \frac{\left| between\left(t_{m_i}, t_{m_{i-1}}\right) \right|}{B} \right\} \tag{4.3}$$

Thus, the objective is to maximize the average memory utilization across the complete run of the monitor.

Let $X = \{X_i \mid 1 \leq i \leq k\}$ be the set, where

$$X_i = t_{m_i} - t_{m_{i-1}}$$

i.e., each X_i is the amount of time elapsed between monitor invocations m_i and m_{i-1}. Thus, we characterize *time predictability* by the following objective:

$$\min \{V(X) \mid \text{for all possible sets of } X\} \tag{4.4}$$

where $V(X)$ is the variance of X. In other words, by minimizing the variance of all possible X's, we achieve predictability in the invocation of the monitor.

Observe that the best case minimum variance is zero, which means that for all i, $t_{m_i} - t_{m_{i-1}}$ remains constant. However, if a monitor adopts a constant monitoring frequency, it may be possible to lose soundness in a reactive system, as the rate of occurrence of events depends upon external stimuli, such as environment actions. Furthermore, for memory utilization, the best case is 100% average utilization. However, such a constraint conflicts with the time predictability requirement, since invoking the monitor whenever the buffer is full will result in a variance that is totally controlled by external actions. This discussion clearly illiterates that memory utilization and time-predictability are conflicting requirements.

Since the sequence of events to be monitored is not given a priori, an optimal monitoring policy that satisfies soundness, time predictability, and high memory utilization cannot be designed before system deployment. In other words, the times and frequency of monitor invocations have to be dynamically adjusted based on the conditions of the system under inspection. Consequently, our goal is to design a runtime *control* mechanism that enforces our objectives (i.e., Equations 4.2, 4.3, and 4.4) simultaneously through identifying T_m (i.e., time of monitor invocations and, hence, k) in a best-effort fashion.

Chapter 5

Monitor Controller Design

This chapter presents in detail the design of our controllers based on the objectives in Equations 4.2, 4.3, and 4.4. As discussed in the introduction (see Figure 1.1), the program under inspection can be multi-threaded running on a single-core machine. We instrument the program, so that it enqueues the events in a bounded-size buffer whenever they are modified. The monitor is a separate thread within the program's process, that executes at a higher priority than the program threads. It is idle for a period of time while events are being enqueued in the buffer, and once invoked, it preempts the program threads due to having a higher priority. The monitor then reads all events and verifies a set of predefined logical properties. Once the verification is complete, the monitor enters idle mode again, and awaits the refilling of the event buffer. The controller (see Figure 1.1) executes within the monitor thread. With every invocation of the monitor, the controller determines when the next invocation should occur to satisfy Equations 4.3 and 4.4. In order to maintain soundness, no events should be dropped from the buffer. Thus, when the buffer is full, the monitor invocation is automatically triggered ahead of its scheduled invocation to ensure soundness. This is called a *buffer-triggered* invocation. Subsections 5.1–5.5, describe the design of our PID and four fuzzy controllers.

5.1 PID Controller

Since we deal with reactive systems, overshoots are inevitable. In the context of our problem, an overshoot refers to the event that the buffer overflows before the monitor is invoked. Our design supports a safety threshold for buffer utilization. For instance, a controller with an 80% safety threshold will attempt to keep the buffer 80% utilized in every monitor invocation.

- **Input.** In order to achieve maximum memory utilization (Equation 4.3), the controller should target maintaining a completely full buffer up to the safety threshold at every invocation of the monitor. Thus, the input error signal to the controller is the number of empty locations in the buffer at the moment the monitor is invoked. The safety threshold is also a configuration parameter of the controller that can be altered depending upon the system requirements. Hence, the input error signal is formally the following:

  $$e(t_{m_i}) = B \times S - \left| \textit{between}\left(t_{m_i}, t_{m_{i-1}}\right) \right|$$

  where B is the buffer size, S is the safety threshold percentage, t_{m_i} is the timestamp of the current invocation of the monitor, $t_{m_{i-1}}$ is the timestamp of the last invocation of the monitor, and $\textit{between}\left(t_{m_i}, t_{m_{i-1}}\right)$ is the set of events received between the two timestamps (defined in Equation 4.1).

- **Output.** Initially the controller schedules the monitor to run after a predefined idle period. The goal of the controller is to change this initial period dynamically to maintain zero error. We refer to this period as the *polling period*, i.e. the period with which the monitor *polls* the application for new events. Thus, the output of the controller is the offset (positive or negative) with which to change the polling period to maintain zero error.

- **Tuning.** The controller is tuned using the Ziegler-Nichols method [50]for classic PID controller. According to this method, the proportional, integral and derivative gains are $0.6K_u$, $2K_p/T_u$, and $K_pT_u/8$, respectively, where T_u is the period of constant oscillation and K_u is the proportional gain at which oscillation occurs (see Figure 3.1(b)).

The controller updates are not periodic due to the fact that the period depends on the output of the controller itself, and also due to buffer triggered invocations. Thus, the integral component is calculated as in a *variable sampling period* PID [21], and not a classic PID, as described in Section 3.1.

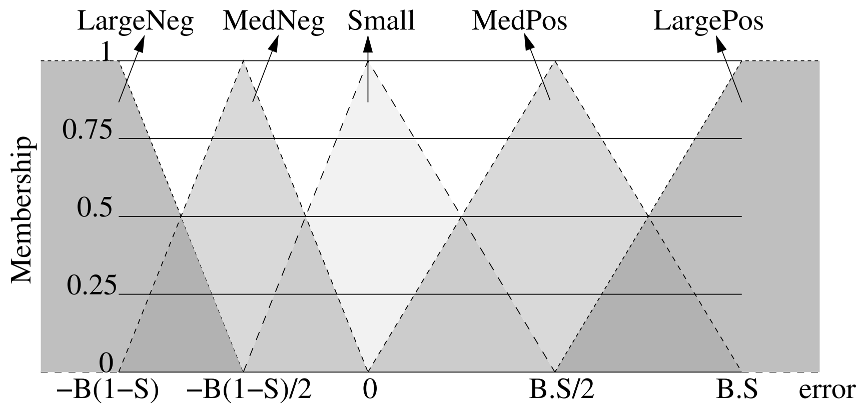


Figure 5.1: Membership functions of the error fuzzy sets.

5.2 Fuzzy Controller 1

The first fuzzy controller attempts to maximize memory utilization, similar to the PID controller.

- **Input.** The input to the controller is the fuzzy variable E_B representing the number of empty locations in the buffer. The crisp value for this variable is calculated the same way $e(t)$ is calculated in the PID controller:

  $$E_B = B \times S - \left| between\left(t_{m_i}, t_{m_{i-1}}\right)\right|$$

  There are 5 fuzzy sets for the error variable based on lambda-type functions as shown in Figure 5.1. The *Small* set has a peak at zero error, with the left x-intercept at $\frac{-B(1-S)}{2}$ and the right x-intercept at $\frac{B\times S}{2}$. The reason these points are not symmetric is that the largest positive error that could be reached is $B \times S$, which denotes that the buffer is completely empty. However, the largest negative error is $-B(1-S)$, since buffer triggering will prevent the error from exceeding that value.

- **Output.** The output of the controller is the offset value from the current polling period, which we denote as Δ_X. The membership functions for the output variable are standard lambda-type functions similar to those in Figure 3.3, with centers at -1, -0.5, 0, 0.5, and 1, respectively. The output from defuzzification is a percentage, which is then converted to absolute Δ_X multiplied by a factor depending on the unit and allowed range of polling period for the given system.

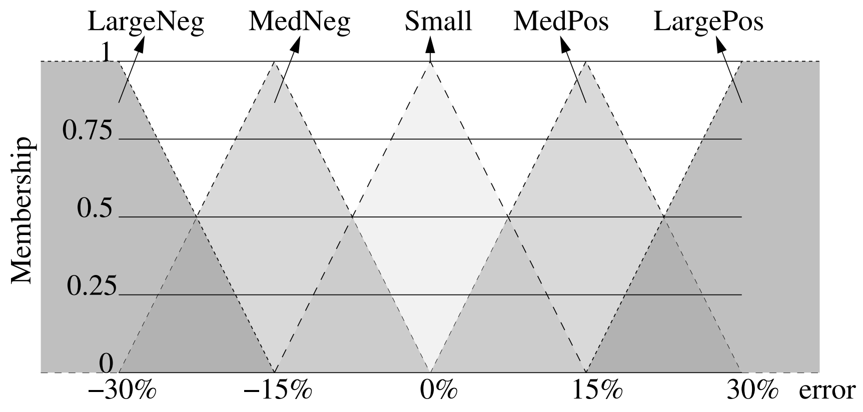


Figure 5.2: Membership functions of $E_{\bar{X}}$

- **If-then rules.** The *if-then* rules for the controller are as follows:
  - if E_B is LargeNeg, Δ_X is LargeNeg
  - if E_B is MedNeg, Δ_X is MedNeg
  - if E_B is Small, Δ_X is Small
  - if E_B is MedPos, Δ_X is MedPos
  - if E_B is LargePos, Δ_X is LargePos
- **Fuzzification, inference, and defuzzification.** The fuzzification module uses scaled inference and the inference engine uses individual rule based firing. The defuzzification module uses the center of gravity method to calculate the output value. The calculations involved in applying these methods are minimal, with the advantage that most of the calculations can be precomputed before the system executes, thus decreasing the processing overhead of the controller in run time.

5.3 Fuzzy Controller 2

Fuzzy controller 2 targets both memory utilization and time predictability. The approach of this controller is to balance between choosing a polling period that would minimize the error in the buffer, and choosing a polling period of a value as close as possible to the mean of all previous polling periods. The second condition ensures that the variance of the polling period is minimized.

- **Input.** In addition to E_B, we introduce a new fuzzy variable $E_{\bar{X}}$ to control the polling period variance. $E_{\bar{X}}$ represents the difference between the current polling period and the mean of all previous polling periods. The crisp values of $E_{\bar{X}}$ is calculated as follows:

$$E_{\bar{X}} = \frac{X - \bar{X}}{\bar{X}} \tag{5.1}$$

  where X is the current polling period and $\bar{X}$ is the mean of all previous polling periods. $E_{\bar{X}}$ is a percentage so as to make the controller computations independent of the time scale at which the system operates. The membership functions for this variable are standard lambda-type, as shown in Figure 5.2. These values are configuration parameters and can be changed according to the user requirement. The choice of the range -30% to 30% produces low variation in polling periods, and consequently high time predictability.

- **Output.** The output of the controller is the same as Fuzzy 1 (i.e., the offset value from the current polling period).

- **If-then rules.** Since the controller is now targeting two simultaneous goals involving two fuzzy variables (E_B and $E_{\bar{X}}$), with 5 fuzzy sets each, there are 25 possible if-then rules. Table 5.1 shows the consequent fuzzy set of each rule based on the combination of the two antecedent fuzzy sets, where the columns are $E_{\bar{X}}$ fuzzy sets, the rows are E_B fuzzy sets, and LN, MN, S, MP, and LP are abbreviations of LargeNeg, MedNeg, Small, MedPos, and LargePos, respectively. The mapping above is symmetric, meaning that no variable has a more significant effect on the output than the other; i.e., both are equally contributing to the decision made by the controller. This mapping is a configuration parameter and could be changed according to the system requirements.

5.4 Fuzzy Controller 3

Instead of minimizing the variance, fuzzy controller 3 attempts to maintain an upper bound on the variance. Thus, this controller adds a configuration parameter to fix that upper bound. However, since the mean of polling period is not known a priori and changes during the program's execution, the value that the user chooses as an upper bound on the variance does not represent the actual variation in the polling period. For instance, a variance of 10 for a polling period mean of 1000 is an indicator for very high predictability and low

		$E_{\bar{X}}$ *Fuzzy sets*				
		LN	MN	S	MP	LP
	LN	S	MN	LN	LN	LN
E_B	MN	MP	S	MN	LN	LN
Fuzzy	S	LP	MP	S	MN	LN
sets	MP	LP	LP	MP	S	MN
	LP	LP	LP	LP	MP	S

Table 5.1: Symmetric mapping of input variables in if-then rules.

variation. However, the same variance when the mean is 10 shows very high variation in polling times. This has led to using the coefficient of variation as the metric that has an upper bound. The coefficient of variation is calculated as $c_v = \sigma_X / \bar{X}$, where σ_X is the standard deviation of all previous polling periods, and $\bar{X}$ is the mean. Since the polling period mean will never be zero, c_v is a safe metric. The coefficient of variation enables the user to dictate the required shape of the distribution of polling periods; i.e., whether to have a broad or narrow curve around the mean.

Fuzzy controller 3 adopts a fuzzy variable E_{c_v} which is simply the last polling period of the controller. Let all polling periods since the start of execution be the sequence $X = X_1 X_2 X_3 \cdots X_N$, where X_N is the last polling period. Since the upper bound of c_v is fixed by a constant k, the controller needs to determine the best X_{N+1} that guarantees k as the coefficient of variation. We expand the coefficient of variation formula, so that we can obtain the value of X_{N+1}. This led to deriving a quadratic equation whose roots are the values for X_{N+1} that produce $c_v = k$. To simplify the equation, we define γ as the following quantity:

$$\gamma = \left(\frac{N + 1 + k^2 N}{N \left(N+1\right)^2} \right) \tag{5.2}$$

where N is the number of polling periods in the sequence X. The quadratic equation to calculate X_{N+1} is as follows:

$$\left(\frac{1}{N} - \gamma \right) X_{N+1}^2 - \left(2\,\gamma \sum_{i=1}^{N} X_i \right) X_{N+1} + \left(\frac{1}{N} \sum_{i=1}^{N} X_i^2 - \gamma \sum_{i=1}^{N} X_i \right) = 0 \tag{5.3}$$

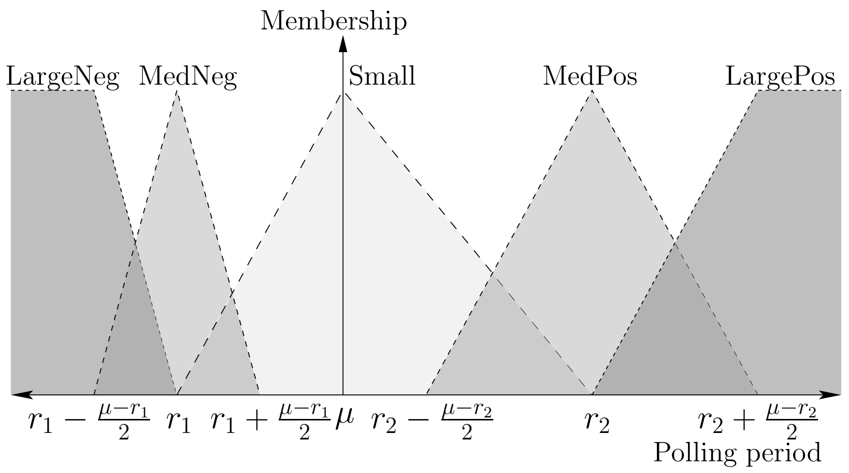

Membership
LargeNeg
MedNeg
Small
MedPos
LargePos
$r_1 - \frac{\mu - r_1}{2}$
r_1
$r_1 + \frac{\mu - r_1}{2}$
μ
$r_2 - \frac{\mu - r_2}{2}$
r_2
$r_2 + \frac{\mu - r_2}{2}$
Polling period

Figure 5.3: Membership functions of E_{c_v}

If Equation 5.3 has complex roots, then it is not possible for X_{N+1} to lower the coefficient of variation down to k. In this case, fuzzy controller 3 falls back to Fuzzy 2, attempting to minimize the variance all together. This will continue until the coefficient of variation is low enough that it can be controlled within the upper bound.

If the two roots of Equation 5.3 are real values, the mean is a number between these two roots. The membership functions for E_{c_v} are designed in such a way that it tries to keep the polling period between the two roots, with preference to the mean. Figure 5.3 shows how these functions are defined, where r_1 and r_2 are the roots of Equation 5.3. The *Small* membership function has a peak at the mean μ, has a left x-intercept at r_1, and a right x-intercept at r_2. *MediumNeg* and *MediumPos* are centered around r_1 and r_2 with intercepts at half the distance between the mean and the roots. This maintains fairness in treating the polling period regardless of which root it is closer to.

The mapping in the *if-then* rules in this controller is similar to the mapping in Fuzzy 2 as shown in Table 5.1, which maintains a balanced trade-off between memory utilization and time predictability.

5.5 Fuzzy Controller 4

Fuzzy controller 4 is essentially the same as Fuzzy 3, with the exception of the mapping for the *if-then* rules. In this controller, the mapping gives preference to controlling mem-

		E_{c_v} Fuzzy sets				
		LN	MN	S	MP	LP
	LN	MN	LN	LN	LN	LN
E_B	MN	S	MN	MN	LN	LN
Fuzzy	S	LP	MP	S	MN	LN
sets	MP	LP	LP	MP	S	MN
	LP	LP	LP	LP	MP	S

Table 5.2: Asymmetric mapping of input variables in if-then rules.

ory utilization when E_B is a large negative value, even if that contradicts with the time predictability requirement. Table 5.2 shows the modified mapping. This mapping enables Fuzzy 4 to react faster to large overshoots in the error, thereby maintaining stability and giving room for the controller to work on a balanced trade-off.

Chapter 6

Experiment Design

In order to analyze the performance of our controllers, we have conducted two case studies: (1) a Bluetooth mobile payment, and (2) a Laser beam stabilizer for aircraft tracking. Each case study involves using different controllers with different configurations.

Our experiments are designed based on three factors:

1. **Controller type.** We incorporate seven controllers in our experiments: PID, Fuzzy 1, Fuzzy 2, Fuzzy 3 with target coefficient of variation $c_v = 0.4$, Fuzzy 3 with $c_v = 0.2$, Fuzzy 4 with $c_v = 0.4$, and Fuzzy 4 with $c_v = 0.2$.
2. **Buffer size (B).** We experiment with three different buffer sizes: 20, 40, and 60 events.
3. **Safety threshold (S).** We experiment with two safety thresholds: 80%, and 90%.

Hence, there is a total of 42 configurations to test all different combinations of the above three factors. For both case studies, we carried out multiple runs with randomization to provide statistical confidence and remove any hidden effects. The five measurement metrics that we observe are:

1. **Error mean.** This is the mean number of empty buffer locations at every invocation of the monitor. This value is a measure of the memory utilization of the monitor, i.e. the lower the value, the more utilized the memory.

2. **Polling period coefficient of variation (C_v).** This value is a measure of time predictability, i.e. the lower the value, the closer polling periods are to their mean, and hence, more time predictable.

3. **Context switches.** This is the number of invocations of the monitor during a run of an experiment. This value is a measure of the overhead introduced by the monitor; i.e. the lower the value, the lesser the context switching between threads, and thus the less overhead.

4. **Buffer triggers.** This is the number of buffer-triggered monitor invocations. This value is a measure of the quality of the controller in the sense that a well designed controller should not overshoot frequently causing many buffer triggers.

Chapter 7

Case Studies

7.1 Bluetooth Mobile Payment (BTP)

Mobile payment is becoming increasingly popular and gaining assurance about the soundness of such a system is an essential requirement. Whether payment is through WiFi, Bluetooth, or NFC, the process relies on a payment hub that communicates with smartphones to process payments, which includes establishing a connection with devices. Verification of such properties is a necessity in a payment system, specially since it is commonly the case that more than one payment will need to be processed at the same time. The hub establishes a connection with these devices and sends/receives messages. We monitor these messages at the operating system level to ensure that every message gets a response and no error.

Our experimental platform is single core machines running under the QNX real-time operating system hosting a Bluetooth 2.1 adapter. Our implementation follows the outline in Figure 1.1, with the exception that there is a single program thread responsible for extracting events using the QNX TraceEvent API and queuing them into the buffer. We use an experimental dataset that has been collected in a shopping mall [22]. It includes Bluetooth contact traces from employee devices around the cashier area of a certain store. To provide statistical confidence in the results, we run 9 replicates of a trial, where each trial consists of running all 42 possible combinations of the experimental factors.

The setup includes 14 virtual machines running the real-time operating system QNX, capable of running 14 experiments of different configurations in parallel. QNX is a reasonable option for a payment hub since many mobile devices (e.g., Blackberry 10) use

QNX and it provides accurate timing behavior and scheduling. We use VMWare ESXi 5.1 to host virtual machine. VMWare ESXi provides near to native performance, and CPU performance is more optimized when core affinity is configured [29]. Thus, for every virtual machine we configure core affinity, so that a core is exclusive to that machine. Every virtual machine has its own physical Bluetooth adapter. These virtual machines run on three physical Core i7 machines, each with 16GB of RAM, and 2TB storage.

In all the figures showing results of the legends on horizontal axis are as follows:

- F2, Δ_X is LargeNeg
- if E_B is MedNeg, Δ_X is MedNeg
- if E_B is Small, Δ_X is Small
- if E_B is MedPos, Δ_X is MedPos
- if E_B is LargePos, Δ_X is LargePos

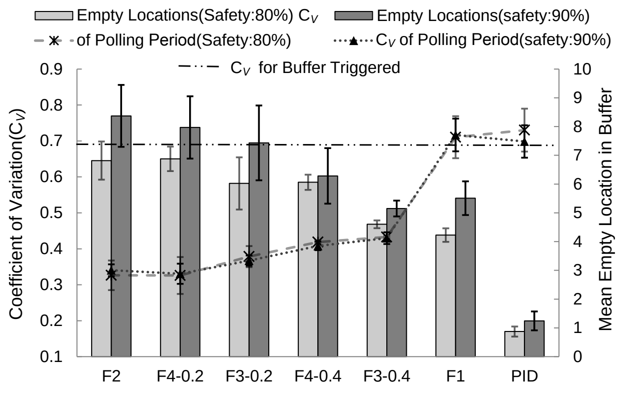


Figure 7.1: Polling period coefficient of variation vs. error mean at buffer size 20 for BTP

Analysis of Time Predictability

Figure 7.1, 7.2, and 7.3 show the average polling period coefficient of variation C_v across all 9 replicates for buffer sizes 20, 40, and 60. As can be seen, Fuzzy 2 exhibits the lowest C_v, since it is designed to control the polling period within $\pm 15\%$ of the mean (see Section 5.3). Fuzzy 3 targets $C_v = 0.2$ (denoted F3-0.2 in the figure) and Fuzzy 4-0.2 show low C_v due to having an aggressive $C_v = 0.2$ goal. In Figure 7.1, Fuzzy 3-0.2 and Fuzzy 4-0.2 fail

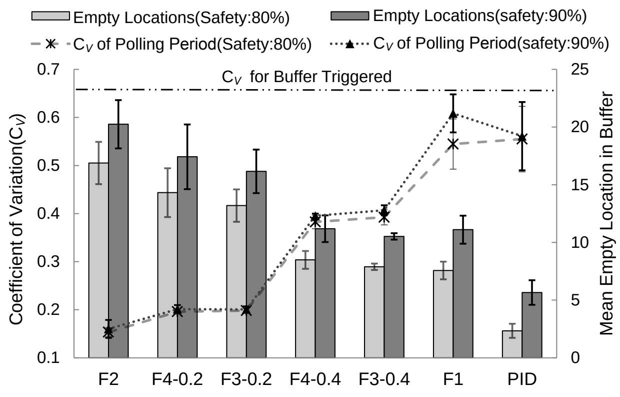


Figure 7.2: Polling period coefficient of variation vs. error mean at buffer size 40 for BTP

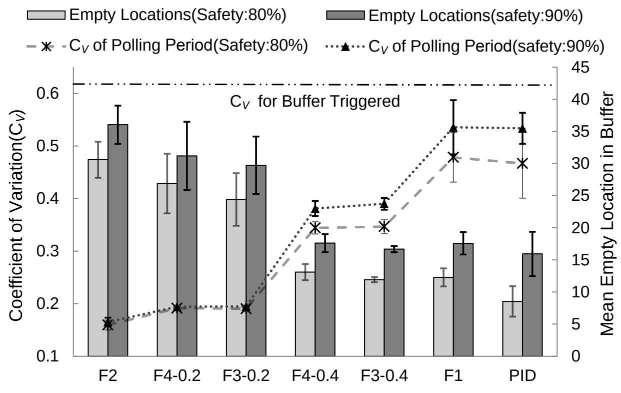


Figure 7.3: Polling period coefficient of variation vs. error mean at buffer size 60 for BTP

to meet their goals, scoring a C_v of 0.32 and 0.37. This is due to the 0.2 goal being too aggressive to reach in a buffer of size 20. Note that at higher buffer sizes, these controllers meet their goals, as shown in Figures 7.2 and 7.3). However, Fuzzy 3-0.4 and Fuzzy 4-0.4 consistently meet their goal ($C_v = 0.4$) across all configurations. Since Fuzzy 1 and PID do not attempt to control C_v, they have the highest values.

An interesting observation is that for a purely buffer triggered implementation, where no control is involved, the C_v is almost always higher than any controller across all configurations (shown as a horizontal line in all three graphs). In fact, for Fuzzy 2, C_v is less than a third of pure buffer triggered for buffer size 40. This shows the advantage of using controllers to improve time predictability of the monitoring system.

Figure 7.4 shows the box-plots of the polling periods for different controllers for buffer size $B = 20$ and safety threshold $S = 80$. The figure shows that a purely buffer triggered

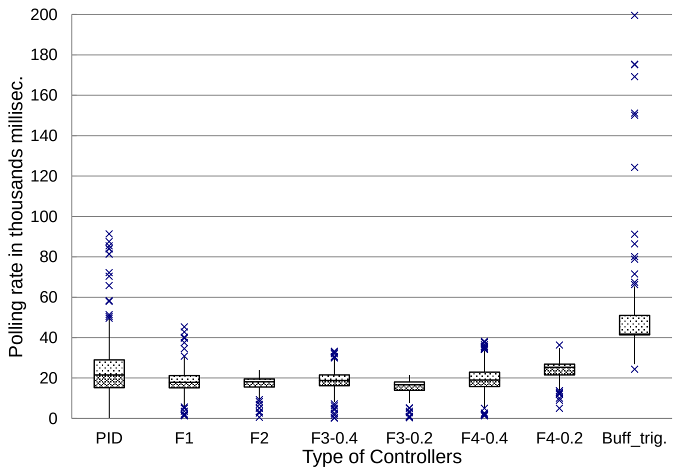


Figure 7.4: Box-plot of polling periods for different controllers of BTP.

implementation exhibits the highest variability. This is expected since this implementation responds transparently to the non-linearity of the system. The second highest variability is present in the PID controller, explained by the inability of the PID to adapt to a non-linear system. Again, it can be seen that using Fuzzy 1, which has the same goal as the PID, can drastically improve the stability of the controller. The lowest variability is - as expected - due to Fuzzy 2, Fuzzy 3-0.2, and Fuzzy 4-0.2.

Memory Utilization

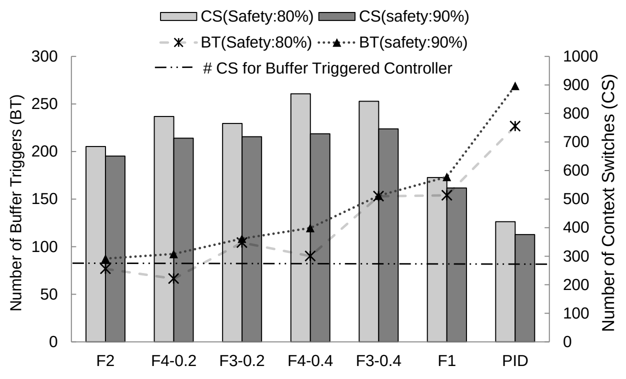


Figure 7.5: Number of buffer triggers vs. number of context switches at buffer size 20 for BTP

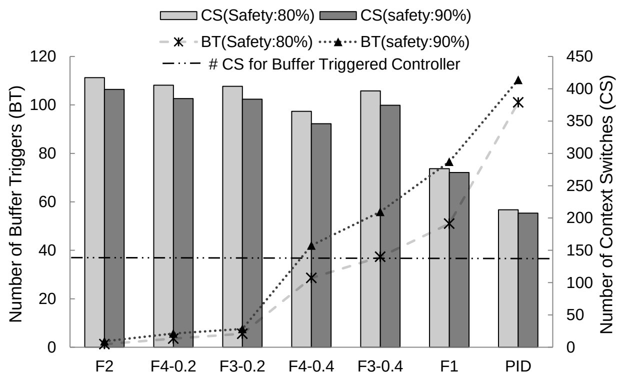


Figure 7.6: Number of buffer triggers vs. number of context switches at buffer size 40 for BTP

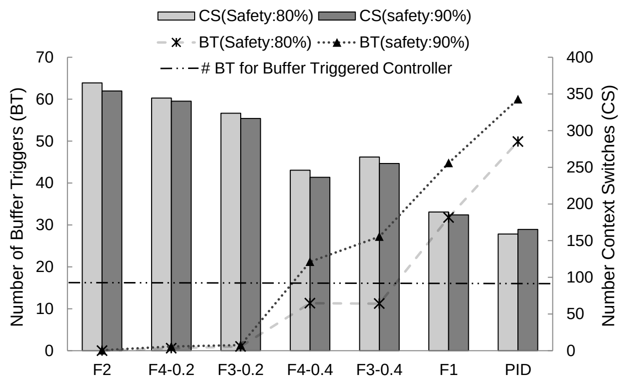


Figure 7.7: Number of buffer triggers vs. number of context switches at buffer size 60 for BTP

Figure 7.1, 7.2, and 7.3 show the mean number of empty buffer locations (error) across all 9 replicates for buffer sizes 20, 40, and 60. The 95% confidence intervals for the error mean are also shown. As can be seen, the PID controller consistently has the lowest error mean, and thus provides the highest memory utilization. The error mean for Fuzzy 1 is also low, and comparable to that of the PID when the buffer size increases (see Figures 7.2 and 7.3). Fuzzy 2 exhibits a consistently high error mean. This is due to the fact that Fuzzy 2 is designed to be aggressive in maintaining a low polling period C_v, which comes at the cost of error. This also applies to Fuzzy 3 aggressively targeting $C_v = 0.2$ (denoted F3-0.2) and Fuzzy 4 targeting $C_v = 0.2$. However, Fuzzy 3-0.4 and Fuzzy 4-0.4 perform comparably to PID and Fuzzy 1, especially with increased buffer size. This stems from the fact that these controllers have a relaxed goal (i.e., $C_v = 0.4$) and are thus more capable of maintaining a low error mean. The error mean of a 90% safety threshold controller is consistently higher than that of 80% simply due to having more space to control in the buffer.

The error mean trend is further clarified in Figure 7.5, 7.5, and 7.5. This figure shows the number of buffer triggers occurred for every controller. It appears that the reason PID has such a low error mean is because it consistently has the highest number of buffer triggers. This is an indication that the PID controller is unable to adapt to the non-linear nature of the system, and as a result is overshooting considerably more than any other controller. This also shows that Fuzzy 1, although having a slightly higher error mean, is more capable of adapting to the change in the system without frequently overshooting. The other fuzzy controllers have a low number of buffer triggers due to their tendency to remain stable.

Execution Time

We next study the effect of using different controllers on the execution time of the program. The execution time includes the CPU time, time of kernel calls, CPU time by child processes, and time of kernel calls made by child processes. We compare this time to the execution time of the program without any monitoring functionality. Note that for this comparison, there is no verification overhead included in the calculation. We assume that the verification overhead can be offloaded to a seperate processing unit.

Since execution time results are subject to many factors affecting variability, we attempt to estimate the worst case overhead introduced by our controllers based on the maximum execution time of the program with our controllers relative to the minimum execution time of the program without any monitoring. This comparison shows that in the worst case, PID and F1 introduce a 19% increase in execution time. However, other fuzzy controllers average around 10%.

Additional Observations

- **Thread context switching.** A high number of buffer triggers indicates that the system is overshooting frequently and, thus, is more frequently filling the buffer completely. This results in a lower number of context switching. Figure 7.5, 7.5, and 7.5 illustrate the number of context switches and a trend that is related to the number of buffer triggers. The figure also shows a horizontal line denoting the number of context switches performed by a purely buffer triggered solution, which is expected to be the lower than any controller-based approach.

- **Time-predictability vs. memory utilization.** The trend of polling period coefficient of variation C_v versus error mean magnifies the trade-off between time predictability and memory utilization. The results show that Fuzzy 3-0.4 and Fuzzy 4-0.4 exhibit the best balance between the two goals consistently across configurations.

- **Resilience to overshoots.** Figure 7.5, 7.5, and 7.5 show that all fuzzy 4 controllers present an advantage over fuzzy 3 in terms of number of buffer triggers. Since these controllers are designed to be more aggressive when an overshoot occurs or is about to occur, their behavior demonstrates a more conservative approach with respect to buffer triggers.

7.2 Laser Beam Stabilization (LBS)

LBS technology is used in aircraft targeting, surveillance and laser-based communication systems. A control system stabilizing a laser beam is required to maintain safety properties, such as ensuring that the offset of the laser from the target should not exceed a certain value. In this case study, we use the Quanser laser beam stabilization system with a mounted motor that produces undesirable vibrations affecting the stability of the laser. When the photodetector registers the laser at an offset larger than 0.01mm, an event is queued into the buffer. Our experiments are based on 9 replicates and we target $C_v = 0.6$ for Fuzzy 3 and 4 controllers. This demonstrates how a more relaxed constraint affects the response of the controller.

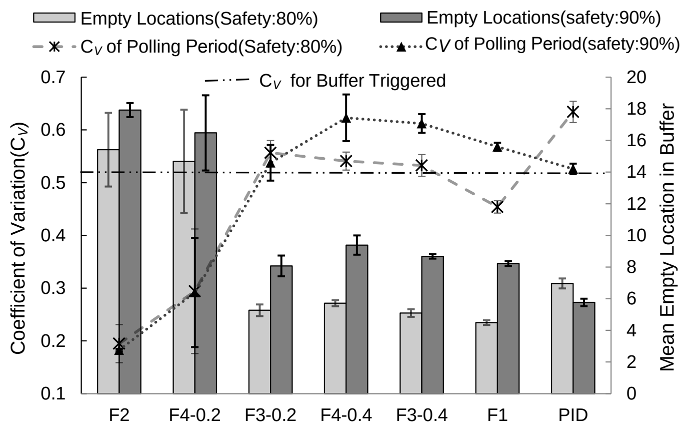


Figure 7.8: Polling period coefficient of variation vs. error mean at buffer size 40 for LBS

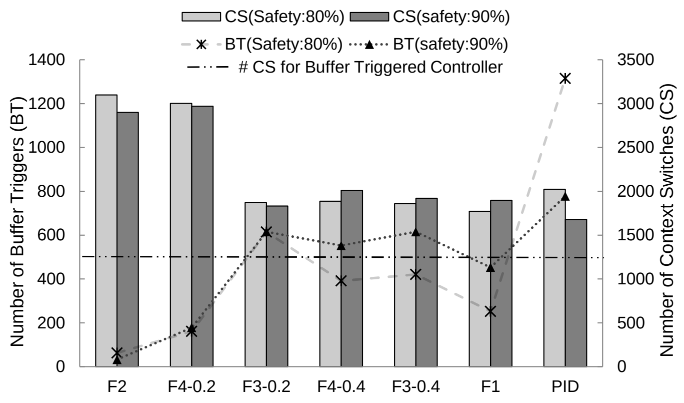


Figure 7.9: Number of buffer triggers vs. number of context switches at buffer size 40 for LBS

Time predictability

Figure 7.8 and 7.9 show the results of the experiments on buffer size of 40. The trend of C_v for polling period versus error mean in Figure 7.8 is similar to that of the Bluetooth experiment. Fuzzy 3-0.2 scores a much higher C_v than its goal (0.6 vs. a goal of 0.2). However, Fuzzy 4-0.2 is closer to its goal, achieving $C_v = 0.3$. This is due to the periodic nature of the oscillations introduced by the motor, which coupled with the aggressiveness of Fuzzy 4 at high errors, enables it more quickly to reach low error and focus on controlling the coefficient of variation. An interesting observation is that C_v of a purely buffer triggered implementation is on average 0.59, which is less than all controllers except for Fuzzy 4-0.2 and Fuzzy 2. This is due to the periodic nature of the events, which enables a purely buffer-triggered solution to naturally produce a lower C_v. Fuzzy 2 and fuzzy 4-0.2, however, are more aggressive in maintaining a low C_v and, thus, they outperform pure buffer triggered.

Memory Utilization

Figure 7.8 shows that low C_v comes at the cost of the error mean. This is contrasted with the number of buffer triggers in Figure 7.9, which shows that PID has the highest number.

Additional Observations

- **Tuning cost.** In Figure 7.9, the difference between the number of buffer triggers for PID when safety is 80% versus 90% is large. This is due to the sensitivity of PID controllers to tuning. Compared to Fuzzy 1 which attempts the same objective, Fuzzy 1 appears to be more consistent. This is further supported by our results for different buffer sizes not shown here.

- **Controller instability.** In Figure 7.8, the trend of C_v for 80% versus 90% safety thresholds is reversed for PID. This is due to the instability of the PID, causing it to revert more to buffer triggers (see Figure 7.9). This causes it to actually produce a lower C_v at 90% because, in that case, it is closer to a pure buffer triggered controller. This is why the resulting C_v is almost the same as that of pure buffer triggered.

Chapter 8

Preceding Work on Runtime Monitoring

In this chapter, we present the work done on runtime monitoring and verification preceding the main problem and its solution presented in this thesis. This includes work on two problems: (1) design and analysis of marking schemes to lower overhead in *sampling-based* execution monitoring and tracing, and (2) reducing monitoring overhead by leveraging the benefits of *event-triggered* and *time-triggered* runtime verification approaches for *hybrid runtime verification*. Section 8.1 and 8.2 briefly discuss work done on these two problems respectively.

8.1 Sampling-based Execution Tracing and Monitoring

Tracing and monitoring of program execution is a commonly used approach to debug real-time systems where stepping through program execution usually violates deadline constrains. However, the applicability of tracing to debug real-time system is severely affected by the nature of overhead that a tracing method imposes on the system. The nature of overhead depends on requirement of CPU cycles and memory for tracing, and the temporal distribution of these overheads across program execution. To make the tracing and monitoring a more effective method of debugging real-time systems, Fischmeister et al. [20] propose a sampling-based execution monitoring approach. According to this approach,

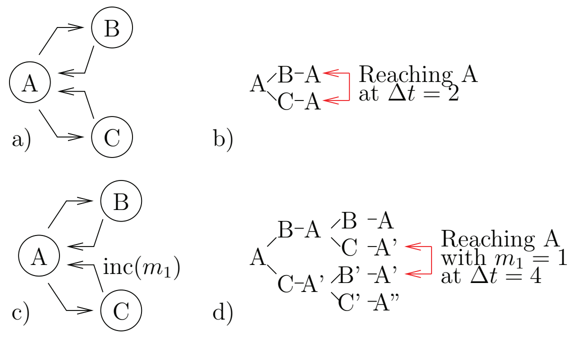


Figure 8.1: Example of a single instrumentation to extend Δt [48].

a monitor, which is a part of the system along with the executing program, takes periodic samples of the program state and then uses these samples to correctly construct the execution path taken by the program. Periodic intervention of monitor ensures periodic monitoring overhead. However, in absence of any instrumentation, the program monitor has to intervene frequently to ensure correct reconstruction of execution path of the program. This will result into a high overhead. Hence the key problem in sampling-based execution monitoring is to increase the sampling period of the monitor. To illustrate it further, Figure 8.1(a) shows control flow graph of a simple program. The node A, B and C represent three basic blocks of the program with best-case execution time of one unit each (the developer has to consider best-case execution time to ensure the correctness of the monitor). Figure 8.1(b) shows that the maximum sampling period $(\Delta)t$ for this program is two units. This is because, after two time units, both execution paths of the program will be in identical states, and the monitor can not identify the path taken by the program. Figure 8.1(c) shows the same program instrumented with an increment marker m_1, which increases every time program execution goes from node C to node A. Figure 8.1(d) explains that the monitor can uniquely identify the execution path of the program by looking at the value of marker m_1 at $t = 4$. This increases the sampling period by 2 units. Fischmeister et al. [20] show that sampling period can be increased drastically by inserting markers in the program and including them in the sample for execution path identification. Effectiveness of an instrumentation depends on: (a) placement of makers, and (b) how the marker is manipulated. To this end, in [48], we design two new marking schemes namely *assignment* and *bit vector* schemes. We compare the expressiveness of these schemes among themselves as well as with other existing schemes.

8.1.1 Overview of Different Marking Schemes

1. *Single Increment Scheme.* The marker function, I, for single increment scheme is defined as $I(m) = m + 1$ for marker m (proposed in [20]).

2. *Multiple Increment Scheme.* Same as single increment scheme except that marker can be increased multiple times in different nodes.

3. *Assignment Scheme.* In this scheme marker is a variable and can be assigned any value; i.e., $I(m) = k$ where is k is an arbitrary constant that might change across nodes.

4. *Bit Vector Scheme.* In this scheme marker a bit vector, whose bit fields are initialized to zero. Marker function can set and clear bit at any specific location in the bit field.

8.1.2 Expressiveness Comparison Results

In [48], with the help of concrete examples, we compare the expressiveness of these marking schemes and show their strength and weakness. Naturally, single increment is less expressive as compared to multiple increment scheme because the later can always emulate the former, but not vice versa. Similarly, assignment is less expressive as compared to bit vector. The strength of increment-based marking schemes is that these can keep the history for long time through increment of marker (depending upon the size of counter variable). Whereas, the other two scheme, assignment and bit vector, can not keep the history because every assignment erases erases previous changes. On the other hand, increment based markers are ineffective when the two execution paths are permutation of same nodes. In such a case, result of increment schemes will be same at the end of both paths. Assignment and bit vector based schemes can solve this problem by assigning different values or setting different bits on the two paths. In [48], with the help of concrete examples, it is argued that bit vector is more powerful than assignment scheme.

After understanding the strengths and weaknesses of these schemes, we design an auto-instrumentation algorithm and establish its termination conditions. Another hybrid scheme was designed by combining bit vector scheme and increment scheme. Experimentation results show that bit vector can almost double the sampling period while keeping number of instrumentation at the same level.

8.2 Hybrid Runtime Verification

Event-triggered (ET) and time-triggered (TT) are two existing approaches commonly used to invoke monitor in a runtime verification system. In *event-triggered* , program immediately invokes the monitor for verification every time state of the program changes (or a critical event happens), whereas in *time-triggered* monitor periodically preempts the program and verifies the correctness properties. The period, known as *sampling period* (SP) is set in such a way that the monitor does not miss any critical event. Work done by Borzoo et al. [10] shows that time-triggered monitoring can potentially reduce the runtime overhead if the program state changes frequently and monitor samples the program at a low frequency. However, if critical events happens sparsely then time-triggered monitor takes redundant samples (i.e., a sample taken without any change in program state since last sample) and imposes higher monitoring overhead as compared to event-triggered monitoring. With this motivation, in [49], we propose a control flow graph (CFG) based static analysis technique, called hybrid runtime verification (HyRV). This approach exploits benefits of both event-triggered runtime verification (ETRV) and time-triggered runtime verification (TTRV) to reduce the runtime overhead. The switching between ET and TT happens with the help of *switches* placed at appropriate locations in the program. However, switching itself imposes some amount of overhead which needs to be considered while formulating a solution. To simplify this problem we divide runtime monitoring overheads into the following five elementary costs:

- 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 (sampling cost)
- $c_{E \to T}$: cost of a switch from ET mode to TT mode
- $c_{T \to E}$: cost of a switch from TT mode to ET mode

Formally, given the above five costs for a monitoring environment and a program to monitor, we assign each basic block of CFG of given program to either ET or TT such that the overall cost of monitoring (i.e., either c_{ET} or c_{hist} for all critical events, c_{TT} for all samples, and costs of switching between the two modes) is minimum. We believe that finding an optimal solution to this problem is intractable. Hence, we formulate an *integer linear program* (ILP) heuristic to solve this problem. Our optimization problem (or ILP) aims to minimize the total monitoring cost of the program with three constraints: (1) every critical event is monitored either by ET or TT, (2) there is an appropriate switch between two consecutive critical events which are monitored with different monitoring methods, and (3) depending upon longest sampling period [10] and use of TT mode, correct number of samples are taken.

SNU Real-time benchmark suite [1] is used to empirically test the validity of HyRV idea. To this end, we build a toolchain to automate the whole process. The tool chain uses static analysis tools **Clang** and **llvm** [34] for computation of program's CFG and execution time of each basic block of resulting CFG. CodeSurfer [24] is used to determine the location of the critical events. Next part of the tool chain is an ILP model generator that takes CFG along with execution time estimation and the five elementary monitoring costs (six different configurations were used), and produces the optimization problem. We use Yices [43] to solve the optimization problem. The solution (instrumentation scheme) along with the original program is then passed to a script which instruments the program and runs it on Keil μVision simulator.

Our experiments show three type of results:

1. For **crc** program, the ILP model suggests a hybrid monitor and this monitor significantly outperforms an ET or TT monitor.

2. For `bs`, `fibcall`, `insertsort`, and `matmult` programs, the ILP model suggests either an ET or TT monitor and the suggested solution outperforms other monitoring modes.

3. For `fir` program, the ILP model suggests monitoring modes that either exhibits slight improvement over other monitoring modes or slightly underperforms in practice. We think the reason for this lies in the use of heuristic to find a solution.

These results validate our belief that hybrid runtime verification can be used to reduce monitoring overhead.

Chapter 9

Conclusion

Gaining assurance about the correctness of embedded systems has always been an active and challenging area of research in computing technology. In this work, we concentrated on designing a scalable approach for runtime verification of reactive embedded systems with three objectives: soundness, minimum jitter in monitor invocation frequency, and maximum memory utilization. To this end, we leveraged the rich literature of control theory. In particular, we designed a PID and four different fuzzy controllers, each targeting a different set of objectives. Our experiments on two embedded systems (a laser beam stabilizer (LBS) and a Bluetooth mobile payment (BTP) system) show that our controller-based approach is quite effective and scalable with minimal runtime overhead. In particular, we observed that for reactive systems, where the environment stimuli occur non-linearly, a fuzzy controller reacts the best with respect to achieving time-predictability in monitor invocations. Fuzzy controllers have an added advantage over PID that their design is system-independent. For example, Fuzzy controllers designed for LBS can be used directly for the BTP case study as well with only minor modifications. However, the PID controller needs much more elaborate tuning to adapt it for the BTP system. Among the four fuzzy controllers, the fuzzy controllers that attempt to maintain an upper bound on the variance of monitor invocation frequency, in most cases, provide the best balance between time-predictability and memory utilization. If users have no knowledge about the coefficient of variance of monitor invocation frequency then we suggest the use of Fuzzy 2 to get an rough estimate about the variance in the system and then define an appropriate target. The average runtime overhead of our approach is around 10%. This makes our scheme suited for practical applications in resource constraint embedded systems.

Chapter 10

FutureWork

For future work, we are planing to change our controllers to handle bimodal distribution of critical events as well. Currently our controller designs do not factor in the bimodal behaviour exhibited by many embedded systems. For example, a system deployed to detect rabbit intrusion into a park will be active during during particular hours in a day and in sleep mode for rest of time. For such system targeting variance minimization will not produce optimal results and controller should able to detect the mode of operation and set the goal accordingly. We are also planning to investigate employing static analysis techniques such as symbolic execution, so our controllers are also aware of the structure of the system under inspection. Another interesting research direction is to design controllers for monitoring distributed embedded systems.

References

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

[2] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Verification, Model Checking, and Abstract Interpretation*, pages 44–57. Springer, 2004.

[3] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Proceedings of the 7th international conference on Runtime verification*, RV'07, pages 126–138, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] Manuel Blum and Sampath Kannan. Designing programs that check their work. *J. ACM*, 42(1):269–291, January 1995.

[5] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *Proceedings of the 21st European conference on Object-Oriented Programming*, ECOOP'07, pages 525–549, Berlin, Heidelberg, 2007. Springer-Verlag.

[6] Eric Bodden. A lightweight LTL runtime verification tool for java. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 306–307, New York, NY, USA, 2004. ACM.

[7] Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 5–14, New York, NY, USA, 2010. ACM.

[8] Eric Bodden and Laurie Hendren. The clara framework for hybrid typestate analysis. *Int. J. Softw. Tools Technol. Transf.*, 14(3):307–326, June 2012.

[9] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. Sampling-based runtime verification. In *Proceedings of the 17th international conference on Formal Methods*, FM'11, pages 88–102, Berlin, Heidelberg, 2011. Springer-Verlag.

[10] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. Time-triggered runtime verification. *Formal Methods in System Design*, 43(1):29–60, 2013.

[11] Borzoo Bonakdarpour, Johnson J. Thomas, and Sebastian Fischmeister. Time-Triggered Program Self-Monitoring. In *Proceedings of the 2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '12, pages 260–269, Washington, DC, USA, 2012. IEEE Computer Society.

[12] Allen R. Bonde, Jr. and Sumit Ghosh. A comparative study of fuzzy versus Fixed thresholds for robust queue management in cell-switching networks. *IEEE/ACM Trans. Netw.*, 2(4):337–344, August 1994.

[13] Robert N. Charette. Why software fails [software failure]. *IEEE Spectr.*, 42(9):42–49, September 2005.

[14] Robert N. Charette. This car runs on code. *IEEE Spectrum*, 46(3):3, 2009.

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

[16] M. d'Amorim and K. Havelund. Event-based runtime verification of java programs. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, May 2005.

[17] M. d'Amorim and G. Roşu. Efficient monitoring of ω-languages. In *Proceedings of the 17th international conference on Computer Aided Verification*, CAV'05, pages 364–378, Berlin, Heidelberg, 2005. Springer-Verlag.

[18] D. Driankov, H. Hellendoorn, and W. Reinfrank. *An introduction to fuzzy control.* Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[19] Matthew B. Dwyer, Alex Kinneer, and Sebastian Elbaum. Adaptive online program analysis. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 220–229, Washington, DC, USA, 2007. IEEE Computer Society.

[20] Sebastian Fischmeister and Yanmeng Ba. Sampling-based program execution monitoring. *SIGPLAN Not.*, 45(4):133–142, April 2010.

[21] Peter Galan. Temperature control based on traditional PID versus fuzzy controllers. *Nortel Networks Control Software Design Documentation*.

[22] A. Galati and C. Greenhalgh. Human mobility in shopping mall environments. In *Proceedings of the Second International Workshop on Mobile Opportunistic Networking*, pages 1–7. ACM, 2010.

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

[24] GrammaTech Inc. CodeSurfer®. http://www.grammatech.com/products/codesurfer/.

[25] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Syst. J.*, 41(1):4–12, January 2002.

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

[27] Klaus Havelund and Allen Goldberg. Verify your Runs. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 374–383. Springer Berlin Heidelberg, 2008.

[28] Xiaowan Huang, Justin Seyster, Sean Callanan, Ketan Dixit, Radu Grosu, ScottA. Smolka, ScottD. Stoller, and Erez Zadok. Software monitoring with controllable overhead. *International Journal on Software Tools for Technology Transfer*, 14(3):327–347, 2012.

[29] N. Huber, M. von Quastl, M. Hauck, and S. Kounev. Evaluating and modeling virtualization performance overhead for cloud environments. In *International Conference on Cloud Computing and Service Science (CLOSER 2011), Noordwijkerhout, The Netherlands*, 2011.

[30] Farnam Jahanian, Ragunathan Rajkumar, and Sitaram C.V. Raju. Runtime monitoring of timing constraints in distributed real-time systems. *Real-Time Systems*, 7(3):247–273, 1994.

[31] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 Object-Oriented Programming*, volume

1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997.

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

[33] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[34] 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*, page 75, 2004.

[35] Patrick Meredith and Grigore Roşu. Runtime verification with the RV system. In *Proceedings of the First international conference on Runtime verification*, RV'10, pages 136–152, Berlin, Heidelberg, 2010. Springer-Verlag.

[36] S. Navabpour, B. Bonakdarpour, and S. Fischmeister. Path-aware time-triggered runtime verification. In *Runtime Verification (RV)*, pages 199–213, 2012.

[37] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. Copilot: A hard real-time runtime monitor. In *Proceedings of the First international conference on Runtime verification*, RV'10, pages 345–359, Berlin, Heidelberg, 2010. Springer-Verlag.

[38] A. Pnueli and A. Zaks. PSL Model Checking and Run-Time Verification via Testers. In *Proceedings of the 14th international conference on Formal Methods*, FM'06, pages 573–586, Berlin, Heidelberg, 2006. Springer-Verlag.

[39] D. E. Rivera, M. Morari, and S. Skogestad. Internal model control: PID controller design. *Industrial & Engineering Chemistry Process Design and Development*, 25(1):252–265, 1986.

[40] Grigore Roşu, Feng Chen, and Thomas Ball. Runtime verification. chapter Synthesizing Monitors for Safety Properties: This Time with Calls and Returns, pages 51–68. Springer-Verlag, Berlin, Heidelberg, 2008.

[41] T. J. Ross. *Fuzzy logic with engineering applications.* Wiley, 2009.

[42] O. Sokolsky, S. Kannan, M. Kim, I. Lee, and M. Viswanathan. Steering of Real-Time Systems Based on Monitoring and Checking. In *Proceedings of the Fifth International*

Workshop on Object-Oriented Real-Time Dependable Systems, WORDS '99, pages 11–, Washington, DC, USA, 1999. IEEE Computer Society.

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

[44] Scott D. Stoller, Ezio Bartocci, Justin Seyster, Radu Grosu, Klaus Havelund, Scott A. Smolka, and Erez Zadok. Runtime verification with state estimation. In *Proceedings of the Second international conference on Runtime verification*, RV'11, pages 193–207, Berlin, Heidelberg, 2012. Springer-Verlag.

[45] Volker Stolz and Eric Bodden. Temporal Assertions using AspectJ. *Electron. Notes Theor. Comput. Sci.*, 144(4):109–124, May 2006.

[46] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007, 2002.

[47] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *Proceedings of the 6th international conference on Generative programming and component engineering*, GPCE '07, pages 95–104, New York, NY, USA, 2007. ACM.

[48] Johnson J. Thomas, Sebastian Fischmeister, and Deepak Kumar. Lowering overhead in sampling-based execution monitoring and tracing. *SIGPLAN Not.*, 46(5):101–110, April 2011.

[49] Wallace Wu, D. Kumar, B. Bonakdarpour, and S. Fischmeister. Reducing monitoring overhead by integrating event- and time-triggered techniques. In *International Conference on Runtime Verification (RV)*, 2013. To appear.

[50] J. G. Ziegler and N. B. Nichols. Optimum settings for automatic controllers. *trans. ASME*, 64(11), 1942.