# Reducing Monitoring Overhead by Integrating Event- and Time-triggered Techniques

Chun Wah Wallace Wu[1], Deepak Kumar[1], Borzoo Bonakdarpour[2], and Sebastian Fischmeister[1]

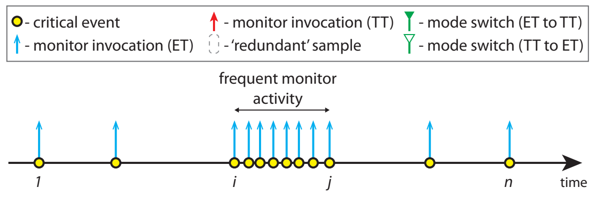
[1] Department of Electrical and Computer Engineering
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada, N2L 3G1
{cwwwu, d6kumar, sfischme}@uwaterloo.ca

[2] School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada, N2L 3G1
borzoo@cs.uwaterloo.ca

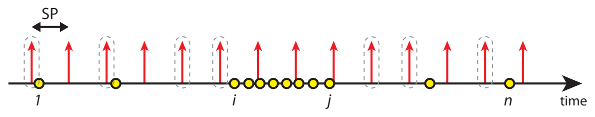
**Abstract.** *Runtime verification* is a formal technique used to check whether a program under inspection satisfies its specification by using a runtime monitor. Existing monitoring approaches use one of two ways for evaluating a set of logical properties: (1) *event-triggered*, where the program invokes the monitor when the state of the program changes, and (2) *time-triggered*, where the monitor periodically preempts the program and reads its state. Realizing the former is straightforward, but the runtime behaviour of event-triggered monitors are difficult to predict. Time-triggered monitoring (designed for real-time embedded systems), on the other hand, provides predictable monitoring behavior and overhead bounds at run time. Our previous work shows that time-triggered monitoring can potentially reduce the runtime overhead provided that the monitor samples the program state at a low frequency.
In this paper, we propose a *hybrid* method that leverages the benefits of both event- and time-triggered methods to reduce the overall monitoring overhead. We formulate an optimization problem, whose solution is a set of instrumentation instructions that switches between event-triggered and time-triggered modes of monitoring at run time; the solution may indicate the use of exactly one mode or a combination of the two modes. We fully implemented this method to produce instrumentation schemes for C programs that run on an ARM Cortex-M3 processor, and experimental results validate the effectiveness of this approach.

## 1 Introduction

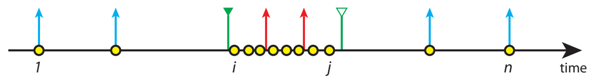
*Runtime verification* [5, 19, 26] is a technique, where a *monitor* checks at run time whether or not the execution of a system under inspection satisfies a given



(a) Event-triggered monitoring.



(b) Time-triggered monitoring.



(c) Hybrid monitoring.

**Fig. 1.** Comparing different methods of monitoring.

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

In [2,3], we introduced a *time-triggered* (TT) method that makes the runtime overhead controllable and predictable, and makes monitoring tasks schedulable. In this method, a monitor samples the state of the program in periodic time intervals. The period, known as the *sampling period* (SP) is such that the monitor misses no critical events. Time-triggered monitoring is especially desirable for designing real-time embedded systems, where time predictability plays a central role. Figure 1(b) shows the interactions that occur between the program and a TT monitor. To decrease the sampling frequency and thus decrease the overhead, we introduced a technique, where the program stores critical events in a history buffer and the monitor reads this buffer to evaluate properties with respect to all state changes stored in the history [2, 3]. From the figure, it is evident that the monitoring activity between events $i$ and $j$ is significantly less than what an event-triggered monitor would require. However, for the sampling period adopted in this example, there are some 'redundant' samples that the monitor takes; a *'redundant' sample* is an invocation of the monitor, where there are no events to process in the buffer. The dashed ovals in Figure 1(b) mark the redundant samples in this example. Although our goal in [2, 3] was tackling the unpredictability of runtime overhead, we observed that time-triggered runtime verification (TTRV) may also reduce the cumulative runtime overhead effectively.

From Figures 1(a) and 1(b), it is evident that both event- and time-triggered monitoring techniques have some advantages and disadvantages with respect to the monitor's execution overhead. Event-triggered monitoring tends to be advantageous in situations, where critical events occur sparsely since the monitor is active only when the program encounters a critical event; time-triggered monitoring tends to be better when many critical events to process within a short time frame.

With this motivation, in this paper, we propose a novel technique based on static analysis that exploits the benefits of both ETRV and TTRV to reduce the runtime overhead, which we call *hybrid* runtime verification (HyRV). Our goal is to supply a program under scrutiny with a monitor that supports both ET and TT modes of operation. The program switches from one mode to another at run time depending upon the current execution path. HyRV automatically obtains the locations to *switch modes* in the program by solving an optimization problem; this method accounts for all monitoring and switching costs in terms of execution time overhead. The main challenge in formulating the optimization problem is threefold:

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

The solution to the problem is an instrumentation scheme for a program that may switch monitoring modes at runtime. For instance, in Figure 1(c), the reduction in monitoring activity will likely reduce the overall monitoring execution overhead. Obviously, using hybrid monitoring will incur overhead costs in performing mode switches. In this example, a mode switch occurs right before $i$ and right after $j$ to switch from ET to TT and TT to ET monitoring modes, respectively.

We implemented this technique in a toolchain that leverages static analysis techniques and integer linear programming (ILP) to solve the optimization problem. The input to our toolchain is a C program and a set of variables to monitor. The toolchain outputs the program source code augmented with the instrumentation scheme that may toggle the monitoring mode at runtime to reduce the monitoring overhead. Currently, our toolchain does not include static analysis of library calls. The results of our experiments on a benchmark suite for real-time embedded programs strongly validate the effectiveness of our technique.

*Organization* The rest of the paper is organized as follows. Section 2 describes the concepts of ETRV and TTRV. Section 3 introduces the HyRV optimization problem. We analyze the results of our experiments in Section 4. Section 5 discusses the related work. Finally, in Section 6, we make concluding remarks and discuss future work.

## 2 Background

Let $P$ be a program under inspection and $\Pi$ be a logical property (e.g., in LTL), where $P$ is expected to satisfy $\Pi$. Let $\mathcal{V}_\Pi$ denote the set of variables that participate in $\Pi$. In *event-triggered runtime verification* (ETRV), the instrumented version of $P$ invokes the monitor to evaluate $\Pi$ whenever the value of some variable in $\mathcal{V}_\Pi$ changes.

In *time-triggered runtime verification* (TTRV) [2, 3], a monitor *samples* the value of variables in $\mathcal{V}_\Pi$ periodically and evaluates $\Pi$. Accurate reconstruction of states of $P$ between two consecutive samples is the main challenge in using this mechanism; e.g., if the value of a variable in $\mathcal{V}_\Pi$ changes more than once between two samples, then the monitor may fail to detect violations of $\Pi$. TTRV usually leverages control-flow analysis to reconstruct the states of $P$.

To ensure that the behaviour of a time-triggered monitor is correct, the monitor must sample at a 'safe' rate determined by statically analyzing $P$'s control-flow graph:

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

- *$V$: is a set of* vertices, *each representing a basic block of $P$. Each basic block consists of a sequence of instructions in $P$.*
- *$v^0$: is the* initial vertex *with in-degree 0, which represents the initial basic block of $P$.*
- *$A$: is a set of* arcs *$(u, v)$, where $u, v \in V$. An arc $(u, v)$ exists in $A$, if and only if the execution of basic block $u$ immediately leads to the execution of basic block $v$.*
- *$w$: is a function $w : A \to \mathbb{N}$, which defines a* weight *for each arc in $A$. The weight of an arc is the* best-case execution time *(BCET) of the source basic block.*
- *$v^f$: is a dummy vertex which acts as final vertex. It has incoming arcs from all actual final vertices. This helps in simplifying analysis by allowing us to easily consider weight of final vertices.*

For example, consider the C program in Figure 2 [2]. Figure 3(a) shows the resulting CFG assuming that the BCET of each line of code is one time unit. Vertices of the graph in Figure 3 list the corresponding line numbers of the C program in Figure 2.

To identify the *sampling period* that a monitor can accurately reconstruct program states between two samples, we modify $CFG_P$ as follows:

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

**Fig. 2.** A simple C program.

**Step 1: Identify Critical Vertices**

We ensure that each *critical instruction* (i.e., an instruction that modifies a variable in $\mathcal{V}_\Pi$) is in a basic block that contains no other critical instructions. We refer to such a basic block as a *critical basic block* or *critical vertex.* For example, in Figure 2, if variables `b`, `c`, and `d` are in $\mathcal{V}_\Pi$, then lines 5, 6, and 9 are critical instructions. Since instructions in lines 5 and 6 are critical and they both reside in basic block $c$, we split $c$ into $c_1$ and $c_2$ as shown in Figure 3(b); the highlighted vertices in the figure denote the critical basic blocks.

**Step 2: Calculate the Longest Sampling Period**

As mentioned earlier, the main challenge in using TTRV is accurate program state reconstruction. To preserve all critical program state changes, the monitor must sample at a rate that can capture all possible critical state changes of $P$ at run time. The corresponding sampling period is called the *longest sampling period* (LSP). Definition 2 formally defines LSP.

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

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

Intuitively, the LSP is the minimum timespan between two successive changes of any two variables in $\mathcal{V}_\Pi$. This means that the minimum distance between all pairs of critical vertices in *CFG* is the LSP. For example, the LSP of the CFG shown in Figure 3(c) is $LSP = 1$, as indicated in the figure. All property violations can be detected if the monitor samples with a period of $LSP$ [2].

**Step 3: Increase the Sampling Period using Auxiliary Memory**

To increase the longest sampling period (and, hence, decrease the involvement of the monitor), we use auxiliary memory to buffer critical state changes between two consecutive samples. Precisely, let $v$ be a critical vertex in a control-flow

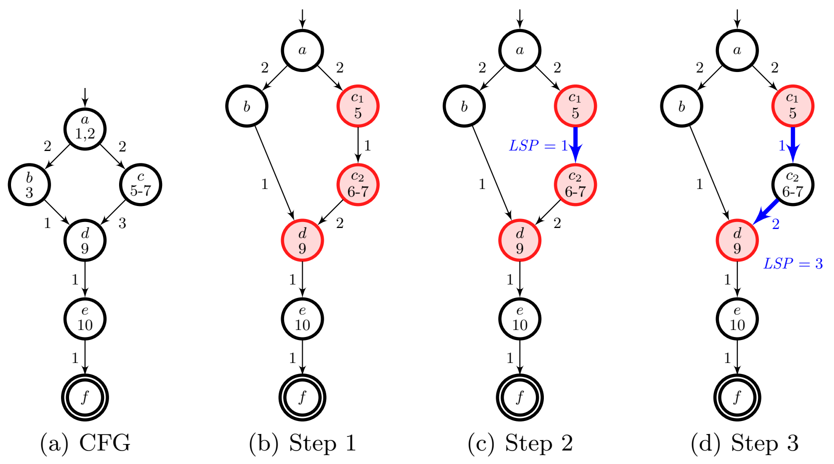


**Fig. 3.** Steps for obtaining optimized instrumentation and sampling period.

graph, *CFG*, where critical instruction *inst* in $v$ changes the value of a variable $a \in \mathcal{V}_\Pi$. We insert an instruction $inst' : a' \leftarrow a$ immediately following *inst*, where $a'$ is an auxiliary memory location, to the sequence of instructions corresponding to vertex $v$. After instrumenting (i.e., adding $inst'$) $v$, $v$ is no longer a critical basic block (i.e., $v \in V \setminus V_c$) because the added instruction guarantees that the monitor will observe this change when it processes the history stored in auxiliary memory. For example, instrumenting vertex $c_2$ in Figure 3(c) by adding an instruction of the form '`ch = c`' directly after line 6 of the program results in the CFG shown in Figure 3(d). Instrumenting the critical instruction in $c_2$ effectively increases the LSP to 3 because of the buffered event. The maximum *violation detection latency* (i.e., the time elapsed between the occurrence of a property violation and the detection of the violation) of $\Pi$, the availability of auxiliary memory and other system constraints limit the number of times we can apply step 3 to increase the LSP.

## 3 Hybrid Event-triggered and Time-triggered Runtime Verification

In this paper, our goal is to select the monitoring scheme that minimizes the expected total overhead incurred from executing the monitor. In order to formally introduce the problem statement, we need to define the underlining monitoring overhead cost model.

### 3.1 Overhead Runtime Costs

Broadly, we classify the *overhead costs* incurred from monitoring into three categories:

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

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

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

Note that these costs are derived in terms of best-case execution time of the corresponding instructions. In particular, we calculate these costs in the same fashion we obtain the arc weights of a control-flow graph (see Definition 1).

## 3.2 Problem Definition

Let $G = \langle V, v^0, A, w, v^f \rangle$ be the control-flow graph of program $P$ and $V_c \subseteq V$ be the set of critical vertices after computing the longest sampling period LSP through application of 3 steps given in Section 2. We are also given five elementary costs $c_{ET}, c_{hist}, c_{TT}, c_{E\rightarrow T}$, and $c_{T\rightarrow E}$ as defined in Subsection 3.1. Assuming all execution paths in $G$ are equally likely, our goal is to find a HyRV monitoring scheme $M$, such that $M_o(G)$ (monitoring overhead of $M$) is minimum. A HyRV monitoring scheme is

$$M : V \rightarrow \{0, 1\} \tag{1}$$

Where 0 denotes that vertex should be monitored using ET monitor whereas 1 indicates TT monitor should be used to monitor the vertex. Note that to uniquely determine the location of a switch, we take domain of $V$ rather than just $V_c$. For a given path $\pi = v^0 \rightarrow v_1 \rightarrow \cdots \rightarrow v^f$ of $G$, the overhead of a monitoring scheme is defined as:

$$\begin{aligned} M_o(\pi) = &\sum_{v \in V_c} [c_{ET} \cdot (1 - M(v)) + c_{hist} \cdot M(v)] \\ &+ \sum_{(v_1, v_2) \in A} [c_{E\rightarrow T} \cdot (1 - M(v_1)) \cdot M(v_2) + c_{T\rightarrow E} \cdot M(v_1) \cdot (1 - M(v_2))] \\ &+ \sum_{\substack{\delta = \langle v_i \rightarrow \ldots \rightarrow v_j \rangle, \\ \delta \in \Delta_\pi}} \left[ c_{TT} \cdot \left( \lceil \frac{\sum_{k=i}^{k=j} w(v_k)}{LSP} \rceil \right) \right] \end{aligned} \tag{2}$$

Where $\Delta_\pi$ is set of longest subpaths of $\pi$ whose vertices are monitored using TT scheme. Three sums in equation 2 correspond to $C_{event}, C_{switch}$, and $C_{sample}$ costs respectively. Let $\Pi$ denotes set of all execution paths of CFG $G$, the overhead of a monitoring scheme $M$ for program $P$ with CFG $G$ is:

$$M_o(G) = \sum_{\pi \in \Pi} M_o(\pi) \tag{3}$$

## 3.3 Complexity Analysis

We believe that finding the monitoring scheme 1, which minimizes the overhead cost (Equation 3) for a given CFG, requires knowledge of execution paths of the CFG. This is because depending upon what had happened on a path it may not be beneficial to switch to the optimal monitoring scheme for the rest of the path. Such an interference is not only present in an execution path but also among interacting paths. To illustrate this further consider Figure 4. In an optimal solution, the distribution of critical events on the path $c \rightsquigarrow d$ affects the decision about the monitoring mode (i.e., TT or ET) for vertices on the path $\rightsquigarrow a$ and vice-versa. It may not be correct to choose optimal strategy for the paths $\rightsquigarrow a$ and $c \rightsquigarrow d$ separately if it causes switching on edge $(a, c)$, and the cost of this switching overruns the benefit gained by choosing local optimal solutions for the two paths. This causes intra-path interference among vertices. Note that monitoring mode decision about vertices on the path $\rightsquigarrow b$ is influenced by choice of monitoring mode for virtices on the path $c \rightsquigarrow d$ which in turn gets affected by events on the path $\rightsquigarrow a$. This results into inter-path interference among intersecting paths. The presence of intra- and inter-path interference among vertices indicates that local optimization cannot guarantee overall optimal solution for a given CFG, and all execution paths should be analyzed. However, the presence of unbounded loops makes analysis of all execution paths impossible. Also, even in the absence of unbounded loops, a general CFG can have exponentially many execution paths. This makes the problem of finding the optimal solution intractable.

In order to tackle the high computational complexity of the problem and to make this technique practical, we introduce a heuristic that aims to return a monitoring scheme whose monitoring overhead is equal to or better (i.e. lower) than exclusively in ET or TT schemes. We formulate an *integer linear program* (ILP) as a heuristic for this problem. In order to make this heuristic reflect the realities of the program without computing all execution paths, we assume that function $\mathcal{F} : (u, v) \rightarrow \mathbb{N}$, $(u, v) \in A$, $u, v \in V$ is provided along with CFG of a program $P$. $\mathcal{F}(u, v)$ defines the expected number of times $P$ will execute the basic block corresponding to $v$ immediately after executing the basic block corresponding to $u$. Figure 5 illustrates a *CFG*, where the critical vertices are highlighted. The set of numerical values within parentheses defines the function, $\mathcal{F}(u, v)$. We note that this function can be evaluated using standard techniques such as program profiling and symbolic execution. The suboptimality stems from the division of the program into subpaths to estimate the monitoring cost and

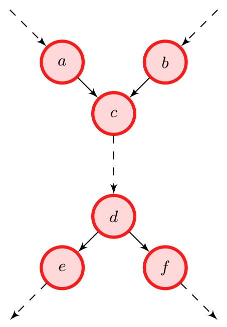


**Fig. 4.** Intra- and inter-path interference among vertices.

the use of function $\mathcal{F}$ which may not represent correct system's behaviour. Computing function $\mathcal{F}$ with high accuracy is desirable because even a small reduction in overhead will have large benefit in the long run of a monitor.

For the rest of this paper, let $CFG = \langle V, v^0, A, w, v^f, \mathcal{F} \rangle$ be a control-flow graph corresponding to a program $P$. Each vertex corresponds to a critical basic block containing one critical instruction. The definitions of $V$, $v^0$, $A$, $w$, and $v^f$ correspond to the Definition 1 (see Figure 3(b) for an example).

## 3.4 The Optimization Problem as an Integer Linear Program

The ILP problem is of the form:

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

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

**Objective Function** The objective function for our ILP model is:

$$minimize \quad (C_{event} + C_{switch} + C_{sample}) \tag{4}$$

We now describe how we map the optimization objective (Equation 4) by introducing ILP variables and computing each of three costs in terms of these variables and given elementary costs for a CFG.

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

$$x_v + y_v = 1 \tag{5}$$

**Constraint of Handling Critical Events** Equation 6 expresses the cost incurred at each critical event in $P$:

$$C_{event} = \sum_{v \in V_c} \sum_{\substack{(u,v) \in A \\ u \in V}} [\mathcal{F}(u,v) \cdot (c_{ET} \cdot x_v + c_{hist} \cdot y_v)] \tag{6}$$

where $V_c \subseteq V$ is the set of nodes that correspond to the critical basic blocks in *CFG*. The number of times that $P$ will transit from the set of nodes $u$ to $v$, where $(u,v) \in A$, determines the expected number of times that the basic block corresponding to $v$ will execute. Equations 5 and 6 guarantee that the cost incurred for the critical event in $v$ is exclusively $c_{ET}$ or $c_{TT}$ if the monitor is operating in ET or TT mode at that point in the program, respectively.

**Constraints of Switching Monitoring Mode** The following equation expresses the cost of switching between ET and TT modes:

$$C_{switch} = \sum_{\substack{(v_1,v_2) \in A \\ v_1,v_2 \in V}} [\mathcal{F}(v_1,v_2) \cdot (c_{E \to T} \cdot x_{v_1} \cdot y_{v_2} + c_{T \to E} \cdot y_{v_1} \cdot x_{v_2})] \tag{7}$$

There exists a mode switch between basic blocks $v_1$ and $v_2$ when $x_{v_1} = y_{v_2} = 1$ or $y_{v_1} = x_{v_2} = 1$. The former case implies that the monitor switches from ET mode to TT mode and the latter case implies that the monitor switches from TT mode to ET mode. Equation 7 is non-linear; to linearize this expression, we introduce the binary variables $p_{v_1,v_2}$, $q_{v_1,v_2}$, $r_{v_1,v_2}$, and $s_{v_1,v_2}$ and rewrite Equation 7 as:

$$C_{switch} = \sum_{\substack{(v_1,v_2) \in A \\ v_1,v_2 \in V}} [\mathcal{F}(v_1,v_2) \cdot (c_{E \to T} \cdot p_{v_1,v_2} + c_{T \to E} \cdot q_{v_1,v_2})] \tag{8}$$

subject to:

$$x_{v_1} + y_{v_2} + 2r_{v_1,v_2} \geq 2 \tag{9}$$

$$p_{v_1,v_2} + r_{v_1,v_2} = 1 \tag{10}$$

$$x_{v_1} + y_{v_2} - 2(1 - r_{v_1,v_2}) < 2 \tag{11}$$

$$y_{v_1} + x_{v_2} + 2s_{v_1,v_2} \geq 2 \tag{12}$$

$$q_{v_1,v_2} + s_{v_1,v_2} = 1 \tag{13}$$

$$y_{v_1} + x_{v_2} - 2(1 - s_{v_1,v_2}) < 2 \tag{14}$$

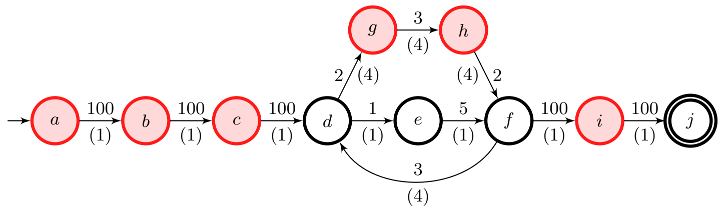


**Fig. 5.** CFG used for illustrating ILP model.

Equations 9 through 11 ensure that if $x_{v_1} = y_{v_2} = 1$, then $p_{v_1,v_2} = 1$, i.e., we incur the cost of switching from ET to TT mode. Similarly, the constraints reflected in Equations 12 through 14 ensure that if there exists a switch from TT to ET mode, then $q_{v_1,v_2} = 1$ and we incur the cost $c_{T \to E}$.

**Constraints of Sampling Cost in TT Mode** Finally, Equation 15 captures the cost incurred from the sampling the monitor does in TT mode:

$$C_{sample} = \sum_{\pi \in \Pi'(CFG)} \left(c_{TT} \cdot \mathcal{F}_\pi \cdot N_{samp_\pi}\right) \tag{15}$$

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

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

For example, if we consider the CFG shown in Figure 5, $\Pi'(CFG) = \{\langle a \to b \to c \to d\rangle, \langle d \to e \to f\rangle, \langle f \to d\rangle, \langle d \to g \to h \to f\rangle, \langle f \to i \to j\rangle\}$. Moreover, in Equation 15, $\mathcal{F}_\pi$ is the expected number of times that $\pi$ will execute at run time. $\mathcal{F}_\pi = \mathcal{F}(v_i, v_j)$, where $(v_i, v_j)$ is any arc on path $\pi$. $N_{samp_\pi}$ is the number of samples that the monitor takes when $P$ executes $\pi$ once:

$$N_{samp_\pi} = \sum_{\substack{\gamma = \langle v_i \to \ldots \to v_j\rangle, \\ \gamma \in \Gamma_\pi}} \left[ \left( \frac{W(\gamma) + c_{hist} \cdot \sum_{m=i}^{j} y_{v_m}}{SP} \right) \cdot \left( x_{v_{i-1}} \cdot x_{v_{j+1}} \cdot \prod_{l=i}^{j} y_{v_l} \right) \right] \tag{16}$$

where $W(\gamma)$ returns the sum of weights of all arcs on the path $\gamma \in \Gamma_\pi$; $v_{i-1}$ and $v_{j+1}$ denote the immediate predecessor and successor of $v_i, v_j \in V$, respectively; and $SP$ is the allowed sampling period of the monitor when it is operating in TT mode. $\mathit{\Gamma}_\pi$ is the enumerated set of paths in $\pi \in \mathit{\Pi}'(\mathit{CFG})$ of length 2 or greater. Using $\mathit{\Pi}'(\mathit{CFG})$ for the CFG shown in Figure 5, if we consider the subpath $\pi = \langle d \to g \to h \to f \rangle$, then $\mathit{\Gamma}_\pi = \{\langle d \to g \to h \to f \rangle, \langle d \to g \to h \rangle, \langle g \to h \to f \rangle, \langle d \to g \rangle, \langle g \to h \rangle, \langle h \to f \rangle\}$. Note that $|\mathit{\Gamma}_\pi| = \mathit{\Theta}\left(|\pi|^2\right)$. If $v_{i-1}$ does not exist in $\pi$, $x_{v_{i-1}} = 1$. Similarly, $x_{v_{j+1}} = 1$ if $v_{j+1}$ does not exist in $\pi$. Considering the example where $\pi = \langle d \to g \to h \to f \rangle$, if $\gamma \in \mathit{\Gamma}_\pi$ starts with $d$ or ends with $f$, then we would ignore the terms $x_{v_{i-1}}$ and $x_{v_{i+1}}$ by substituting them with the value of 1, respectively. $N_{samp_\pi}$ is linearized by the linearization technique employed for $C_{switch}$.

## 4 Implementation and Experimental Results

We empirically tested and verified our hybrid monitoring approach for a subset of programs from the SNU Real-time benchmark suite [1] on an embedded development platform with real-time guarantees. In Subsection 4.1, we describe the experimental setup and the toolchain. Then, in Subsection 4.2, we present and analyze the results of our experiments.

### 4.1 Experimental Setup

Figure 6 depicts the constructed toolchain used to generate instrumentation schemes from the model described in Section 3. The toolchain generates the program's CFG with estimated execution times of basic blocks by statically analyzing the program's source code with clang and llvm [18]. We use the tool CodeSurfer [9] to determine the location of the critical events the monitor should track at run time. The model generator takes this information along with the estimated monitoring costs to produce the corresponding model for the program. The toolchain then uses Yices [23], an SMT solver, to identify a solution (i.e., an instrumentation scheme) to the optimization problem described in Section 3. A script then takes the instrumentation scheme and instruments the program source with the necessary instructions required to monitor the program accordingly.

The monitor and programs were compiled and executed on the Keil µVision simulator that emulates the behavior of the MCB1700 development platform, which sports an ARM Cortex-M3 processor. We emphasize that the observed execution time across multiple runs of the experiment remains constant because the hardware platform provides accurate timing behavior of instructions, and in each experiment, the only tasks running were the program under inspection and the monitor. Therefore, it is safe to present the results without reporting statistical measures.

We used SNU-RT [1] benchmark suite for the performance analysis. We selected six programs from the suite with different sizes: `bs`, `fibcall`, `insertsort`, `fir`, `crc`, and `matmult`. The largest program has 250 lines of code, and the smallest has 20. We picked two sets of variables for monitoring for each program: (1) a set containing frequently changing variables and (2) a set containing rarely

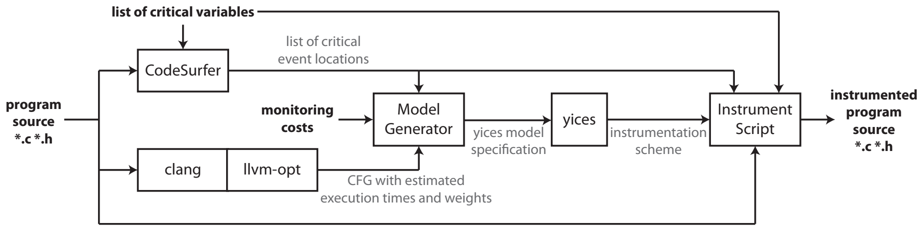

list of critical variables
list of critical event locations
CodeSurfer
program source *.c *.h
monitoring costs
Model Generator
yices model specification
yices
instrumentation scheme
Instrument Script
instrumented program source *.c *.h
clang
llvm-opt
CFG with estimated execution times and weights

**Fig. 6.** HyRV instrumentation toolchain for C applications.

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

**Table 1.** Monitor cost configurations [clock cycles].

changing variables. Instructions that potentially change the value of these variables form the set of critical instructions monitored in the experiments. For each program, the monitoring overheads were measured using the cost configurations (listed in Table 1) and associated instrumentation schemes. The cost configurations depend on the implementation of the monitor (e.g., running on the same processor, distributed). We use the configurations in Table 1 to demonstrate that the instrumentation schemes may change as a result of the relative differences in the elementary monitoring costs.

## 4.2 Experimental Results

We classify the results of our experiments based on the generated instrumentation scheme and runtime overhead:

1. The first class consists of cases, where our ILP model suggests a hybrid monitor and the monitor indeed significantly outperforms an ET or TT monitor in practice (see Figure 7).
2. The second class consists of cases where the ILP model suggests either an ET or TT monitor and the suggested solution indeed outperforms other monitoring modes (see Figure 8).
3. The third class consists of cases where the solution to the ILP model either exhibits slight improvement over other monitoring modes or it slightly underperforms in practice (see Figure 9).

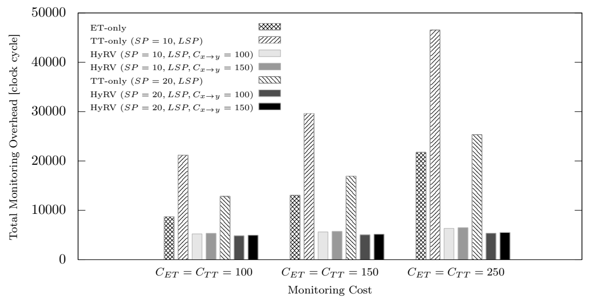


**Fig. 7.** Monitoring overhead of `crc` for three monitoring modes under all cost configurations.

In the rest of this section, we will discuss the experimental results and focus on one program from each class. We note that the three other programs not specifically discussed in this section exhibit similar results.

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

On the contrary, the large initialization function does not contain critical events; hence, a TT monitor would suffer from redundant sampling overhead. We hypothesize that the combination of these two characteristics can exploit the benefits of employing a hybrid monitor. The graph in Figure 7 validates our hypothesis. As can be seen, in all cost configurations, the hybrid monitor incurs significantly less overhead than both the ET monitor and TT monitor operating with the same sampling period. Another interesting observation is that increasing the cost of ET and TT monitor invocations does not greatly increase the overhead of the hybrid monitor. This is because the hybrid monitor only samples when the program reaches its tight loop, which reduces the cost of monitoring frequently occurring critical events by buffering them into memory before sampling. In addition, the monitoring scheme reduces the number of redundant samples by letting the monitor run in ET mode when critical events are infrequent. In such cases, the behavior of a hybrid monitor is quite robust when the cost of monitor invocation increases.

**Time-triggered Monitor with Significant Improvement** The common characteristic of the member programs of this class (i.e., `bs`, `fibcall`, `insertsort`,

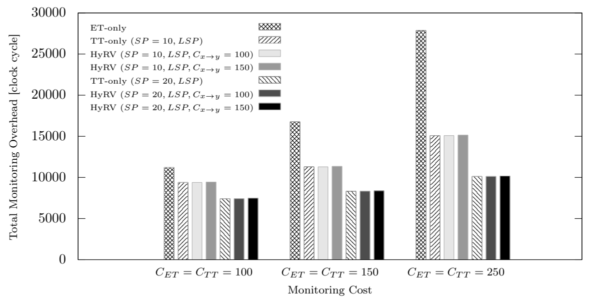


**Fig. 8.** Monitoring overhead of `insertsort` for three monitoring modes under all cost configurations.

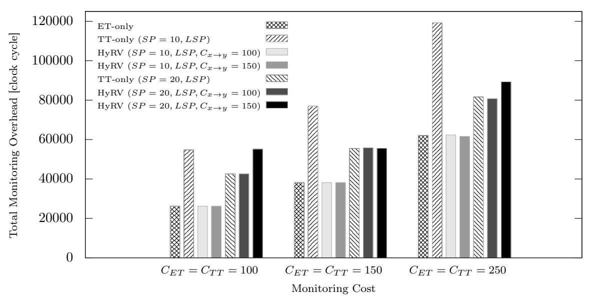


**Fig. 9.** Monitoring overhead of `fir` for three monitoring modes under all cost configurations.

and `matmult`) is that the programs have dense and evenly distributed critical instructions in their respective CFG. This makes the use of TT mode a suitable choice to monitor this class of programs. Figure 8 shows the overhead of monitoring `insertsort` with three monitoring modes (ET-only, TT-only, and hybrid) for all cost configurations. The rest of the programs in this class also exhibit similar monitoring overhead patterns. From Figure 8, one can observe that the corresponding ILP model correctly detects the even distribution of events and the solution suggests monitoring exclusively in TT mode as its solution for all cost configurations. Another observation in these experiments is that the number of redundant samples for these programs is either zero or close to zero. The low number of redundant samples again validates the choice of monitoring these programs using the time-triggered method.

**Hybrid Monitor with Mixed Behavior** The program representing this class (i.e. `fir` with CFG of the size 24 vertices and 27 arcs) does not clearly belong to the previous two classes. The number of redundant samples for this program reduces by a factor of six as the sampling period increases from $10 \times LSP$ to $20 \times LSP$. This brings the overheads of ET and TT modes to a comparable level and makes the ILP model outcome highly sensitive to the elementary monitoring costs. Figure 9 shows the monitoring overhead of `fir` under the three modes of monitoring for different cost configurations. One can observe that when the sampling period is $10 \times LSP$, the model correctly chooses ET mode for the monitoring schemes. However, if we set the sampling period to $20 \times LSP$, then the ILP model provides a hybrid solution for all three cost configurations. The proposed hybrid solutions have slightly higher overheads in comparison to ET mode, but perform as good as TT mode except for two cases in practice. The reason for this discrepancy lies in the fact that our approach is a heuristic algorithm and, hence, finds suboptimal solutions in some cases. Note, however, that this discrepancy does not dramatically affect the usefulness of our approach.

## 5 Related Work

In classic runtime verification [21], a system is composed with an external observer, called the monitor. This monitor is normally an automaton synthesized from a set of properties under which the system is scrutinized. From the logical and language point of view, runtime verification has mostly been studied in the context of Linear Temporal Logic (LTL) properties [8, 10–12, 25] and, in particular, safety properties [14, 22]. Other languages and frameworks have also been developed for facilitating specification of temporal properties [15,16,27]. [6] considered runtime verification of $\omega$-languages. In [7], the authors address runtime verification of safety-progress [4, 20] properties.

The main focus in the literature of runtime verification is on *event-triggered* monitors [17], where every change in the state of the system triggers the monitor for analysis. Alternatively, in *time-triggered* monitoring [2, 3], the monitor samples the state of the program under inspection at regular time intervals. The time-triggered approach involves solving an optimization problem that aims at minimizing the size of auxiliary memory required so that the monitor can correctly reconstruct the sequence of program state changes. Several heuristics were introduced to tackle

Finally, in [13], the authors propose a method to control the overhead of software monitoring using control theory for discrete event systems. In this work, overhead control is achieved by temporarily disabling involvement of monitor, thus avoiding the overhead to pass a user-defined threshold. Another relevant work to this line of research is [24], where the authors propose sampling using state estimation. In particular, they use hidden Markov models to estimate future reachable states for deciding whether or not the monitor must sample the program under inspection. However, the methods in [13] and [24] do not guarantee correct state reconstruction because the monitor is unaware of all program state changes that may occur between samples.

## 6 Conclusion

In this paper, we concentrated on combining two techniques in the literature of runtime verification to reduce the overhead: (1) the traditional event-triggered (ET) approach, and (2) the time-triggered (TT) method for real-time systems. We showed that one can effectively exploit the advantages of both approaches to reduce the overhead of runtime monitoring. To this end, we formulated an optimization problem that takes into account the cost of different monitoring interactions (i.e., monitor invocation in ET, sampling and building history in TT, and mode switching). In particular, the objective of the problem is to minimize the cumulative overhead in all execution paths using the aforementioned costs. Since solving the general problem can be computationally unsolvable (e.g., due to the existence of unbounded loops) or intractable, we proposed a heuristic that finds suboptimal but effective solutions to the problem by transforming it into an instance of the integer linear programming problem. Our experimental results on the SNU-RT benchmark suite showed that our technique effectively reduces the overhead as compared to selecting the ET or TT method in an ad-hoc manner.

There exist several interesting future research directions. We plan to employ symbolic execution techniques to implement a more accurate and realistic prediction function used for conditional and loop statements (see Section 3). Another open problem is to design other heuristics with lower time complexity that eliminate subpath generation. Examples include techniques that exploit static analysis such as graph density and dynamic analysis such as feedback control.

## 7 Acknowledgements

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.

## References

1. SNU Real-Time Benchmarks. `http://www.cprover.org/goto-cc/examples/snu.html`.
2. B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Sampling-based runtime verification. In *Formal Methods (FM)*, pages 88–102, 2011.
3. B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Time-triggered runtime verification. *Formal Methods in Systems Design (FMSD)*, 43(1):29–60, 2013.
4. E. Y. Chang, Z. Manna, and A. Pnueli. Characterization of Temporal Property Classes. In *Automata, Languages and Programming (ICALP)*, pages 474–486, 1992.
5. S. Colin and L. Mariani. *Run-Time Verification*, chapter 18. Springer-Verlag LNCS 3472, 2005.
6. M. d'Amorim and G. Rosu. Efficient Monitoring of omega-Languages. In *Computer Aided Verification (CAV)*, pages 364–378, 2005.

7. Y. Falcone, J.-C. Fernandez, and L. Mounier. Runtime Verification of Safety-Progress Properties. In *Runtime Verification (RV)*, pages 40–59, 2009.
8. D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Automated Software Engineering (ASE)*, pages 412–416, 2001.
9. GrammaTech Inc. CodeSurfer®. `http://www.grammatech.com/products/codesurfer/`.
10. K. Havelund and G. Rosu. Monitoring Programs Using Rewriting. In *Automated Software Engineering (ASE)*, pages 135–143, 2001.
11. K. Havelund and G. Rosu. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 342–356, 2002.
12. K. Havelund and Gr. Rosu. Monitoring Java Programs with Java PathExplorer. *Electronic Notes in Theoretical. Computer Science*, 55(2), 2001.
13. 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)*, 2011. To appear.
14. Havelund K and G. Rosu. Efficient Monitoring of Safety Sroperties. *Software Tools and Technology Transfer (STTT)*, 6(2):158–173, 2004.
15. M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky. Monitoring, Checking, and Steering of Real-Time Systems. *Electronic. Notes in Theoretical Computer Science*, 70(4), 2002.
16. M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A Run-Time Assurance Approach for Java Programs. *Formal Methods in System Design (FMSD)*, 24(2):129–155, 2004.
17. O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. In *Computer Aided Verification (CAV)*, pages 172–183, 1999.
18. C Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization: Feedback Directed and Runtime Optimization*, page 75, 2004.
19. Martin Leucker and Christian Schallhart. A Brief Account of Runtime Verification. *Journal of Logic and Algebraic Programming (JLAP)*, (78):293–303, 2009.
20. Z. Manna and A. Pnueli. A Hierarchy of Temporal Properties. In *Principles of Distributed Computing (PODC)*, pages 377–410, 1990.
21. A. Pnueli and A. Zaks. PSL Model Checking and Run-Time Verification via Testers. In *Symposium on Formal Methods (FM)*, pages 573–586, 2006.
22. G. Rosu, F. Chen, and T. Ball. Synthesizing Monitors for Safety Properties: This Time with Calls and Returns. In *Runtime Verification (RV)*, pages 51–68, 2008.
23. SRI. Yices: An SMT Solver (1.0.34). `http://yices.csl.sri.com/index.shtml`.
24. 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.
25. V. Stolz and E. Bodden. Temporal Assertions using Aspectj. *Electronic Notes in Theoretical Computer Science*, 144(4), 2006.
26. Karen Zee, Viktor Kuncak, Michael Taylor, and Martin Rinard. Runtime checking for program verification. In *Proceedings of the 7th international conference on Runtime verification*, RV'07, pages 202–213, Berlin, Heidelberg, 2007. Springer-Verlag.
27. W. Zhou, O. Sokolsky, B. T. Loo, and I. Lee. *MaC*: Distributed Monitoring and Checking. In *Runtime Verification (RV)*, pages 184–201, 2009.