

Runtime Monitoring of Cyber-physical Systems under Timing and Memory Constraints

RAMY MEDHAT, Department of Electrical and Computer Engineering, University of Waterloo
BORZOO BONAKDARPOUR, Department of Computing and Software, McMaster University
DEEPAK KUMAR, Department of Electrical and Computer Engineering, University of Waterloo
SEBASTIAN FISCHMEISTER, Department of Electrical and Computer Engineering, University of Waterloo

The goal of *runtime monitoring* is to inspect the well-being of a system by employing a *monitor process* that reads the state of the system during execution and evaluates a set of properties expressed in some specification language. The main challenge in runtime monitoring is dealing with the costs imposed in terms of resource utilization. In the context of cyber-physical systems, it is crucial for a software monitoring solution to be *time-predictable* to improve scheduling, as well as support composition of monitoring solutions with an overall predictable behavior. Moreover, a small memory footprint is often required in components of cyber-physical systems, especially in deeply embedded systems. In this article, we propose a novel control-theoretic software monitoring solution for coordinating time predictability and memory utilization in runtime monitoring of systems that interact with the physical world. The controllers attempt to reduce monitoring jitter and maximize memory utilization while simultaneously ensuring the soundness of evaluation of properties. For systems where multiple properties are required to be monitored simultaneously, we construct a buffer sharing mechanism in which controllers dynamically share the memory space to negate the effect of bursts of environment actions, thus, reducing jitter due to transient high loads.

To validate our design choices, we present three case studies: (1) a Bluetooth mobile payment system which shows a sporadic rate of events during peak hours; (2) a laser beam stabilizer for target tracking, and (3) a monitoring system for air/fuel ratio in a car engine exhaust and the CAM inlet position in the engine's cylinders. The experimental results of the case studies demonstrate up to 40% improvement in time predictability of the monitoring solution when compared to a basic event-triggered approach. Moreover, memory utilization reaches an average of 90% when using our dynamic buffer resizing mechanism.

CCS Concepts: • **Computer systems organization** → **Embedded and cyber-physical systems; Embedded software;** • **Software and its engineering** → *Software verification and validation*;

Additional Key Words and Phrases: Runtime monitoring, resource efficiency, cyber-physical systems

1. INTRODUCTION

Given the complexity of today's computing systems, exhaustive verification techniques such as model checking and theorem proving may not realistically scale to analyze the system's correctness. On the other side of the spectrum, testing is an established best-effort method to examine the correctness, which scrutinizes only a subset of behaviors of the system. However, testing may not reveal corner cases that complex software

This research was supported in part by Canada NSERC Strategic Project Grants 463324-2014 and 430575-2012, and, NSERC Discovery Grant 418396-2012.

Author's addresses: R. Medhat, Department of Electrical and Computer Engineering, University of Waterloo, Canada, Email: ramy.medhat@uwaterloo.ca; B. Bonakdarpour (**corresponding author**), Department of Computing and Software, McMaster University, Canada, Email: borzoo@mcmaster.ca; D. Kumar, Department of Electrical and Computer Engineering, University of Waterloo, Canada, Email: d6kumar@uwaterloo.ca; S. Fischmeister, Department of Electrical and Computer Engineering, University of Waterloo, Canada, Email: sfischme@uwaterloo.ca.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© YYYY ACM. 1539-9087/YYYY/01-ARTA \$15.00

DOI: <http://dx.doi.org/10.1145/0000000.0000000>

may reach at run time. *Runtime verification* (RV) [Colin and Mariani 2005; Pnueli and Zaks 2006; Giannakopoulou and Havelund 2001] is a 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 execution overhead. In the context of soft real-time systems, decreasing overhead is not the only challenge in adopting RV. We argue that another 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. This problem is even more amplified in the context of cyber-physical systems because physical processes in different environments often exhibit highly unpredictable behavior. This is further illustrated in [Kopetz and Bauer 2003; Kopetz 1991] where the author demonstrates the advantages of a time-triggered approach versus event triggered in terms of schedulability as well as compositionality.

With this motivation, in this paper, we focus on designing an RV technique that targets time-sensitive cyber-physical 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 invoked periodically, where the period of invocation is called the *polling period*. Events that occur between two monitor invocations are buffered and the monitor processes these events in batches when invoked. In order to enforce a property violation detection latency, the polling period cannot be greater than some value given as a system parameter. The monitor is required to maintain a polling period with *minimum jitter*. We refer to such a monitor as a *time-predictable* monitor.
- (2) The monitor must be *sound*; i.e., false-positives and false-negatives are not acceptable. This implies that no event can be dropped.
- (3) We assume a bounded-size buffer for storing events between monitor invocations. This buffer is required to be filled with *maximum utilization*, meaning that after the expiry of a polling period, the available memory space is completely filled with buffered events. This is a requirement since low utilization of the available space implies more frequent monitor invocations, resulting in an increased overhead.

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 employ the rich literature of control theory to enforce the three aforementioned requirements. Using a feedback loop, a controller can learn and predict the behavior of the environment and adapt to the sporadicity of the system. Adapting to such sporadicity implies maintaining minimum jitter due to monitoring, which can be achieved using two approaches:

- **Directly** by controlling the monitor polling period. If the polling periods have a low variance during the execution of the program, jitter is reduced. We refer to this type of controller as a *Polling Period Controller* (PPC). Direct control is more suitable for a single monitor setting, where the monitor operates within a fixed memory bound.

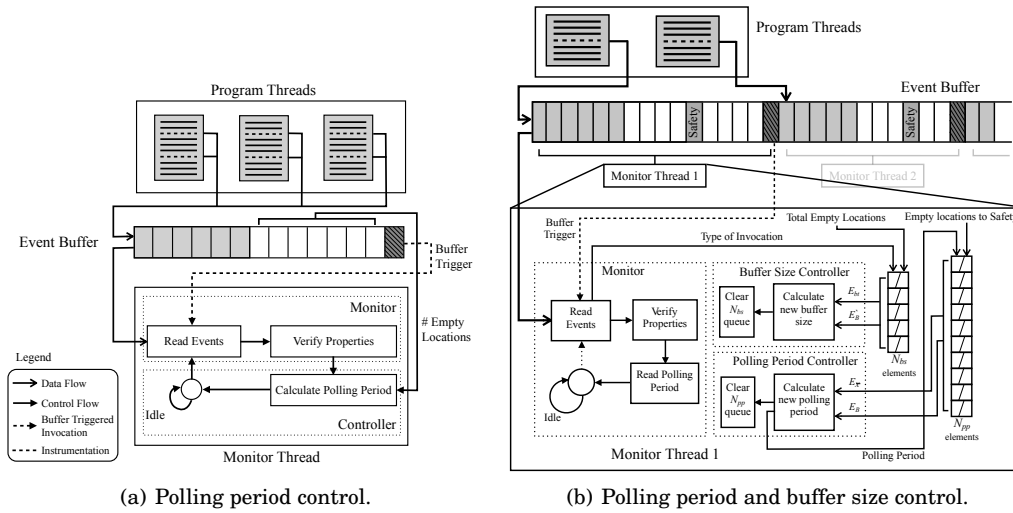


Fig. 1. Outline of direct and indirect control.

— **Indirectly** through allocating temporary memory to absorb transient high loads on the system. The intuition behind indirect control is based on the observation that multiple monitors often exist in systems to verify a set of properties. Thus, instead of providing a fixed size buffer for every monitor, we design a *Buffer Size Controller* (BSC) that allows the individual monitor buffers to fluctuate dynamically based on demand. Hence, a monitor that is experiencing a steady influx of events can maintain its already allocated buffer size, allowing another monitor to extend its buffer size when a surge of events occurs.

Direct Control. Direct control (see Figure 1(a)) 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. To this end, we design five controllers: a PID controller for systems with expected linear rate of occurrence of events, and 4 fuzzy controllers for non-linear systems with different design objectives.

Fuzzy controllers provide a set of advantages that make them a suitable choice for controlling memory utilization and time predictability: They

- (1) support non-linear systems where incoming events can potentially exhibit high variability in short periods of time;
- (2) are computationally efficient, thus, inducing minimal overhead on the system, which is an especially desirable characteristic in runtime monitoring; and
- (3) show significant improvement over an uncontrolled system, even without customizations and optimizations.

To verify our fuzzy controllers, we conduct two thorough case studies. The first case study is on a Bluetooth mobile payment system, which shows non-linear behavior. We experiment with different memory and timing constraints and select a subset that demonstrates a trend in the controller's behavior. The total memory requirement of our approach is linear in the size of the buffer, hence making it suitable for deeply embedded systems, where the buffer size can be limited to a few KBs. Our results show that our controllers on average improve time predictability by a factor of 2.4, while maintaining an average memory utilization of 70%. The second case study is on

a laser beam stabilizer (LBS), which has applications such as aircraft tracking or laser eye surgery. Our results show that our controller introduces negligible disturbance to the control software of LBS. This was verified statistically, where the hypothesis that *the mean response time is different between controlled and uncontrolled systems* failed to be proved using a 95% confidence interval. This result suggests the applicability of our approach to real-time embedded systems.

Indirect Control. To indirectly control predictability through buffer size, we design a second fuzzy controller: BSC (see Figure 1(b)). We emphasize that the design of our all fuzzy controllers in this paper (i.e., fuzzy sets and membership functions) is quite simple and straightforward and the designer does not need to incorporate sophisticated knowledge about the system in the controller design. To validate the use of BSC, we conduct a set of experiments on monitoring of air/fuel ratio in a Toyota 2JZ engine exhaust, as well as the CAM inlet position. Naturally, embedding monitors in the engine control unit (ECU) should not impose non-uniform load and should be time-predictable. Our experiments show (1) that BCS can prevent up to 27.5% of the overshoots (i.e., buffer overloads) and improve time predictability by 40%, and (2) a positive correlation coefficient of 0.71 between the number of buffer overshoots and the coefficient of variation in the polling period.

These results strongly support the basis on which the controllers are designed:

Whether direct or indirect, feedback based control of the monitor's polling period significantly improves its resource utilization.

The scalability of the solution is inherent in its support for composition. Each monitor is responsible for maintaining its polling period and, hence, multiple monitors can coexist. By demonstrating two coexisting monitors in our engine experiments sharing buffer space, we show that one can compose two sets of monitors with a single buffer size controller.

Contributions. We make the following contributions:

- An analysis of the requirement of time predictability and a metric to quantify the effectiveness of a solution that could be used in future comparisons.
- A novel control-theoretic approach to construct time-predictable run time monitors suited for the dynamics of cyber-physical systems.
- A set of case studies that rigorously study the performance of the proposed approaches.
- A novel buffer sharing mechanism to support multiple concurrent monitors and demonstrate the scalability of the proposed approach.

Organization. The rest of the paper is organized as follows. In Section 2, we formally state the monitoring objectives of a single monitor. Section 3 recaps the basic concepts on PID and fuzzy controllers. Our polling period controller design choices are explained in Section 4. We present experiments on Bluetooth payment and the laser beam stabilizer in Section 5. We extend the problem formulation to cover multiple monitors in Section 6. We introduce the design of the buffer size controller and how it interacts with the polling period controller in Section 7. Experimental design and results of the ECU case study are presented in Section 8. Section 9 summarizes the results from both sets of experiments and provides insight into the applicability and limitations of the proposed approach. Related work is discussed in Section 10. Finally, we make concluding remarks and discuss future work in Section 11.

2. SINGLE MONITOR PROBLEM DESCRIPTION

A logical property (e.g., a simple Boolean expression) is often expressed in terms of a set of program variables whose values may change over time. We call such a change of value an *event*. Thus, monitoring a property involves invoking a process (called the *monitor*) for each event that may change the valuation of the property. This paper 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.
- **Time predictability.** Since invocation of the monitor interrupts the normal execution of the program, we require that these interruptions to be predictable with respect to time. This requirement assists in achieving more accurate system-wide scheduling.
- **Resource utilization.** The monitor may use bounded-size 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 that is under inspection and Φ be the system specification expressed in some language (e.g., in LTL), where R is expected to satisfy Φ . Since, R has limited memory, we assume that the number of events it can buffer for monitoring has an upper bound B .

Let $E = e_1 e_2 \cdots e_n$, where $n \in \mathbb{N}$, 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 in E . Also, let $V = v_1 v_2 \cdots v_k$ be the output finite sequence of monitor invocations and $T_v = t_{v_0} t_{v_1} t_{v_2} \cdots t_{v_k}$ be the finite sequence of timestamps of monitor invocations, where $k \in \mathbb{N}$ and t_{v_0} is the start time of the monitor. We note that k is a variable to be controlled, meaning that depending upon the monitoring policy, k may change.

Let function $between(\tau_1, \tau_2)$ be a function that returns all the events that occur between times τ_1 and τ_2 :

$$between(\tau_1, \tau_2) = \{e_i \mid \tau_1 < t_{e_i} < \tau_2\} \quad (1)$$

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

$$\forall i \in \{1 \cdots k\} : |between(t_{v_{i-1}}, t_{v_i})| \leq B \quad (2)$$

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

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

$$\max \left\{ \frac{1}{k} \sum_{i=1}^k \frac{|between(t_{v_{i-1}}, t_{v_i})|}{B} \right\} \quad (3)$$

Thus, the objective is to maximize the average memory utilization across the complete run of the monitor essentially by maximizing the filling ratio of the buffer ($|between(t_{v_{i-1}}, t_{v_i})|/B$) in the smallest number of monitor invocations (i.e., k).

Let $X = \{X_i \mid 1 \leq i \leq k\}$ be the set, where $X_i = t_{v_i} - t_{v_{i-1}}$. That is, each X_i is the amount of time elapsed between monitor invocations v_i and v_{i-1} . We characterize *time predictability* by the following objective:

$$\min \{V(X) \mid \text{for all possible sets of } X\} \quad (4)$$

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

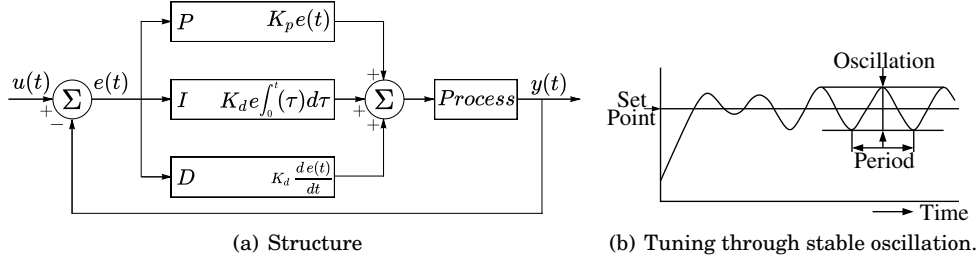


Fig. 2. PID controller.

Observe that the best case minimum variance is zero, which means that for all i , $t_{v_i} - t_{v_{i-1}}$ remains constant. However, if a monitor adopts a constant monitoring frequency, it may lose soundness in a reactive system, as the rate of occurrence of events depends upon external stimuli, such as actions of physical processes. 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 illustrates 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 2, 3, and 4) simultaneously through identifying T_v (i.e., time of monitor invocations and, hence, k) in a best-effort fashion.

3. BASIC CONTROL THEORY

Since our approach is based on controller design, we recap the concepts of PID controllers in Subsection 3.1 and Fuzzy controllers in Subsection 3.2.

3.1. PID Controllers

A PID feedback controller [Rivera et al. 1986] 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 the *sampling period*. The three components are then applied collectively to $e(t)$ as follows:

$$u(t) = K_P e(t) + K_I \int e(t)dt + K_D \frac{d}{dt} e(t) \quad (5)$$

where K_P is the proportional gain, K_I is the integral gain, and K_D is the differential gain. Figure 2(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, so as to deduce ideal gains that ensure controllable behavior. Another method is using experience to configure these controllers; often engineers on site can postulate an initial configuration for PID controllers using well-known methods. In this paper, we use the popular Ziegler-Nichols method [Ziegler and Nichols 1942] 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 2(b)), where

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

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 Zeigler-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 overshoot¹ 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 [Driankov et al. 1993]. 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. These fuzzy conditions/actions resemble 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.

3.2.1. Fuzzy Logic. The first function of a fuzzy controller is to transform a discrete measured value called a *crisp* value (e.g., 30° or $1.9m$) into a *fuzzy* value (e.g., High or Tall). 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*, which maps each value $u \in \mathcal{U}$ to a value within the interval $[0, 1]$. 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 variable that can be expressed in fuzzy values instead of numerical crisp values), 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:

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

where x is 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:

$$\mu_I(u) = \max_k \{ \mu_{A^{(k)}}(x) \cdot \mu_{B^{(k)}}(u) \} \quad (6)$$

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

¹An *overshoot* is when a signal or function exceeds its target.

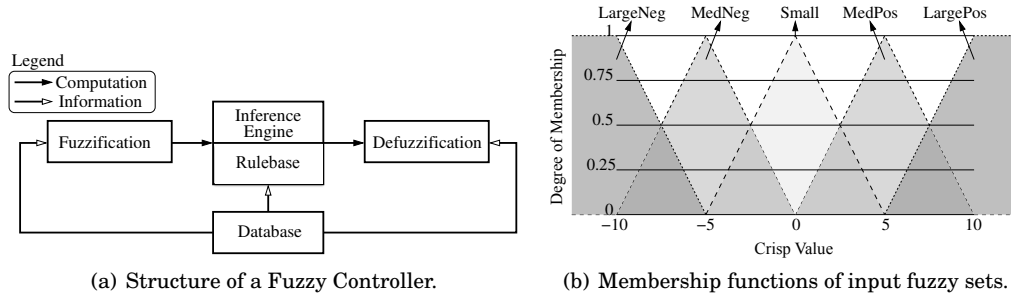


Fig. 3. Fuzzy controller.

3.2.2. Structure of a Fuzzy Controller. Figure 3(a) shows the structure of a typical fuzzy controller [Driankov et al. 1993]. 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 paper, we use scaled inference for fuzzification.
- **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 [Ross 2009]. An example of these functions is shown in Figure 3(b). LargeNeg is a *Z-type* function, LargePos is an *S-type* function, and MedNeg, Small, and MedPos are *lambda-type* functions.
- **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 6. The output crisp value u^* is calculated as follows:

$$u^* = \frac{\int_{-\infty}^{+\infty} u \cdot \mu_I(u) du}{\int_{-\infty}^{+\infty} \mu_I(u) du} \quad (7)$$

4. POLLING PERIOD CONTROLLER (PPC) DESIGN

This section presents in detail the design of our polling period controllers based on the objectives in Equations 2, 3, and 4. As shown in Figure 1(a), the program under inspection can be multi-threaded. We instrument the program, so that it enqueues the events in a bounded-size buffer whenever variables of interest 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.

PPC (see Figure 1(a)) executes within the monitor thread. With every invocation of the monitor, it determines when the next invocation should occur to satisfy Equations 3 and 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 4.1–4.5, describe the design of our PID and four fuzzy controllers.

4.1. PID Polling Period Controller (PPC:PID)

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. Our design is as follows:

- **Input.** In order to achieve maximum memory utilization (Equation 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:

$$e(t_{v_i}) = B \times S - |between(t_{v_{i-1}}, t_{v_i})|$$

where B is the buffer size, S is the safety threshold percentage, t_{v_i} is the timestamp of the current invocation of the monitor, $t_{v_{i-1}}$ is the timestamp of the last invocation of the monitor, and $between(t_{v_{i-1}}, t_{v_i})$ is the set of events received between the two timestamps (defined in Equation 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. The proportional, integral, and derivative gains are $0.6K_u$, $2K_p/T_u$, and $K_p T_u/8$, respectively, where T_u is the period of constant oscillation and K_u is the proportional gain at which oscillation occurs (see Figure 2(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 [Galan 2003].

4.2. Fuzzy 1 Polling Period Controller (PPC:F1)

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

- **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:

$$E_B = B \times S - |between(t_{v_{i-1}}, t_{v_i})|$$

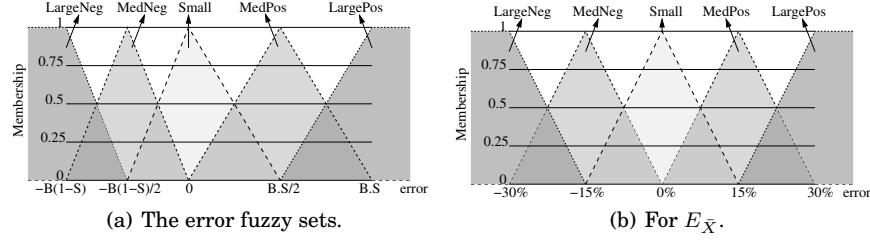


Fig. 4. Membership functions.

There are 5 fuzzy sets for the error variable based on lambda-type functions as shown in Figure 4(a). 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(b), with centers at -1 , -0.5 , 0 , 0.5 , and 1 , respectively. The output is multiplied by a factor depending on the nature of the system.
- **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.

4.3. Fuzzy 2 Polling Period Controller (PPC:F2)

PPC:F2 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.

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

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

- **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:

$$E_{\bar{X}} = \frac{X - \bar{X}}{\bar{X}} \quad (8)$$

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 4(b). 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 PPC:F1 (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 I 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. This mapping is a configuration parameter and could be changed according to the system requirements.

4.4. Fuzzy 3 Polling Period Controller (PPC:F3)

Instead of minimizing the variance, PPC:F3 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 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

$$c_v = \frac{\sigma_X}{\bar{X}}$$

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.

PPC:F3 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 leads 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:

$$\gamma = \left(\frac{N + 1 + k^2 N}{N(N + 1)^2} \right) \quad (9)$$

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

$$\left(\frac{1}{N} - \gamma\right) X_{N+1}^2 - \left(2\gamma \sum_{i=1}^N X_i\right) X_{N+1} + \left(\frac{1}{N} \sum_{i=1}^N X_i^2 - \gamma \sum_{i=1}^N X_i\right) = 0 \quad (10)$$

If Equation 10 has complex roots, then it is not possible for X_{N+1} to lower the coefficient of variation down to k . In this case, PPC:F3 falls back to PPC:F2, 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 10 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 shows how these functions are defined, where r_1 and r_2 are the roots of Equation 10. 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.

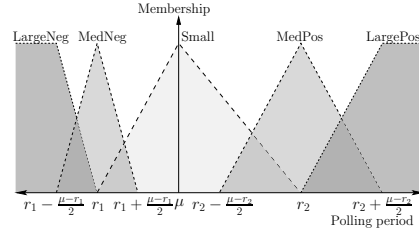


Fig. 5. Membership functions of E_{c_v} .

The mapping in the *if-then* rules in this controller is similar to the mapping in PPC:F2 as shown in Table I, which maintains a balanced trade-off between memory utilization and time predictability.

4.5. Fuzzy 4 Polling Period Controller (PPC:F4)

PPC:F4 is essentially the same as PPC:F3, with the exception of the mapping for the *if-then* rules. In this controller, the mapping gives preference to controlling memory utilization when E_B is a LargeNeg value, even if that contradicts with the time predictability requirement. Table II shows the modified mapping. This mapping enables

PPC:F4 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.

Table II. Asymmetric mapping of input variables in *if-then* rules.

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

5. SINGLE MONITOR EXPERIMENTS AND RESULTS

In order to analyze the performance of our polling period controllers, we have conducted experiments on two cyber-physical systems: (1) a Bluetooth mobile payment, and (2) a laser beam stabilizer for aircraft tracking and eye surgery. Each case study involves using different controllers with different configurations. The controllers presented in this article are implemented in C, and use the real-time API to handle scheduling the monitor thread. This includes using real-time clocks and real-time thread priority.

Our experiments are designed based on three factors:

- (1) **Controller type.** We incorporate seven controllers in our experiments: PPC:PID, PPC:F1, PPC:F2, PPC:F3 with target coefficient of variation $c_v = 0.4$, PPC:F3 with $c_v = 0.2$, PPC:F4 with $c_v = 0.4$, and PPC:F4 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 four 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.** 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 predictability.
- (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.
- (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.

5.1. Case Study1: 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 communicating with devices. 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 occurs.

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(a), 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 [Galati and Greenhalgh 2010]. 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.

5.1.1. Analysis of Time Predictability. Figure 6 shows the average polling period coefficient of variation C_v across all 9 replicates for buffer sizes 20, 40, and 60. As can be seen, PPC:F2 exhibits the lowest C_v , since it is designed to control the polling period within $\pm 15\%$ of the mean (see Section 4.3). PPC:F3 targeting $C_v = 0.2$ (denoted PPC:F3-0.2 in the figure) and PPC:F4-0.2 show low C_v due to having an aggressive $C_v = 0.2$ goal. In Figure 6(a), PPC:F3-0.2 and PPC:F4-0.2 fail 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 6(b) and 6(c). However, PPC:F3-0.4 and PPC:F4-0.4 consistently

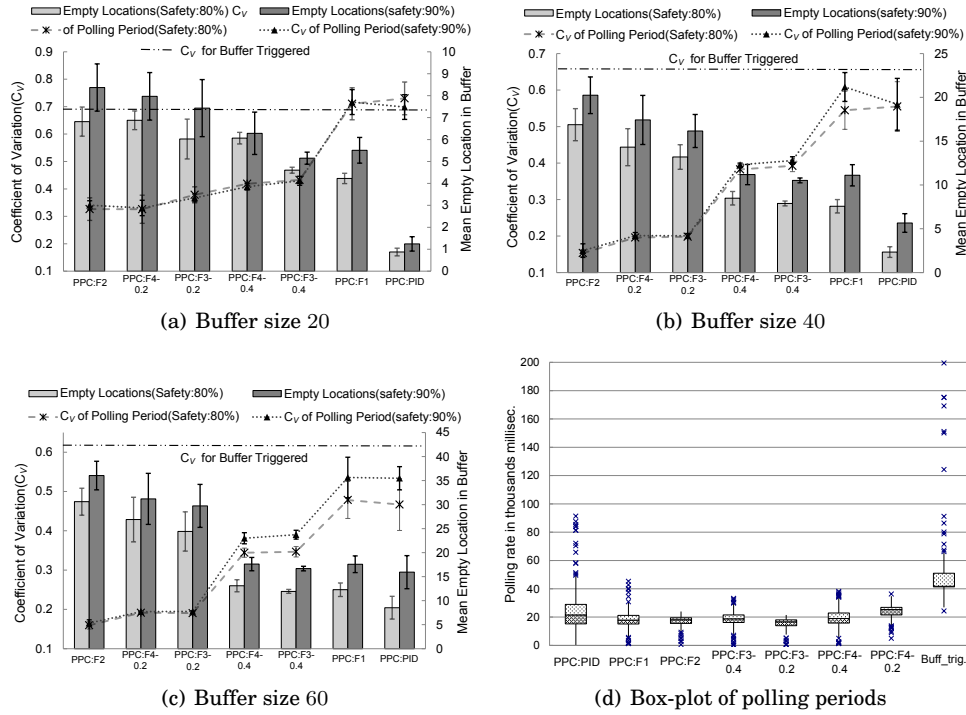


Fig. 6. Polling period predictability results on all 7 controllers and both safety threshold values for BTP.

meet their goal ($C_v = 0.4$) across all configurations. Since PPC:F1 and PPC: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 PPC:F2, 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, where the feedback loop aids the controlling in adapting to the changes in the rate of incoming events, thus smoothing the transitions in the polling period and maintaining low variation.

Figure 6(d) 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 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 PPC:PID, explained by the inability of the PID to adapt to a non-linear system. Again, it can be seen that using PPC:F1, which has the same goal as the PPC:PID, can drastically improve the stability of the controller. The lowest variability is - as expected - due to PPC:F2, PPC:F3-0.2, and PPC:F4-0.2.

5.1.2. Memory Utilization. Figure 6 shows 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, PPC:PID consistently has the lowest error mean, and thus provides the highest memory utilization. The error mean for PPC:F1 is also low, and comparable to that of the PPC:PID when the buffer size increases (see Figures 6(b) and 6(c)). PPC:F2 exhibits a consistently high error

mean. This is due to the fact that PPC:F2 is designed to be aggressive in maintaining a low polling period coefficient of variation C_v , which comes at the cost of error. This also applies to PPC:F3 aggressively targeting $C_v = 0.2$ (denoted PPC:F3-0.2) and PPC:F4 targeting $C_v = 0.2$. However, PPC:F3-0.4 and PPC:F4-0.4 perform comparably to PPC:PID and PPC:F1, 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. This figure shows the number of buffer triggers occurred for every controller. It appears that the reason PPC: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 cyber-physical systems, and as a result is overshooting considerably more than any other controller. This also shows that PPC:F1, 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.

5.1.3. 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 separate 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, PPC:PID and PPC:F1 introduce a 19% increase in execution time. However, other fuzzy controllers average around 10%.

5.1.4. Other 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 illustrates 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 lower than any controller-based approach. The actual overhead of monitoring has been measured for both controlled monitoring and uncontrolled monitoring (buffer triggered). We then performed a hypothesis test on whether the means of the distributions of both sets of overheads is different. The hypothesis failed to prove (with 95% confidence) that there is a significant impact on overhead when using our controlled 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 PPC:F3-0.4 and PPC:F4-0.4 exhibit the best balance between the two goals consistently across configurations.

Resilience to overshoots. Figure 7 shows that all PPC:F4 controllers present an advantage over PPC:F3 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.

5.2. Case Study 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 PPC:F3 and PPC:F4. This demonstrates how a more relaxed constraint affects the response of the controller.

5.2.1. Time predictability. Figure 8 shows the results of the experiments on buffer size of 40. The trend of C_v for polling period versus error mean in Figure 8(a) is similar to that of the BTP experiment. PPC:F3-0.2 scores a much higher C_v than its goal (0.6 vs. a goal of 0.2). However, FPPC:F4-0.2 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 PPC:F4 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 PPC:F4-0.2 and PPC:F2. This is due to the periodic nature of the events, which enables a purely buffer-triggered solution to naturally produce a lower C_v . PPC:F2 and PPC:F4-0.2, however, are more aggressive in maintaining a low C_v and, thus, they outperform pure buffer triggered.

5.2.2. Memory Utilization. Figure 8(a) shows that low C_v comes at the cost of the error mean. This is contrasted with the number of buffer triggers in Figure 8(b), which shows that PID has the highest number.

5.2.3. Additional Observations

— **Overhead.** As with the BTP case study, multiple trials resulted in a set of noisy measurements of monitoring overhead, which statistically appear to be from the

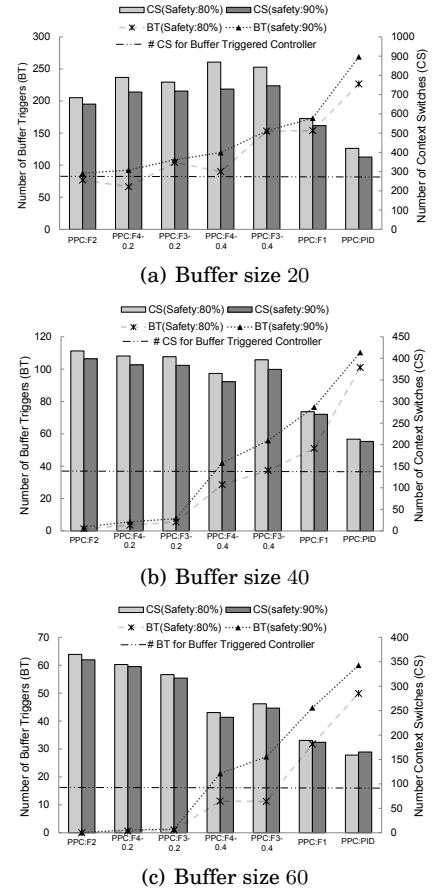


Fig. 7. Number of buffer triggers vs. number of context switches for BTP

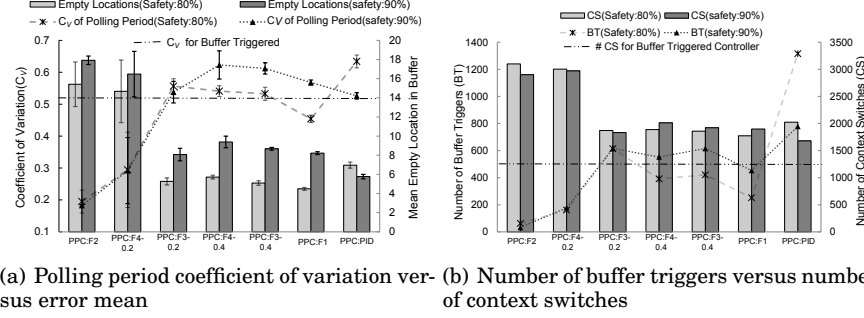


Fig. 8. Results of all 7 controllers at buffer size 40 and both safety threshold values for LBS

same distribution. Hence, no negative impact of using our monitor versus a buffer triggered monitor can be observed.

- **Tuning cost.** In Figure 8(b), 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 PPC:F1 which attempts the same objective, PPC:F1 appears to be more consistent.
- **Controller instability.** In Figure 8(a), 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 8(b)). 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.

6. MULTIPLE MONITOR PROBLEM DESCRIPTION

This section presents the formulation of resource-efficient monitoring problem, where multiple monitors inspect a system. In this setting, several monitors coexist in the system, sharing a bounded buffer space. There is no strict constraint on the size of the buffer available for every monitor, allowing more flexibility in the problem definition. Our hypothesis is that giving more space to a monitor (even though bounded) when burst of events occur, assists in maintaining time predictability of monitor invocations.

First, we define $M = \{m_1, m_2, \dots, m_l\}$ as a set of l monitors in the system. Thus, we redefine the sequence of invocations of monitor m_j as follows:

$$V_j = v_{j,1} v_{j,2} \dots v_{j,k_j} \quad (11)$$

where k_j is the index of the final invocation of that monitor m_j . Next, we redefine the sequence of invocation timestamps of monitor m_j as follows:

$$T_{v_j} = t_{v_{j,0}} t_{v_{j,1}} \dots t_{v_{j,k_j}}$$

with $t_{v_{j,0}}$ being the time monitor m_j starts. We now formulate the dynamicity of memory available to multiple monitors by augmenting the constraints presented in Section 2. The total memory available for monitoring is \mathbb{B} , which is divided into a static buffer and a shared buffer (not necessarily equal). These buffers are defined as follows:

- **Static Buffer (\mathcal{B}).** The static buffer is a preallocated space in memory. Each monitor m_j is guaranteed an exclusive portion of the static buffer of size \mathcal{B}_{m_j} . Thus,

$$\mathcal{B} = \sum_{j=1}^l \mathcal{B}_{m_j}$$

— **Shared Buffer** (β). The shared buffer is a memory space from which monitors can allocate chunks when they are experiencing transient high loads. A monitor can only allocate/release a chunk of shared memory when it is invoked. Since monitor m_j is invoked k_j times (Equation 11), the following sequence denotes the sizes of chunks reserved by monitor m_j from the shared buffer:

$$\beta_{m_j} = \beta_{j,0}\beta_{j,1}\cdots\beta_{j,k_j}$$

where $\beta_{j,i}$ is the size of the chunk of memory reserved by monitor m_j at its i^{th} invocation.

Thus, the total memory available for monitoring is

$$\mathbb{B} = \mathcal{B} + \beta$$

We denote the total amount of memory available for monitor m_j as its *extended buffer*. Let $B_{m_j} = b_{j,0}b_{j,1}b_{j,2}\cdots b_{j,k_j}$ be the sequence of extended buffer sizes at each invocation of monitor m_j , where $b_{j,i}$ is the sum of the static buffer space allocated for m_j and the chunk of memory allocated from the shared buffer at the i^{th} invocation:

$$b_{j,i} = \mathcal{B}_{m_j} + \beta_{j,i}$$

Since the extended buffer size is decided by the monitor, the total reserved space by all monitors at any given point in time should never exceed the size of total memory available for monitoring (denoted \mathbb{B}). To formulate this condition, we define a function π :

$$\pi(\mathcal{T}, v_{j,i}) = \begin{cases} b_{j,i} & \text{if } t_{v_{j,i}} \leq \mathcal{T} < t_{v_{j,i+1}} \\ 0 & \text{otherwise} \end{cases} \quad (12)$$

where \mathcal{T} is a point in time. The following condition ensures that the maximum total buffer size is never surpassed:

$$\forall \mathcal{T} : \sum_{j=1}^l \sum_{i=1}^{k_j} \pi(\mathcal{T}, v_{j,i}) \leq \mathbb{B}$$

The extended buffer size should also not fall below the lower bound \mathcal{B}_{m_j} which is the static buffer size for monitor m_j . Thus, the following condition must hold:

$$\forall j \forall i : \mathcal{B}_{m_j} \leq b_{j,i} \quad (13)$$

where $j \in \{1 \cdots l\}$ and $i \in \{1 \cdots k_j\}$. Based on Equation 1, we redefine *soundness* as follows:

$$\forall j \forall i \mid \text{between}(t_{v_{j,i-1}}, t_{v_{j,i}}) \leq b_{j,i} \quad (14)$$

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

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

$$\max_{T_{v_j}} \left\{ j \in [1, l] \mid \frac{1}{k_j} \sum_{i=1}^{k_j} \frac{|\text{between}(t_{v_{j,i-1}}, t_{v_{j,i}})|}{b_{j,i}} \right\} \quad (15)$$

Thus, the objective is to maximize the average memory utilization across all monitors by maximizing the filling ratio of the extended buffer ($|\text{between}(t_{v_{j,i-1}}, t_{v_{j,i}})|/b_{j,i}$). Since the total number of events in any invocation pattern is the same, the smaller the k_j , the higher the average.

This memory utilization is relative to the extended buffer space at the time of monitor invocation. However, since we allow dynamic reservation and release of memory, the monitoring solution should also attempt to minimize amount of memory allocated from the shared buffer β :

$$\min \left\{ \sum_{j=1}^l \sum_{i=1}^{k_j} \beta_{j,i} \right\} \quad (16)$$

That is, the total size of extra reservations across all invocations of all monitors should be minimized.

The constraint on time predictability is modified such that objective is to minimize the *mean* variation of polling periods for individual monitors. That is, upon calculating the variance of every monitor separately, the average variance across all monitors should be minimized. To formulate this, let $X_j = \{X_{j,i} \mid 1 \leq i \leq k_j\}$ be the set, where

$$X_{j,i} = t_{v_{j,i}} - t_{v_{j,i-1}}$$

Thus, time predictability imposes the following constraint:

$$\min \left\{ \frac{1}{l} \sum_{j=1}^l V(X_j) \right\} \quad (17)$$

Consequently, our goal is to enhance the design in Section 4 to enforce the augmented objectives (i.e., Equations 14, 15, 16, and 17) simultaneously.

7. BUFFER SIZE CONTROLLER (BSC) DESIGN

The intuition behind solving the time predictability of multiple coinciding monitors is that buffer triggers cause jitter in monitor invocation. A shared memory space available for reservation during transient high loads should help negate the unpredictability caused by this jitter. Thus, we present a controller that dynamically reserves more memory to decrease the number of buffer triggers. Based on the number of buffer triggers and the average number of empty buffer locations in the last N_{bs} invocations, the controller makes the decision on whether to reserve or release memory. The controller can only reserve from the shared buffer β .

BSC operates at a higher priority than PPC. The reason for this is to prevent PPC from reacting prematurely, resulting in sudden changes in the polling period. By prioritizing the buffer size controller, the buffer size will be adjusted first, thus alleviating the stress of changing the polling period immediately. To achieve this, the buffer size controller adjusts the safety threshold percentage along with every change in the buffer size to maintain the same threshold value. For instance, assume the buffer size is 20, and the safety threshold is 80% (as set in Subsection 4). This means that PPC will attempt to stabilize memory utilization around $20 \times 0.8 = 16$ elements at every invocation. If the BSC decides to increase the buffer size to 24, it will change the safety threshold to 66%. This will essentially decrease the number of buffer triggers using the extra memory without affecting the polling period controller, since the target is still 16. The only difference is that PPC is under less pressure to change the polling period.

The design of the controller is as follows:

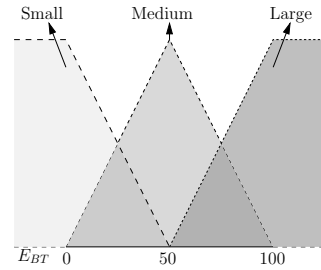


Fig. 9. Membership functions of E_{BT} .

- **Input.** The controller has two inputs. The first input is E_{bt} , which is the percentage of buffer triggered invocations in the last N_{bs} invocations. The second input is E'_B , which is the average percentage of utilization of the buffer in the last N_{bs} invocations. The formula for E'_B is as follows:

$$E'_{Bj}(c) = \frac{1}{N_{bs}} \left(\sum_{i=(c-1) \cdot N_{bs}}^{c \cdot N_{bs}} \frac{|between(t_{j,i-1}, t_{j,i})|}{b_{j,i}} \right) \quad (18)$$

where j is the index of the monitor, and c is the index of the BSC invocation. The reason for omitting safety in the calculations is that BSC dynamically changes the safety threshold percentage as explained above. E_{bt} and E'_B are fuzzified into three sets: Small, Medium, and Large. There is no negative or positive since they are percentages. The membership functions for these sets are centered around 0%, 50%, and 100% (see Figure 9). These values are configuration parameters and can be changed according to the user requirement.

- **Output.** The output of the controller is the offset value from the current buffer size, which we denote as Δ_{bs} . The membership functions for the output variable are standard lambda-type functions similar to those in Figure 3(b), with centers at -1 , -0.5 , 0 , 0.5 , and 1 , respectively. The output is multiplied by a factor depending on the nature of the system.
- **If-then rules.** Table III shows the consequent fuzzy set of each rule based on the combination of the two antecedent fuzzy sets, where the columns are E'_B fuzzy sets, the rows are E_{bt} fuzzy sets, and S, M, and L are abbreviations of Small, Medium, and Large, respectively. The purpose of incorporating E'_B in the design is visible in the mapping. As can be seen in the table, the controller attempts to release memory when there is a large empty space in the buffer. This helps the system reclaim some reserved and underutilized memory, which in effect attempts to satisfy Equation 16. The mapping here favors minimizing buffer triggers, since we can afford to let memory releases gradually accumulate small values with less urgency than, for instance, a high percentage of buffer triggers.
- **Invocation.** Instead of running BSC in a separate thread at its own frequency, we utilize the existing invocation of PPC (caused by the invocation of the monitor) to run BSC. Thus, there are no separate invocations of BSC and, hence, no jitter due to these invocations. The N_{bs} parameter determines how responsive the controller is. A smaller number indicates that BSC reacts quicker to sudden buffer triggers. In Section 8, we will discuss the impact of changing N_{bs} .
- **Customizability.** A lot of tweaking goes in applying fuzzy controllers. A basic customization that could be applied to BSC is to bias it towards being more conservative towards buffer triggers. In Figure 9, the center point is a neutral 50%. Moving this point to the left causes the controller to be less tolerant of buffer triggers, aggressively trying to maintain stability around the now tighter left region. This customization may enhance the performance of the controller versus a trivial implementation such as the one in Figure 9 for some systems where buffer triggers are not tolerable or when non-linearity is very high. We discuss this notion further in our experiments section.
- **Scalability.** The buffer size controller is inherently scalable due to its support of compositionality. Since monitors operate independently, we can group monitors into sets such that the memory allocated for each set is managed by a buffer size controller. We can then compose these sets together into a larger set, that operates within a larger buffer size, and is managed by one buffer size controller. This hierarchical approach is inherent in the design of BSC.

8. MULTIPLE MONITOR EXPERIMENTS AND RESULTS

In order to analyze the performance of our controllers, we conduct a case study on a cyber-physical system where two properties are required to hold: the air/fuel ratio in the exhaust of a Toyota 2JZ engine, and the CAM Inlet positional error. These two properties require two coinciding monitors, thus, introducing the opportunity to use two BSCs.

Table III. Mapping of input variables in if-then rules for BSC.

		E'_B Fuzzy sets		
		S	M	L
E_{BT} Fuzzy sets	S	S	S	MN
	M	MP	MP	S
	L	LP	LP	MP

8.1. Experimental Background

Environmental concerns require diligent monitoring of air/fuel ratio in a vehicles exhaust. The *Lambda value* is the ratio between the amount of oxygen present in the exhaust versus the amount of oxygen that would be present in the exhaust had there been perfect combustion. A lambda value of 1.00 indicates a perfect combustion. If the value is less than 1.00, then the amount of fuel in the exhaust is higher, which is known as *rich*. If greater than 1.00, it is known as *lean*. A vehicle running rich is less environment friendly. However, a highly lean condition indicates a possible misfire which could result in serious engine damage.

A control system is required to maintain an acceptable safe lambda value, which may differ depending on driving conditions. For instance, accelerating the vehicle with a wide open throttle (WOT) requires a richer mixture in the exhaust, while cruising or driving in economy mode would require a leaner running condition. This dynamic behavior is common in cyber-physical systems due to their dependence on the physical environment. An engine control unit (ECU) software determines how rich or lean the engine should run, such that the desired level of emission is maintained and the required performance is achieved. Failure to do so can be caused by multiple reasons, including malfunctioning sensors, or badly timed piston firings. In these critical cases, a verification system is required to report this failure. However, the verification system should not produce largely varying processing overhead that might negatively impact the performance of the main ECU functionality. Hence, time predictability is crucial in this setting. It is more beneficial to bound detection latency and provide a predictable and schedulable runtime monitor than a monitor that has low overhead but does not handle surges of events.

The ECU also monitors the CAM inlet position, ensuring successful fuel injection and sparking. The ECU includes a separate control system responsible for ignition control, which uses a CAM sensor to provide feedback on the position of every cylinder prior to sparking. If the position of the CAM is not optimum repeatedly, this indicates a potential failure in the engine. Such a system needs to be monitored to ensure that the CAM position does not deviate further than is safe for the engine.

8.2. Experimental Settings

The experimental setup is composed of the Toyota 2JZ engine, a multitude of sensors and actuators controlled by the ECU, and a data logging station that allows the operator to monitor engine health. We attempt to employ our controllers to maintain predictable behavior during the run of the engine, regardless of its speed or the rate of incoming events. The events received are divided into two categories:

- Changes in the air/fuel ratio read by the lambda sensor. In the cases of rapid increase or decrease in the speed of the vehicle, the lambda sensor reading also exhibits rapid changes. This causes non-linearity in the rate of events that need processing for verification purposes.

— Changes in the position of the CAM inlet which are used to indicate the efficiency and stability of fuel injection control.

For each category, there is a dedicated monitor responsible for verifying the property associated with the sensor reading. *Monitor 1* is responsible for verifying air/fuel ratio, and *Monitor 2* is responsible for verifying CAM position. We use the sensor logs produced by running the engine through a standard test to evaluate the performance of our solution. The test performed is a step test, in which the RPM of the engine is increased in a stepwise fashion as shown in Figure 10(a). The lambda sensor readings of the engine are shown in Figure 10(b), and the CAM position sensor readings are shown in Figure 10(c). A combined waveform of both inputs is shown in Figure 10(d), where the light grey represents the number of events that monitor 2 receives at any point in time, the medium grey represents the number of events that monitor 1 receives, and the dark grey shows the points in time when both monitors are receiving events concurrently.

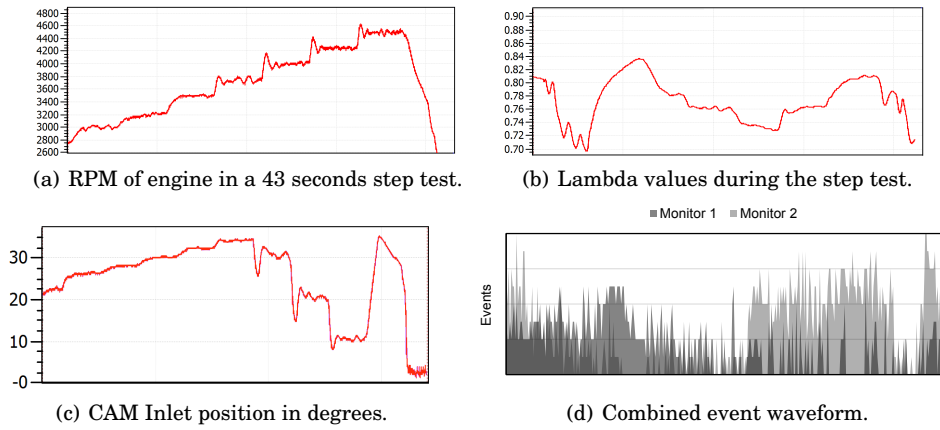


Fig. 10. Lambda value and CAM inlet position readings.

Our experiments are designed based on four factors:

- (1) **Type of PPC.** We incorporate PPC:F1 and PPC:F2 (see Section 4).
- (2) **BSC enable status.** We experiment with enabling and disabling BSC.
- (3) **Per monitor static buffer size (β_{m_j}).** We consider two configurations: (a) A static buffer of size 20 per monitor (total $\beta = 40$) plus a shared buffer of size 20, and (b) a static buffer of size 40 per monitor (total $\beta = 80$) plus a shared buffer of size 20. Thus, the total memory available is $\mathbb{B} = 60$ in Configuration (a), and $\mathbb{B} = 100$ in Configuration (b).
- (4) **Invocation frequency.** Recall from Figure 1(a) that the controllers build a history of previous input signals. We experiment with varying the invocation frequency of both PPC and BSC. We configure the controllers with 2 different combinations of frequencies: (5, 5) and (10, 5), where the first item in the pair denotes the invocation frequency of PPC, and the second item denotes the invocation frequency of BSC.

Hence, there is a total of 32 configurations to test all different combinations of the above four factors. We carry out 30 runs (replicates) with randomization to provide statistical confidence in the results.

We observe five measurement metrics for each monitor. The metrics are as follows:

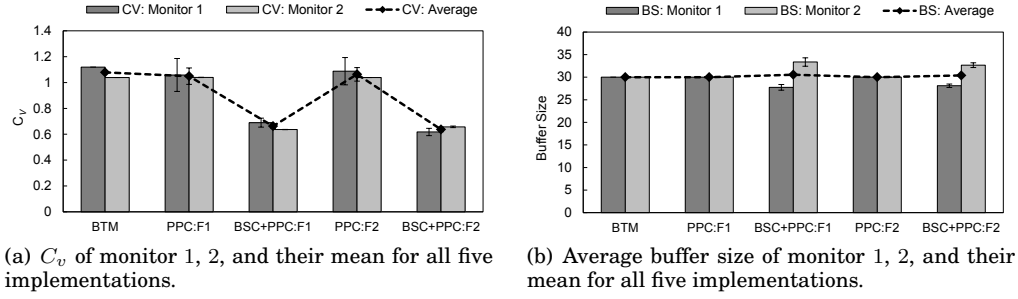


Fig. 11. C_v and average buffer size of both monitors across all five implementations.

- (1) **Error mean ($\overline{E_{B_j}}$)**. This is the mean number of empty buffer locations at every invocation of every monitor. This value is a measure of the memory utilization of the monitor (Equation 3).
- (2) **Polling period coefficient of variation (C_{v_j})**. This value is a measure of time predictability (Equation 4).
- (3) **Context switches (CS_j)**. This value is a measure of the overhead introduced by the monitor.
- (4) **Buffer triggers (BT_j)**. This is the number of buffer-triggered monitor invocations. This value is a measure of the quality of the controller.
- (5) **Average buffer size ($\overline{B_j}$)**. The average size of the extended buffer across the run of an experiment. A value closer to the static buffer size is more desirable since it indicates that the monitor is reserving less extra memory (Equation 16).

We also test the performance of the system in terms of the above metrics when the monitor is invoked *only* due to buffer triggers. This test is performed for both buffer sizes 20 and 40, making the total number of runs in a full replicate equal to 34.

Finally, we implemented the proposed controllers on QNX to ensure predictable timing behavior. To that end, we used QNX hi-res timers to schedule the invocation of monitors. Further, we ported the code to run on three other platforms: Ubuntu 12.04, Windows 7, and Mac OS X 10.9 to garner insight into the effect of these non-real-time platforms on predictability.

8.3. Analysis of Results

Since there are 34 possible combinations of the experimental factors, we first present only a subset of these combinations which are most significant. There are 5 main experiments that emphasize the design tradeoffs:

- (*BTM*) In this implementation, the monitor is always invoked when the static buffer is full.
- (*PPC:F1*) This implementation uses only PPC:F1 (see Section 4.2).
- (*PPC:F2*) This implementation only PPC:F2 (see Section 4.3).
- (*BSC+PPC:F1*) This design implements BSC in addition to PPC:F1.
- (*BSC+PPC:F2*) In this implementation, BSC is used in addition to PPC:F2.

It is important to note that all implementations use exactly the same total amount of memory. The shared space available for implementations that utilize BSC is in fact provided as a static buffer for BTM, PPC:F1, and PPC:F2. Thus, the improvements shown in the results come at no extra memory cost whatsoever.

8.3.1. Time Predictability. Figure 11(a) shows the polling period coefficient of variation (C_v) for the five different implementations mentioned above. The C_v of each monitor is

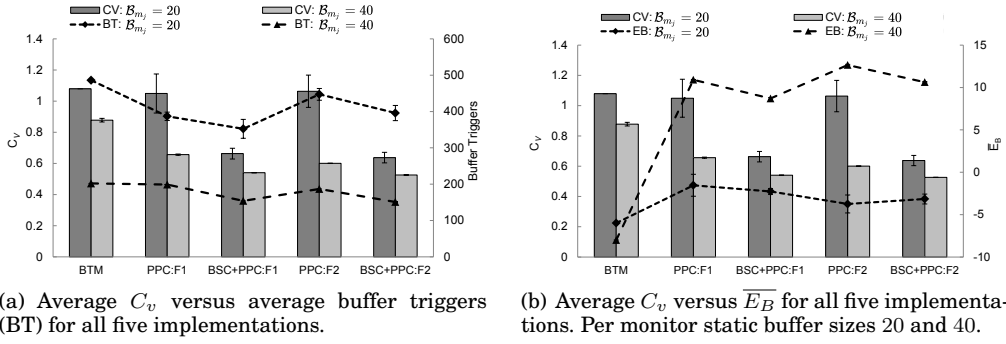


Fig. 12. Averages of both monitors for different buffer sizes.

shown separately, as well as the average C_v of both monitors. In this figure, the static buffer size per monitor is $B_{m_j} = 20$. Upon studying Monitor 1, the figure shows that the improvement in C_v introduced by using PPC only is negligible. However, the significant improvement is introduced by incorporating BSC, which decreases C_v from 1.1 down to 0.68, which is a 38% improvement over a trivial buffer-triggered implementation and a 35% improvement over the PPC design.

The same trend can be extended to BSC+PPC:F2, which scores a $C_v = 0.61$, almost 45% improvement over BTM and 43% improvement over PPC:F2. The larger improvement when using BSC+PPC:F2 is due to its design, which targets minimizing C_v by selecting polling periods closer to their mean. Monitor 2 shows similar results, with BSC+PPC:F2 scoring a 36% improvement over BTM and PPC:F2. Naturally, the average C_v shows favorable results for using BSC. This indicates that utilizing buffer resizing significantly improves time predictability. Figure 12(b) also shows improvements when $B_{m_j} = 40$, however they are less significant. This is due to the saturation of the system by the larger buffer, which is capable of accommodating transient loads without the urgent need for control.

Figure 12(a) confirms the intuition behind the design of BSC, which attempts to decrease the number of buffer triggers to improve predictability. As shown in the figure, C_v reflects the number of buffer triggers, where a 27.5% reduction in buffer triggers results in a 38% reduction in C_v (comparing BSC+PPC:F1 with BTM). In fact, the coefficient of correlation between C_v and the number of buffer triggers across all trials is 0.71. Interestingly, although BSC+PPC:F2 incurs more buffer triggers than BSC+PPC:F1, it performs better in terms of predictability. This is attributed to the aggressiveness of PPC:F2, which although incurs more buffer triggers, they are in proximity with the scheduled invocation, thus causing improved predictability results.

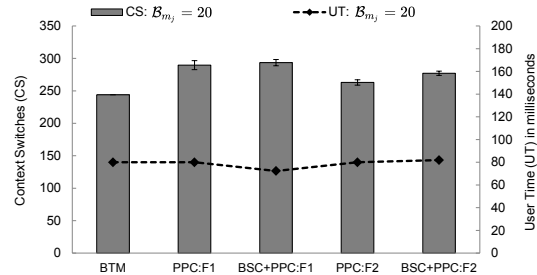


Fig. 13. CS versus UT for the five main implementations.

8.3.2. Memory Utilization. Figure 11(b) shows the average buffer size of each monitor in all five implementations. BTM, PPC:F1, and PPC:F2 have a constant buffer size. However, BSC implementations show that Monitor 2 is slightly dominating Monitor 1 in terms of memory usage. This can be explained by Figure 10(d) which shows that Monitor 2 generally experiences higher load than Monitor 1. BSC adapts to that load

and adjusts the buffer sizes accordingly. The average shown in Figure 11(b) is almost exactly $\mathbb{B}/2$ except for some negligible increases due to the non-uniformity of the time periods for each buffer size readings are extracted.

Figure 12(b) shows the memory utilization (measured using $\overline{E_B}$) of the different implementations versus the coefficient of variation (C_v). For BTM, the memory utilization is the highest by design (lowest $\overline{E_B}$), since the monitor is invoked only when the buffer is full. This explains the negative $\overline{E_B}$ for BTM, since the utilization exceeds the safety limit (see Equation 18). The figure shows that the utilization of BSC+PPC is at least the same as PPC if not better (BSC+PPC:F1). This result further motivates the use of BSC, since the improvement of time predictability does not come at the cost of memory utilization.

8.3.3. CPU Utilization. We measure the user CPU time (UT) used for every experiment. Figure 13 shows the UT results versus the number of context switches (CS) for the five implementations. The results show relatively similar performance in terms of user time, which is expected since fuzzy control imposes minimal computational cost on the system. The number of context switches of BTM is also expectedly the lowest. Using BSC slightly increases the number of context switches, yet that is the cost of maintaining a much more predictable behavior.

9. INSIGHTS GAINED FROM EXPERIMENTS

This section summarizes the insights gained from experiments on single and multiple monitor systems. In a single monitor setting, the Bluetooth payment system (which exhibits sporadic changes in the rate of incoming events) demonstrated a significant improvement using our controllers. Time predictability was improved by 45% versus a simple buffer triggered approach. Memory utilization maintained a value of 70%, while a buffer triggered approach is obviously 100%. This observation suggests that for non-linear systems, our approach is advantageous in terms of time predictability without a drastic negative effect on memory utilization. When applying our approach to a system such as the Laser Beam Stabilizer, where events arrive periodically, some controllers (PPC:F3 and PPC:F4-0.4) causes a slight decrease in time predictability (15 – 20%). Other controllers (PPC:F2 and PPC:F4-0.2) cause an improvement in time predictability due to their aggressiveness in adjusting the polling period and their ability to overcome the noise in OS timers. This shows that our approach is better suited for systems that exhibit sporadic or aperiodic rates of incoming events. Nevertheless, at least per the LBS case study, our technique does not cause a violent reduction in time predictability.

The buffer sizes chosen in the experiments demonstrate a trend of behavior, where the lowest size of 20 is a strict constraint that puts any implementation under stress. At buffer sizes lower than 20 the manipulations of the polling period become so small that they are overshadowed by the noise in the OS timers. On the other hand, the highest size of 60 is relaxed and, thus, no real differences appear between implementations, since the incoming rate of events never fills the buffer. Thus, at very small or very large buffer sizes, a buffer-triggered approach is most suitable due to its simplicity.

In a multiple monitor setting, the introduction of the buffer size controller (BSC) is proved to be advantageous. Our case study of two concurrent monitors demonstrates the applicability of the approach to multiple monitor systems. The approach can be naturally extended to more monitors simply by increasing the available shared buffer space to all monitors, or by hierarchical composition, where a BSC controls the memory distribution of two *sets* of monitors. The results are promising, showing a 40% improvement in time predictability, and above 90% average memory utilization. The significant improvement in memory utilization versus our single monitor solution is

due to dynamic buffer resizing. Similar to the single monitor experiments, the buffer sizes we experiment, serve the purpose of demonstrating a trend based on the engine datasets.

10. RELATED WORK

The focus of traditional event-based runtime monitoring is to reduce the monitoring overhead through improved instrumentation [Dwyer et al. 2007; d’Amorim and Roşu 2005], static analysis [Bodden et al. 2007], and efficient monitor generation [d’Amorim and Roşu 2005]. Chilimbi and Hauswirth [Hauswirth and Chilimbi 2004] propose a control theoretic approach to memory leak detection. They indirectly control overhead by reducing the sampling rate for blocks of execution that generate a higher number of memory accesses. While this approach lowers overhead, it does not enforce an upper bound, and hence a system facing a high surge of events will suffer from overhead problems. Huang, et al. [Huang et al. 2012] propose a control-theoretic software monitoring technique, where cascaded PID controllers are used to temporarily disable monitors in order to keep monitoring overhead below a user-defined threshold. This approach successfully bounds the monitoring overhead, yet results in an unsound monitor, since events are being dropped when a particular monitor is disabled. The paper argues that in some properties dropping events does not affect the verdict of verification, which is true. However, the need for a sound monitor still exists in case dropped events can cause false positives or negatives. To tackle this problem, Bartocci et al. [Stoller et al. 2011] augment the method presented in [Huang et al. 2012] using a hidden Markov model (HMM) to fill in the gaps in event sequences. The paper introduces a method to probabilistically estimate whether a violation occurred during a period where the monitor was disabled. Both these methods require tuning of PID controllers and training of HMM in [Stoller et al. 2011]. Our approach utilizes fuzzy controllers that are more suited to non-linear systems and require less efforts in tuning. Moreover, our approach results in a sound monitor that does not drop events.

The work in [Chen and Roşu 2007; d’Amorim and Havelund 2005] introduces powerful frameworks for runtime verification, where multiple monitors can coexist. However, no guarantees on time predictability of these monitors exist, mainly since they target Java programs and hence real-time constraints generally do not exist.

In the context of time predictability, in time-triggered runtime verification [Bonakdarpour et al. 2013; 2011] a monitor periodically samples the system state and verifies critical properties of the system. The time-triggered approach involves the problem of finding an optimal sampling period to minimize the size of auxiliary memory required, so that the monitor can correctly reconstruct the sequence of program state changes. [Navabpour et al. 2012] uses symbolic execution to compute the sampling period at run time. However, deep static analysis techniques suffer from two drawbacks: they (1) may not scale, and (2) are completely blind to system dynamics and the environment actions at run time, especially in *reactive* systems, which is the characteristic of most cyber-physical systems. This approach is extended in [Navabpour et al. 2015] to the context of component-based multi-core systems.

11. CONCLUSION

In this article, we focused on designing a scalable approach for controlling buffer overshoots in time-predictable runtime monitoring of systems that heavily interact with the physical world. We approached the problem by identifying two types of systems: (1) single monitor systems, and (2) multiple monitor systems. We followed three objectives: soundness, minimum jitter in monitor invocation frequency, and maximum memory utilization. To achieve these objectives in single monitor systems, we proposed a monitoring technique that utilizes control theory to dynamically tweak the polling

period of monitoring such that the three objectives are achieved. Using feedback-based control, the monitoring solution is capable of dynamically improving time predictability. Our case studies demonstrate that our approach significantly improves time predictability in case of sporadic load, and does not negatively impact time predictability in case of uniform load.

In the case of multiple monitor systems, we proposed an additional controller that dynamically (and temporarily) extends the buffer size using a shared space among all monitors in periods of bursts of incoming events. We demonstrated how this design is applicable to systems that heavily interact with the physical world. Our experimental results show that the proposed buffer size controller can prevent up to 27.5% of the overshoots and improve time predictability by 40%. The significance of this result is that these improvements come at no extra memory cost, just well-timed fluctuations of individual monitor's buffer size. Our experiments also show a 0.71 correlation between the number of overshoots and the coefficient of variation in the period of monitor invocations, confirming the intuition behind the buffer size controller design.

For future work, one can incorporate static analysis techniques such as analysis of control-flow graphs and symbolic execution, so controllers are aware of the structure of the system under inspection. Another interesting research direction is to design parallel monitors and controllers that observe different execution threads of time-sensitive concurrent multi-core applications.

REFERENCES

- E. Bodden, L. Hendren, and O. Lhoták. 2007. 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)*. Springer-Verlag, Berlin, Heidelberg, 525–549.
- B. Bonakdarpour, S. Navabpour, and S. Fischmeister. 2011. Sampling-based Runtime Verification. In *Proceedings of the 17th International Symposium on Formal Methods (FM)*. 88–102.
- B. Bonakdarpour, S. Navabpour, and S. Fischmeister. 2013. Time-triggered Runtime Verification. *Formal Methods in Systems Design (FMSD)* 43, 1 (2013), 29–60.
- F. Chen and G. Roşu. 2007. MOP: An Efficient and Generic Runtime Verification Framework. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 569–588.
- S. Colin and L. Mariani. 2005. *Run-Time Verification*. Springer-Verlag LNCS 3472, Chapter 18.
- M. d'Amorim and K. Havelund. 2005. Event-based runtime verification of Java programs. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 1–7.
- M. d'Amorim and G. Roşu. 2005. Efficient monitoring of ω -languages. In *Proceedings of the 17th international conference on Computer Aided Verification (CAV'05)*. Springer-Verlag, Berlin, Heidelberg, 364–378.
- D. Driankov, H. Hellendoorn, and W. Reinfrank. 1993. *An introduction to fuzzy control*. Springer-Verlag New York, Inc., New York, NY, USA.
- M. B. Dwyer, A. Kinneer, and S. Elbaum. 2007. Adaptive Online Program Analysis. In *Proceedings of the 29th international conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 220–229.
- P. Galan. 2003. Temperature control based on traditional PID versus fuzzy controllers. *Nortel Networks Control Software Design Documentation* (2003).
- A. Galati and C. Greenhalgh. 2010. Human mobility in shopping mall environments. In *Proceedings of the Second International Workshop on Mobile Opportunistic Networking*. ACM, 1–7.
- D. Giannakopoulou and K. Havelund. 2001. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings of the IEEE/ACM international conference on Automated Software Engineering (ASE)*. 412–416.
- M. Hauswirth and T. M. Chilimbi. 2004. Low-overhead memory leak detection using adaptive statistical profiling. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 156–164.
- X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok. 2012. Software Monitoring with Controllable Overhead. *Software tools for technology transfer (STTT)* 14, 3 (2012), 327–347.
- H. Kopetz. 1991. Event-triggered versus time-triggered real-time systems. In *Operating Systems of the 90s and Beyond*. Springer, 86–101.
- H. Kopetz and G. Bauer. 2003. The time-triggered architecture. *Proc. IEEE* 91, 1 (2003), 112–126.

- S. Navabpour, B. Bonakdarpour, and S. Fischmeister. 2012. Path-aware Time-triggered Runtime Verification. In *Runtime Verification (RV)*. 199–213.
- S. Navabpour, B. Bonakdarpour, and S. Fischmeister. 2015. Time-triggered Runtime Verification of Component-Based Multi-core Systems. In *Proceedings of the 15th International Conference on Runtime Verification (RV)*. 153–168.
- A. Pnueli and A. Zaks. 2006. PSL Model Checking and Run-Time Verification via Testers. In *Symposium on Formal Methods (FM)*. 573–586.
- D. E. Rivera, M. Morari, and S. Skogestad. 1986. Internal model control: PID controller design. *Industrial & engineering chemistry process design and development* 25, 1 (1986), 252–265.
- T. J. Ross. 2009. *Fuzzy logic with engineering applications*. Wiley.
- S. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. Smolka, and E. Zadok. 2011. Runtime verification with state estimation. In *Proceedings of the Second international conference on Runtime verification*. Springer-Verlag, 193–207.
- J. G. Ziegler and N. B. Nichols. 1942. Optimum settings for automatic controllers. *trans. ASME* 64, 11 (1942).