Sacrificing a Little Space Can Significantly Improve Monitoring of Time-sensitive Cyber-physical Systems

Ramy Medhat
Dept. of Elec. and Comp. Eng.
University of Waterloo
Waterloo, Canada
rmedhat@uwaterloo.ca

Deepak Kumar
Dept. of Elec. and Comp. Eng.
University of Waterloo
Waterloo, Canada
d6kumar@uwaterloo.ca

Borzoo Bonakdarpour
School of Computer Science
University of Waterloo
Waterloo, Canada
borzoo@cs.uwaterloo.ca

Sebastian Fischmeister
Dept. of Elec. and Comp. Eng.
University of Waterloo
Waterloo, Canada
sfischme@uwaterloo.ca

ABSTRACT

The goal of *runtime verification* is to inspect the correctness of a system by incorporating a monitor during its execution. Predictability of time distribution of monitor invocations and memory usage are two indicators of the quality of a monitoring solution, specially in cyber-physical systems, where the physical environment is a part of the system dynamics. In our previous work, we proposed a control-theoretic approach for coordinating time predictability and memory utilization in runtime verification of time-sensitive systems. To this end, we designed controllers that attempt to improve time predictability, while ensuring the soundness of verification by incorporating a maximally utilized bounded memory buffer that accumulates events.

Since the frequency of occurrence of environment actions in cyber-physical systems is not known a priori, the system may run into situations, where the buffer is full, but a monitor invocation has not yet been scheduled. In control theory, this is called the *overshooting* phenomenon, which inherently decreases time predictability. In this paper, we address the issue of overshoots, by employing a second controller that allows limited memory reservations to temporarily extend the size of the event buffer when the system is subject to bursts of environment actions. We apply our solution to the verification of the air/fuel ratio in a car engine exhaust. The acceptable ratio varies depending on the driving circumstances, and monitoring that ratio is important to control emissions in a vehicle. A highly predictable monitor imposes uniform load on the engine control unit (ECU), thus, not negatively or sporadically affecting its control tasks. The experimental results exhibit two significant contributions: we (1) demonstrate the advantages of applying our approach to achieve low variation in the frequency of monitor invocations for verification, while maintaining maximum memory utilization, and (2) clearly illustrate that by negligible temporary increases in the size of the event buffer, the number of overshoots decreases significantly, which in turn substantially increases time predictability of runtime verification.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Monitors*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Control theory*

General Terms

Verification,Performance

Keywords

Runtime Verification, Control Theory, Monitoring, Time Predictability

1. INTRODUCTION

Runtime verification (RV) is a technique that employs tools for monitoring software execution to gain assurance about the correctness of systems. The inherent cost of RV is runtime overhead. In the context of systems that are required to meet soft or hard timing constraints, this cost by itself is not the main obstacle in augmenting a system with RV technology. Rather, the monitoring mechanism must be designed in a non-intrusive fashion; i.e., the monitor should not interfere with the normal timing behavior of the system. To achieve this goal, the monitor should behave in a manner, where it is invoked with some level of time predictability regardless of the system behavior and more importantly environment actions. Cyber-physical systems interact heavily with the physical world, often making it more difficult to predict the behavior of the system due to the unpredictability of the environment. In such systems, time predictability is crucial to avoid the negative impact of sporadic load in verification.

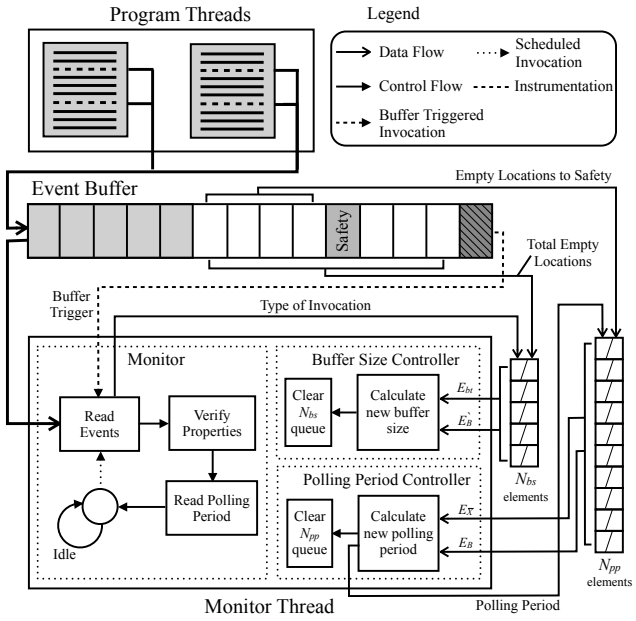

Figure 1: Outline of the proposed controller design.

There exist several approaches in the literature for monitoring real-time systems. In [4, 5], the authors introduce a monitoring approach, where a *time-triggered* monitor polls the state of the system under inspection within fixed time intervals. In order to reconstruct the system state soundly, this technique analyzes the timing characteristics of the code in detail. However, this work falls short in dealing with systems that need to react to the environment actions in a timely fashion at run time. Other approaches (e.g., [16]) only make recommendations on how to design such monitors for real-time systems. Thus, our premise is that we currently lack time-predictable monitoring methods that can deal with reactive systems, where the time and frequency of occurrence of environment actions are unanticipated.

With this motivation, in this paper, we concentrate on designing an RV technique, where the monitor should react to system dynamics and environment actions, while taking resource limitations into account. We require the following:

1. The monitor is invoked within time intervals, called the *polling period*, subject to an upper bound to enforce property violation detection latency. 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, the program under inspection may store events between two subsequent polls in a bounded-size *static* buffer. This buffer is required to be filled with *maximum utilization*.

In [11], we show that the above objectives can be achieved elegantly by designing fuzzy controllers that dynamically determine the polling period to optimize memory utilization and time predictability. The input to this fuzzy controller is the average number of empty locations in the static buffer over a period of time. When the buffer is about to get full, the controller decreases the polling period. Moreover, to maintain soundness, when the static buffer is full, the monitor invocation is automatically triggered ahead of its scheduled invocation. We call this phenomenon, a *buffer trigger* (also called an *overshoot* in control theory).

Obviously, buffer triggers may cause undesirable variations in timing behavior of monitor invocations. Thus, in this paper, we build on our previous results by designing a method that can dynamically extend the buffer size in periods of time that the monitor is overloaded with bursts of incoming events (see Figure 1). We require that the extension to the buffer is (1) temporary, (2) of minimum size, and (3) acceptable if the system indeed has extra unutilized memory space. To this end, we design a second fuzzy controller (see the "Buffer Size Controller" in Figure 1) that reserves memory, if there is space, and releases it after transient overloads. The input to this controller is the average number of empty locations in the buffer as well as the average number of monitor invocations and their types (regular polling vs. buffer trigger) over a period of time. If the system experiences a number of transient buffer triggers, the controller extends the buffer size to avoid additional buffer triggers. This additional buffer space is allocated for monitoring purposes, so that monitors can reserve parts of it to maintain predictable behavior. Moreover, our design gives more priority to the decisions made by the second controller; i.e., extensions to the buffer have higher priority than decreasing the polling period. We demonstrate using experiments that memory reservations by the controller are on average very low, indicating that the additional space is generally available during execution (i.e. for use by other monitors) except during periods of transient load. We emphasize that the design of our fuzzy controllers (i.e., fuzzy sets and membership functions) are quite simple and straightforward and the designer does not need to incorporate sophisticated knowledge about the system in the controller design.

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

1. They support non-linear systems where incoming events can potentially exhibit high variability in short periods of time.

2. They are easy to implement due to the simple mathematics involved in their computation, thus, inducing low processing overhead.

3. Although there are many customizations that can optimize a fuzzy controller, we show that a trivial implementation can show an improvement over an uncontrolled system.

To validate our approach, we conduct a set of thorough experiments on the verification of air/fuel ratio in a Toyota 2JZ engine exhaust. For environmental purposes as well as protecting the engine from failure, a control system is present in vehicles to control the air/fuel ratio. It is crucial to verify the validity of the measured ratio to detect any failures that

could potentially be harmful to the environment or to the engine itself. An engine control unit (ECU) is responsible for managing the many control systems within an engine. Similar to most cyber-physical systems, the ECU output is affected heavily by the physical world. Thus, verification in this system should not impose non-uniform load and should be predictable. Our experiments clearly demonstrate the advantages of employing our simple fuzzy controllers to achieve low variation in the monitor polling period, while maintaining maximum memory utilization in highly non-linear environments. Our GPU-based verification engine [2] is run by our RV tool RiTHM [13] for specifications expressed in the 3-valued semantics of linear temporal logic (LTL_3 [1]). In particular, our results show that the second controller can prevent up to 83% of the overshoots and improve time predictability by a factor of 2. Moreover, the average size of buffer extension is negligible (as low as 2.5%). Our experiments also show a positive correlation coefficient of 0.96 between the number of buffer triggers and the coefficient of variation in the polling period. This observation strongly supports the basis on which the second controller is designed:

> *Decreasing the number of buffer triggers will significantly improve time predictability of runtime monitoring.*

Organization.

The rest of the paper is organized as follows. In Section 2, we formally state our runtime monitoring objectives. Section 3 recaps the basic concepts on fuzzy controllers. Our controller design choices are explained in Section 4. We present our case study and experimental results in Section 5. Related work is discussed in Section 6. Finally, we make concluding remarks and discuss future work in Section 7.

2. FORMAL 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 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. 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 has a preallocated *static buffer* to store events that need to be monitored. However, it may temporarily extend the static buffer by dynamically reserving memory, if the system indeed has unutilized space. We call the static buffer plus the newly reserved memory space the *extended buffer*. The system maintains a memory space to be reserved by monitors suffering high transient loads. This space is also bounded. We require minimum reservations and maximum utilization of the static 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 $\mathbb{B}$.

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 t_{m_0}. Thus, we extend T_m as $t_{m_0} t_{m_1} t_{m_2} \cdots t_{m_k}$. Also, let $B_m = b_{m_1} b_{m_2} \cdots b_{m_k}$ be the output sequence of extended buffer sizes for each monitor invocation.

Since the extended buffer size is decided by the monitor, it should never exceed a given upper bound $\mathbb{B}$, and should not fall below a given lower bound $\mathcal{B}$. The lower bound is the size of the static buffer when the monitor is initialized. Thus, the following condition must hold:

$$\forall\, i \ \in \{1 \cdots k\} : \mathcal{B} \leq b_{m_i} \leq \mathbb{B} \tag{1}$$

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

$$between(\tau_1, \tau_2) = \{e_i \ \mid \ \tau_1 < t_{e_i} < \tau_2\} \tag{2}$$

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

$$\forall\, i \ \in \{1 \cdots k\} : \left|between\left(t_{m_{i-1}}, t_{m_i}\right)\right| \leq b_{m_i} \tag{3}$$

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:

$$\max_{T_m} \left\{ \frac{1}{k} \sum_{i=1}^{k} \frac{\left|between\left(t_{m_{i-1}}, t_{m_i}\right)\right|}{b_{m_i}} \right\} \tag{4}$$

Thus, the objective is to maximize the average memory utilization by maximizing the filling ratio of the buffer ($\left|between\left(t_{m_{i-1}}, t_{m_i}\right)\right| / b_{m_i}$). Since k is a variable over which the formula is maximized, the smaller the k, 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 the extra reservations above the $\mathcal{B}$ lower bound:

$$\min_{B_m} \left\{ \frac{1}{k} \sum_{i=1}^{k} b_{m_i} - \mathcal{B} \right\} \tag{5}$$

Thus, the average size of extra reservations across all runs of the monitor should be minimized.

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:

$$\min \{V(X) \mid \text{for all possible sets of } X\} \tag{6}$$

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.

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 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 3, 4, 5, and 6) simultaneously through identifying T_m (i.e., time of monitor invocations and, hence, k) in a best-effort fashion.

3. BACKGROUND: FUZZY CONTROLLERS

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 [8]. 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.

3.1 Fuzzy Logic

The first function of a fuzzy controller is to transform a discrete measured value called a *crisp* value (e.g., 30° or $1.9m$) to 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:

$$\mu_I(u) = \max_k \{\mu_{A^{(k)}}(x) \cdot \mu_{B^{(k)}}(u)\} \tag{7}$$

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

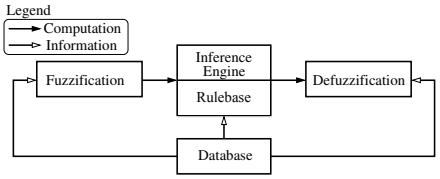

Figure 2: Structure of a Fuzzy Controller [8].

3.2 Structure of a Fuzzy Controller

Figure 2 shows the structure of a typical fuzzy controller [8]. 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 [14]. The lambda-type function is defined as follows:

$$L(x) = \begin{cases} (x-l)/(m-l) & l \leq x \leq m \\ (r-x)/(r-m) & m < x \leq r \\ 0 & \text{otherwise} \end{cases}$$

where l, m, and r denote left, middle, and right points, respectively with $l < m < r$. Similarly, the S-type function is defined as follows:

$$S(x) = \begin{cases} 0 & x \leq l \\ (x-l)/(r-l) & l < x \leq r \\ 1 & x > r \end{cases}$$

Finally, the Z-type function is defined as follows:

$$Z(x) = \begin{cases} 1 & x \leq l \\ (r-x)/(r-l) & l < x \leq r \\ 0 & x > r \end{cases}$$

An example of these functions is shown in Figure 3. 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 pointwise 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 7. The output crisp value u^* is calculated as follows:

$$u^* = \frac{\int_{-\infty}^{+\infty} u \cdot \mu_I(u)\, du}{\int_{-\infty}^{+\infty} \mu_I(u)\, du} \quad (8)$$

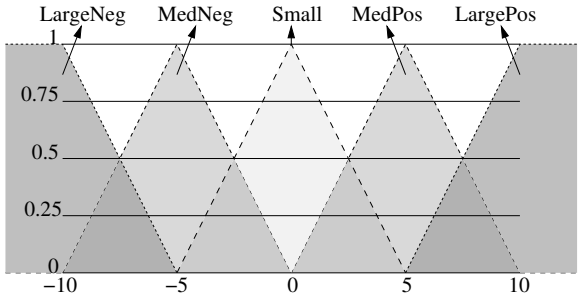

Figure 3: Membership functions of input fuzzy sets.

4. CONTROLLER DESIGN

Recall that Figure 1 shows the general outline of our solution. The program under inspection can be multi-threaded running on a single-core machine. We instrument the program under inspection, so that it enqueues the value of variables of interest in a 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 (the *polling period*) while events are being enqueued in the buffer, and it is scheduled for invocation once the polling period expires. Once invoked, the monitor 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. In order to maintain soundness, no events should be dropped from the buffer. Thus, when the static buffer is full, the monitor is automatically triggered ahead of its scheduled invocation. We call this phenomenon a *buffer trigger*. Our design is based on the collaboration of two controllers: (1) a controller is responsible for managing the polling period (described in Subsection 4.1), and (2) a controller is responsible for managing the buffer size (described in Subsection 4.2).

4.1 Polling Period Controller (PPC)

We now describe two designs for PPC previously presented in [11]: *Fuzzy 1* which is a fuzzy controller attempting to maximize memory utilization (Equation 4), and *Fuzzy 2* which attempts to simultaneously maximize memory utilization and enhance time predictability (Equations 4 and 6).

4.1.1 Fuzzy Controller 1

Since we deal with reactive systems, in principle, buffer-triggered invocations of the monitor are inevitable. Our design supports a safety threshold for buffer utilization. For instance, a controller with 80% safety threshold will attempt to keep the buffer 80% utilized in every monitor invocation. The purpose of having a safety threshold is to decrease the number of buffer triggers. This controller operates within the monitor thread. It is invoked every N_{pp} invocations of the monitor, where N_{pp} is a design parameter. In Figure 1, PPC buffers the number of empty locations of the last N_{pp} invocations, to be used to calculate the average when the controller is invoked. The design of the controller is as follows:

- **Input.** The input to the controller is the fuzzy variable E_B representing the average number of empty locations in the buffer in the last N_{pp} monitor invocations. The crisp value for this variable is calculated as

follows:

$$E_B(j) = \frac{1}{N_{pp}} \times \left\{ \sum_{i=(j-1)\cdot N_{pp}}^{j \cdot N_{pp}} b_{m_i} \times S_{m_i} - \left| between\left(t_{m_{i-1}}, t_{m_i}\right) \right| \right\} \quad (9)$$

where j is the index of the controller invocation, i is the index of the monitor invocation, b_{m_i} is the size of the buffer at monitor invocation i, and S_{m_i} is the safety threshold at monitor invocation i. There are 5 fuzzy sets for the error variable based on lambda-type functions as shown in Figure 4. The *Small* set has a peak at zero error, with the left x-intercept at

$$\frac{-b_{m_i}(1 - S_{m_i})}{2}$$

and the right x-intercept at

$$\frac{b_{m_i} \times S_{m_i}}{2}$$

The reason these points are not symmetric is that the largest positive error that could be reached is $b_{m_i} \times S_{m_i}$, which denotes that the buffer is completely empty. However, the largest negative error is $-b_{m_i}(1 - S_{m_i})$, 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, 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 runtime.

4.1.2 Fuzzy Controller 2

This controller targets both memory utilization and time predictability. The approach of this controller is to balance between choosing a polling period that would minimize the number of empty locations in the extended 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.

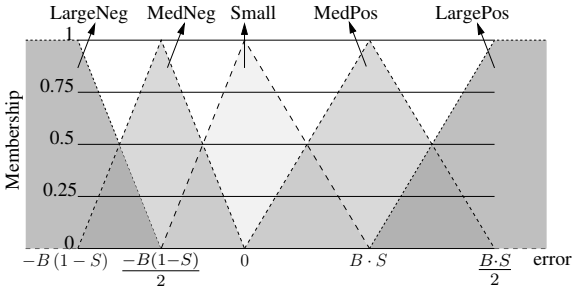

Figure 4: Membership functions of E_B fuzzy sets.

- **Input.** In addition to E_B, we introduce a new fuzzy variable $E_{\bar{X}}$ to control the polling period variance. Polling period here refers to the period of time between two monitor invocations, regardless of whether the invocations were scheduled or buffer triggered. The reason for this is twofold: (1) The controller needs to account for buffer triggers in its calculations to stabilize the polling period, and (2) scheduled invocations are not always accurate due to system noise. $E_{\bar{X}}$ represents the difference between the average polling period in the last N_{pp} invocations and the mean of all previous polling periods. As in Fuzzy 1, the polling period is buffered in an N_{pp} element buffer to calculate the average. The crisp values of $E_{\bar{X}}$ is calculated as follows:

$$E_{\bar{X}}(j) = \frac{\frac{1}{N} \sum_{i=(j-1)\cdot N_{pp}}^{j \cdot N_{pp}} X_i - \bar{X}}{\bar{X}}$$

where i is the index of the monitor invocation, X_i is the ith polling period, j is the index of the controller invocation 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. 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 the polling periods, and consequently high time predictability.

- **Output.** The output of the controller is the same as Fuzzy 1.

- **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 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,

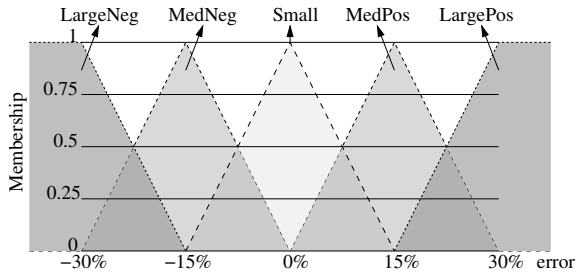

Figure 5: Membership functions of $E_{\bar{X}}$.

Small, MedPos, and LargePos, respectively. The mapping above is symmetric and 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.

4.2 Buffer Size Controller (BSC)

We expect the frequency of buffer triggers to affect the time predictability of the system. 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 cannot reserve more memory than the upper bound $\mathbb{B}$ (e.g., the preallocated space available for monitoring), and cannot release memory beyond the lower bound $\mathcal{B}$ (i.e., its static buffer).

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, suppose that the buffer size is 20, and the safety threshold is 80% (as set in Subsection 4.1). 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 decreases the number of buffer triggers using the extra memory without affecting the polling period controller, since the target is still

		$E_{\bar{X}}$ **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

Table 1: Symmetric mapping of input variables in if-then rules.

16. The only difference is that PPC is under less pressure to change the polling period.

The design of the controller is as follows:

- **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:

$$E'_B(j) = \frac{1}{N_{bs}} \left\{ \sum_{i=(j-1)\cdot N_{bs}}^{j \cdot N_{bs}} \frac{\left| between\left(t_{m_{i-1}}, t_{m_i}\right)\right|}{b_{m_i}} \right\}$$

  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% (Figure 6). 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, 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 2 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 5. 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 thus, no jitter due to these invocations. Since BSC performs its processing every N_{bs} invocations of the monitor, it performs its processing after the verification process is completed every N_{bs}^{th} invocation. Since it has a higher priority than PPC, it performs its processing first in the case where both controllers coincide.

  The N_{bs} parameter determines how responsive the controller to change. A smaller number indicates that

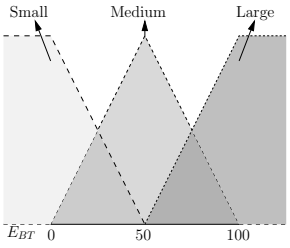

Figure 6: Membership functions of E_{BT}.

BSC reacts quicker to sudden buffer triggers. In our experiments section we 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 6, 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 6 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.

5. IMPLEMENTATION AND EXPERIMENTAL RESULTS

In order to analyze the performance of our controllers, we conduct a case study on a cyber-physical system, the air/fuel ratio in the exhaust of a Toyota 2JZ engine. The case study involves using different controllers with different configurations. Experimental background, settings and results are presented in Subsections 5.1, 5.2, and 5.3, respectively.

5.1 Experimental Background

Environmental concerns require diligent monitoring of air/fuel ratio in a vehicles exhaust. The *Lambda value* is an industry standard value calculated as 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 that the measured air/fuel ratio is exactly the same as that of a perfect combustion. If the value is less than 1.00, this indicates that the amount of fuel in the exhaust is higher, which is known as *rich*. If greater than 1.00, it is known

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

Table 2: Mapping of input variables in if-then rules for the buffer size controller.

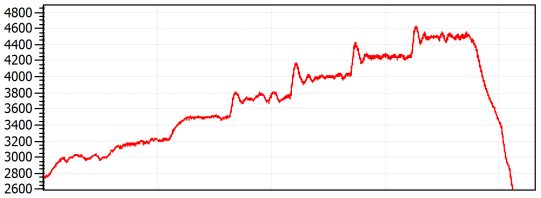

Figure 7: RPM of engine in a 43 seconds long step test.

as *lean*. A vehicle running rich is less environment friendly, due to the higher percentage of hydrocarbons and carbon monoxide present in the exhaust. However, a highly lean condition indicates a possible misfire which could result in serious engine damage.

A control system is required to maintain a safe lambda value, that protects the environment as well as the engine. Depending on driving conditions, the acceptable values differ. 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) controls all actuators in the engine using a multitude of sensors, one of which is the lambda sensor. The ECU software determines how rich or lean the engine should run, such that the desired level of emissions is maintained and the required performance is achieved. Failure to do so can be caused by multiple reasons, including malfunctioning sensors, or bad 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.

5.2 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. As mentioned above, the verification software should not exhibit unpredictable behavior that would negatively impact the ECU. 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 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. 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 7. The lambda sensor readings of the engine are shown in Figure 8. A photograph of the experimental setup is shown in Figure 9.

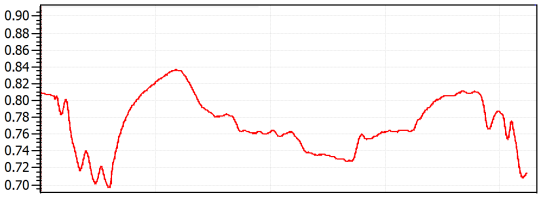

Figure 8: Lambda values during the step test.

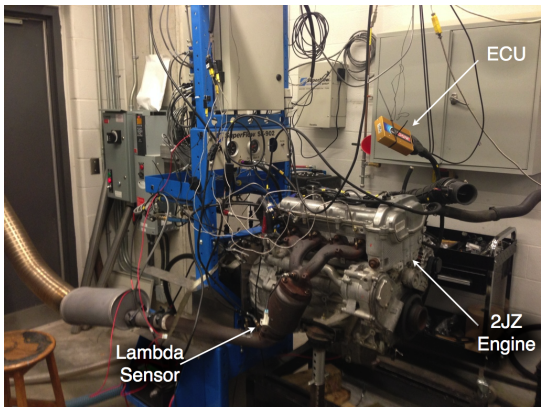

Figure 9: The 2JZ engine in the engine test room.

Our experiments are designed based on five factors:

1. **Type of PPC.** We incorporate Fuzzy 1 and Fuzzy 2 controllers introduced in Section 4.1.
2. **BSC enable status.** We experiment with enabling and disabling the buffer size controller.
3. **Static buffer size ($\mathcal{B}$).** We consider two different static buffer sizes: 20 and 40.
4. **Invocation frequency.** Recall from Figure 1 that the controllers build a history of previous input signals. We experiment with varying the invocation frequency of both the polling period controller and the buffer size controller. We configure the controllers with 2 different combinations of frequencies: $(5, 5)$ and $(10, 5)$ for the polling period controller and the buffer size controller, respectively.
5. **Biased membership functions.** We consider changing the center point of the E_{bt} membership functions from 0.5 to 0.2 (see Section 4.2). This is in an attempt to force the controller to be more conservative about buffer triggers. We study the effect of this change on the different metrics.

Hence, there is a total of 32 configurations to test all different combinations of the above four factors. We carry out 10 runs (replicates) with randomization to provide statistical confidence in the results.

The five measurement metrics that we observe are:

1. **Error mean ($\overline{E_B}$).** 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. This metric is based on Equation 4.
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, the more predictable the timing behavior of the monitor. This metric is based on Equation 6.
3. **Context switches (CS).** This is the number of context switches due to 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 (BT).** 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. Thus, a lower number of buffer triggers indicates that the controller is adapting better to the non-linearity of the external events.
5. **Average buffer size ($\overline{B}$).** 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. This metric is based on Equation 5.

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.

5.3 Analysis of Results

This section analyzes the results of our experiments with respect to the metrics discussed in Subsection 5.2. Since there are 34 possible combinations of the experimental factors, we first present only a subset of these combinations that are most significant. Then, we present the remaining combinations and demonstrate the reasons for selecting a subset. 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 with the Fuzzy 1 design (recall that this is the design in our previous work [11]).
- *(PPC:F2)* In this implementation, only PPC is used, with the Fuzzy 2 design (recall that this is also the design in our previous work [11]).
- *(BSC+PPC:F1)* This design implements BSC in addition to PPC, with the Fuzzy 1 design.
- *(BSC+PPC:F2)* In this implementation, BSC is used in addition to PPC, with the Fuzzy 2 design.

Time Predictability

Figure 10 shows the polling period coefficient of variation (C_v) versus the mean number of empty locations in the

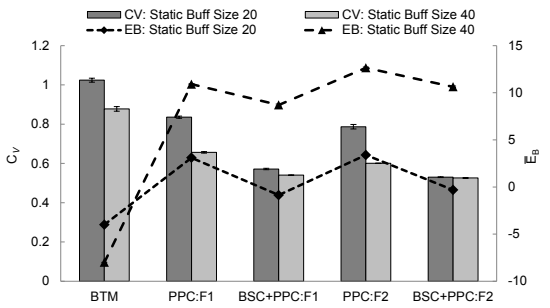

Figure 10: C_v versus $\overline{E_B}$ for the five main implementations.

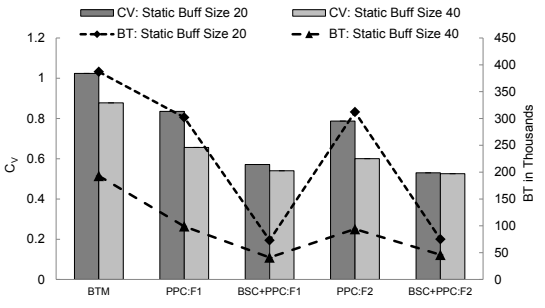

Figure 11: C_v versus BT for the five main implementations.

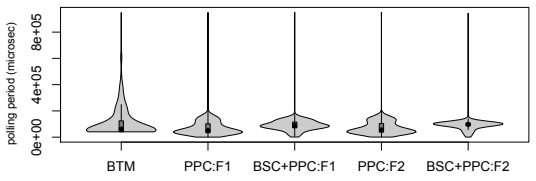

Figure 12: Violin plot of polling periods for the five main implementations.

buffer ($\overline{E_B}$) for the five different implementations mentioned above. For static buffer size $\mathcal{B} = 20$, the figure shows the improvement in C_v introduced by using PPC only (i.e., the design in [11]) compared to a buffer triggered monitor, which in the case of Fuzzy 1, causes a drop in C_v from 1.02 to 0.83. However, the more significant improvement is introduced by incorporating BSC, which decreases C_v further down to 0.57, which is a 45% improvement over a trivial buffer-triggered implementation and a 31% improvement over the PPC design in [11].

The same trend can be extended to Fuzzy 2, which scores a $C_v = 0.52$, almost 50% improvement over BTM and 34% improvement over PPC:F2. The larger improvement when using Fuzzy 2 is due to its design, which targets minimizing C_v by selecting polling periods closer to their mean. Figure 10 also shows similar results when $\mathcal{B} = 40$. An interesting point is that in BSC+PPC, the improvement still exists yet is not as significant as when the initial buffer size is 20. This is attributed to the nature of non-linearity in the measured lamda readings, which naturally becomes more stable at a buffer size of 40.

Figure 11 shows a significant decrease in the number of buffer triggers (BT) when BSC is used. For BSC+PPC:F1, the number of buffer triggers is 66. Compared to BTM which has 387 buffer triggers, BSC+PPC:F1 is preventing 83% of the buffer triggers. Figure 11 also shows the relationship between C_v and the number of buffer triggers (BT). The correlation coefficient of C_v and BT is 0.96, which confirms a strong positive relationship. This supports our hypothesis basis on which the BSC controller is designed: decreasing the number of buffer triggers will improve the time predictability of the monitor.

Figure 12 shows the distribution of polling periods for the five main implementations. The figure shows that BTM has the highest variability and dispersion. PPC:F1 has a narrower range, yet wide variability within that range. BSC+PPC:F1 shows improvement in the variability of polling periods. BSC+PPC:F2 shows a significant improvement over its PPC counterpart, and also over its F1 counterpart. This is due to it leveraging the aggressiveness of Fuzzy 2 with the polling period in its favor, producing the tighter distribution.

Memory Utilization

Figure 10 shows the memory utilization (measured using $\overline{E_B}$) of the different implementations. 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 9). 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.

Extended Buffer Size

Table 3 shows the average size of the buffer ($\overline{B}$) for both BSC+PPC:F1 and BSC+PPC:F2, versus both static buffer sizes 20 and 40. For Fuzzy 1, the extended buffer size scales with the increase in the size of the static buffer, which shows that the controller attempts to stabilize around the extended buffer size suitable for the rate of external events. For Fuzzy 2, the extended buffer size is only 4 more locations for buffer size 20 and one more location for buffer size 40. Referring to Figure 10 for $\mathcal{B} = 20$, C_v of BSC+PPC:F2 is 0.52, whereas C_v of PPC:F2 is 0.79. This shows that adding BSC to PPC:F2 decreases C_v by approximately a third, at almost no cost of extra memory. The controller achieves this result by adapting to the external events and reserving extra memory at critical points in time to preserve predictability.

CPU Utilization

We measure the user CPU time (UT) used for every experi-

	Static Buffer Size $\mathcal{B}$	
	20	**40**
BSC+PPC:F1	35	47
BSC+PPC:F2	24	41

Table 3: Average buffer size $\overline{B}$ results.

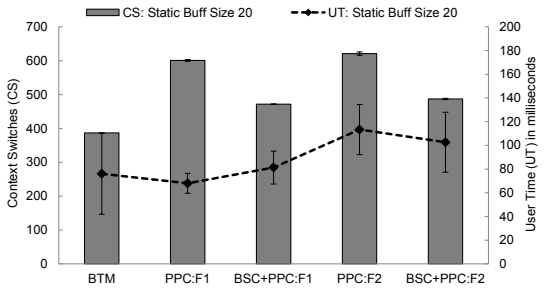

Figure 13: CS versus UT for the five main implementations.

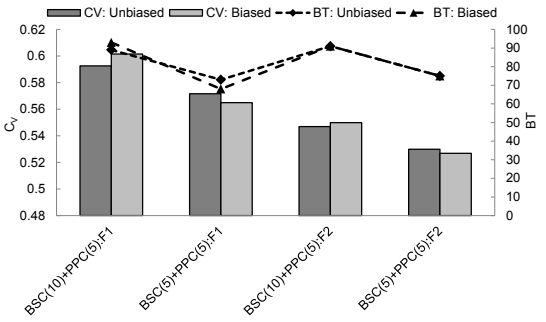

Figure 14: A comparison of biased versus unbiased membership functions.

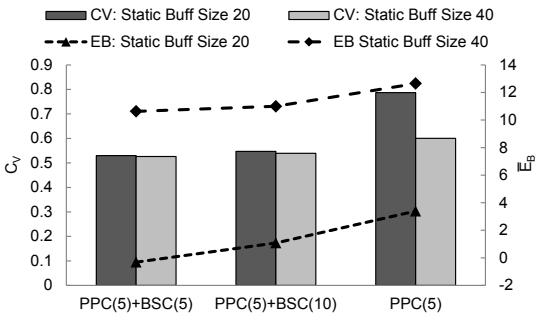

Figure 15: A comparison of different invocation frequencies of BSC.

ment. Figure 13 shows the UT results versus the number of context switches (CS) for the five implementations. The figure exhibits relatively large error bars, due to factors such as cache misses and paging that affect the UT of the monitor. However, a conclusion that could be drawn is that system noise is a far more deciding factor in UT than the actual different implementations. This is attributed to the simplicity of fuzzy controllers, in that the processing involved is always a fixed number of basic arithmetic operations. Thus, adding a second controller (BSC) is not expected to have a large impact on time.

BSC Design Parameters

There are two experimental factors that are specific to the BSC+PPC implementations: (1) Biased versus unbiased membership functions, and (2) invocation frequency. Figure 14 shows the C_v and BT of biased and unbiased versions of BSC+PPC. We compare two invocation frequencies: 5-5 and 10-5 for BSC and PPC, respectively, and also the two designs for PPC:F1 and PPC:F2. As shown in Figure 14, although there is improvement in C_v introduced by using the biased implementation, that improvement is minimal and only visible in cases where both controllers are running at the same frequency. The reason for this is that if BSC is running slower, its bias has a less significant effect on the output.

The percentage improvement in C_v is at most 6%. This result demonstrates that although tweaking the membership functions can introduce improvements, satisfactory results can be achieved using basic membership functions and a trivial implementation.

Figure 15 shows the C_v for different invocation frequencies when PPC is using the Fuzzy 1 design. The figure shows the trend of C_v when the invocation frequency of BSC is decreased while fixing the invocation frequency of PPC. Three data points are compared: $N_{BS} = 5$ (meaning BSC is invoked every 5 monitor invocations), $N_{BS} = 10$ (every 10 invocations), and $N_{BS} = \infty$ (no invocation of BSC at all). The results show a consistently increasing trend for both static buffer sizes. Thus, if PPC is operating at a higher frequency than BSC, granting it higher priority than BSC, the improvement that should be introduced by BSC diminishes. This is due to PPC reacting faster to changes in external events before BSC can adjust the extended buffer size.

Based on these results, the elected implementation for BSC+PPC is an unbiased implementation with 5-5 invocation frequency for BSC and PPC, respectively.

6. RELATED WORK

The main focus of classic event-based runtime verification is to reduce the monitoring overhead through improved instrumentation [6,9], static analysis [3], and efficient monitor generation [7]. Huang, et al. [10] propose a control-theoretic based software monitoring technique. In this work, cascaded PID controllers are used to temporarily disable monitors in order to keep monitoring overhead below a user-defined threshold. This results in an unsound monitor, since events are dropped when a particular monitor is disabled. To tackle this problem, Bartocci et al. [15] augment the method presented in [10] with a Hidden Markov Model (HMM) to fill in the gaps in event sequences. However, both these methods require tuning of PID controllers and training of HMM in [15].

In the context of time-predictability, in time-triggered runtime verification [5] 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. [12] uses symbolic execution to compute the sampling period at run time. However, static analysis does not scale well in large systems, and the technique is inapplicable in reactive non-linear systems.

7. CONCLUSION

In this paper, we focused on designing a scalable approach for controlling buffer overshoots in time-predictable runtime verification of systems that heavily interact with the physical world. We followed three objectives: soundness, minimum jitter in monitor invocation frequency, and maximum memory utilization. To this end, we proposed a monitoring technique that utilizes two fuzzy controllers: one that controls the period of invocation of the monitor and another that dynamically (and temporarily) extends the buffer size in periods of time that the monitor is overloaded with bursts of incoming events. We demonstrated how this design is applicable to cyber-physical systems that rely heavily on interaction with the physical world. Our experimental results show that the proposed controller can prevent up to 83% of the overshoots and improve time predictability by a factor of 2. Moreover, the average size of buffer extension is negligible (as low as 2.5%). Our experiments also show a very strong correlation between the number of overshoots and the coefficient of variation in the period of monitor invocations.

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. We are also working on developing time-predictable runtime monitors that are also power-efficient.

8. ACKNOWLEDGMENTS

We would like to thank the Engine test facility of the Faculty of Engineering's Sedra Student Design Center, University of Waterloo, for their efforts during the implementation of the case study of this work. The Engine test facility provided us with the necessary guidance in operating the engine, as well as the execution of test cases and collection of data. This facility is a strong platform for research in engine control, leveraging advanced testing equipment. We would like to specifically thank Stephanie Bruinsma and Peter Teertstra for their amazing support.

This research was supported in part by NSERC Discovery Grant 418396-2012, NSERC Strategic Grant 430575-2012, NSERC DG 357121-2008, ORF-RE03-045, ORF-RE04-036, ORF-RE04-039, CFI 20314, CMC, and the industrial partners associated with these projects.

9. REFERENCES

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

[2] S. Berkovich, B. Bonakdarpour, and S. Fischmeister. Gpu-based runtime verification. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 1025–1036. IEEE, 2013.

[3] 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.

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

[5] B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Time-triggered runtime verification. *Formal Methods in System Design*, pages 1–32, 2012.

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

[7] 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.

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

[9] M. B. Dwyer, A. Kinneer, and S. 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.

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

[11] R. Medhat, D. Kumar, B. Bonakdarpour, and S. Fischmeister. Runtime verification with controllable time predictability and memory utilization. Technical Report CS-2013-02, University of Waterloo, 2013.

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

[13] S. Navabpour, Y. Joshi, C. W. W. Wu, S. Berkovich, R. Medhat, B. Bonakdarpour, and S. Fischmeister. RiTHM: a tool for enabling time-triggered runtime verification for c programs. In *ACM Symposium on the Foundations of Software Engineering*, pages 603–606, 2013.

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

[15] S. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. Smolka, and E. Zadok. Runtime verification with state estimation. In *International Conference on Runtime Verification (RV)*, 2011.

[16] C. Watterson and D. Heffernan. Runtime verification and monitoring of embedded systems. *IET Software*, 1(5):172–179, 2007.