Time-triggered Program Self-monitoring

Borzoo Bonakdarpour
School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo N2L 3G1, Canada
Email: borzoo@cs.uwaterloo.ca

Johnson J. Thomas
Dept. of Elec. and Comp. Eng.
University of Waterloo
200 University Avenue West
Waterloo, ON, N2L 3G1, Canada
Email: j22thoma@uwaterloo.ca

Sebastian Fischmeister
Dept. of Elec. and Comp. Eng.
University of Waterloo
200 University Avenue West
Waterloo, ON, N2L 3G1, Canada
Email: sfischme@uwaterloo.ca

***Abstract*—*Runtime monitoring* aims at analyzing the well-being of a system at run time in order to detect errors and steer the system towards a healthy behavior. Such monitoring is a complementary technique to other approaches for ensuring correctness, such as formal verification and testing. In *time-triggered* runtime monitoring, a monitor runs as a separate process in parallel with an application program under scrutiny and samples the program's state periodically to evaluate a set of properties. Applying this technique in a computing system results in obtaining bounded and predictable overhead. Gaining such characteristics for overhead is highly desirable for designing and engineering time-critical applications, such as safety-critical embedded systems. However, a time-triggered monitor requires certain synchronization features at operating system level and may suffer from various concurrency and synchronization dependencies and overheads as well as possible unreliability of synchronization primitives in a real-time setting.**

In this paper, we propose a new method, where the program under inspection is instrumented, so that it *self-samples* its state in a periodic fashion without requiring assistance from an external monitor or internal timer. We call this technique time-triggered *self-monitoring*. First, we formulate an optimization problem for minimizing the number of points in a program, where self-sampling instrumentation instructions must be inserted. We show that this problem is NP-complete. Consequently, we propose a SAT-based solution and a heuristic to cope with the exponential complexity. Our experimental results show that a time-triggered self-monitored program performs significantly better than the same program monitored by an external time-triggered monitor.

***Keywords*-Runtime verification; monitoring; time-triggered; instrumentation; predictability; self-monitoring; real-time; embedded systems**

I. Introduction

In computing systems, *correctness* refers to the assertion that a system satisfies its specification. *Verification* is a technique for checking such an assertion. *Runtime verification* [1]–[6] refers to a lightweight technique, where a *monitor* checks at run time whether or not the execution of a system under inspection satisfies a given correctness property. Runtime verification complements exhaustive verification methods, such as model checking and theorem proving, as well as incomplete solutions, such as testing and debugging. This is because exhaustive verification often requires developing a rigorous abstract model of the system and suffers from the state-explosion problem. Testing and debugging, on the other hand, provide us with under-approximated confidence about the correctness of a system, as these methods only check for the presence of defects under specific scenarios.

Most monitoring approaches in runtime verification are *event-triggered*. In these approaches, the occurrence of new events (e.g., change of value of a variable) triggers the monitor. This constant invocation of the monitor leads to *unpredictable overhead* and potentially *bursts* of monitoring intervention at run time. These defects can cause serious issues especially in real-time embedded safety/mission-critical systems. To tackle these drawbacks, in [7], [8], the authors propose *time-triggered* runtime verification and execution monitoring, where a time-triggered monitor runs in parallel with the program and samples the program state periodically to evaluate a set of properties. Employing such a monitor results in observing bounded and predictable overhead at runtime, which are critical design parameters for a designer of embedded time-sensitive systems.

Although time-triggered monitoring results in obtaining a monitor with predictable overhead and probe effects, it introduces certain complexities as well. For example, the monitor needs to run as a separate process or thread. The first drawback of such a structure is the high cost of context switching and synchronization. Moreover, synchronization data structures require the underlying operating system to provide kernel-level system call primitives and inter-process communication features. In fact, some of the widely used embedded environments (e.g., TinyOS) lack such multi-tasking features. Moreover, if the program under scrutiny is blocked (e.g., for I/O), the monitor continues trying to sample the program periodically. This will waste system resources. Furthermore, a monitor process coupled with a program creates a tight dependency between them at run time. For instance, if the monitor crashes while evaluating properties, it may never resume the program's normal operation. Finally, since the monitor cannot directly read the state of the program, it will keep taking samples from the program even if no new events have occurred between two samples.

To address the aforementioned problems, in this paper, we introduce a new concept called time-triggered runtime *self-monitoring*. Our idea is to instrument the program under scrutiny with instructions in such a way that it *self-samples* (i.e., records the program's state) itself periodically without maintaining an internal timer. In other words, the time-triggered monitor is weaved into the program. The main challenges in instrumenting the program for enabling self-monitoring are the following:

1) *(Correctness)* How should the program be instrumented such that the time interval between two successive self-sampling points does not exceed the desired sampling period? We assume that the sampling period is provided by the system designer based on the structure of the program under inspection and properties of interest (e.g., determined by the automated methods in [7]–[10]).
2) *(Instrumentation optimality)* How can we minimize the overhead of instrumentation at run time while enabling time-triggered self-monitoring that respects the correctness condition?
3) *(Minimum deviation)* How should the program be instrumented such that execution of sampling instructions are as close as possible to the given sampling period in all execution paths?

Our approach works as follows for *sequential* programs. We formally define the concept of time-triggered self-monitoring in terms of the correctness and instrumentation optimality constraints mentioned above. In order to ensure correctness, we construct the program's control-flow graph (CFG), where the weight of a vertex is its best-case execution time (BCET). Computing the sampling period of a CFG based on BCET of basic blocks is quite realistic, as (1) all hardware vendors publish the BCET of their instruction set in terms of clock cycles, and (2) BCET is a conservative approximation and no execution occurs faster than that. Using vertex weights, one can design simple algorithms that identify vertices where self-sampling instructions should be added. These instructions simply read the state of the program (e.g., variable values, contents of stacks, register values, etc) and pass the state to a monitor function in the program for evaluating properties.

To ensure optimal instrumentation, we require that the number of instrumented vertices in the control-flow graph is minimum. We show that the corresponding optimization decision procedure is NP-complete in the size of the program's control-flow graph. To remedy the exponential complexity, we follow two approaches. First, we propose a mapping from our optimization problem to the Boolean satisfiability problem. This mapping enables us to utilize powerful SAT-solvers to solve our optimization problem. Secondly, we propose a heuristic that finds nearly optimal solutions to the problem.

We emphasize that we do not address the third constraint introduced above (i.e., minimum deviation) in this paper. Also, our current approach works only for sequential programs. This is because enabling self-monitoring with optimal instrumentation requires analysis of the causal order of occurrence of events in a concurrent program for identifying optimal instrumentation points. This is outside the scope of this paper. Our method is fully implemented in a tool chain. The tool takes a C program as input and computes the instrumentation locations. We have conducted a set of experiments to compare the behavior of self-monitoring programs with their counterparts monitored by an external process. The experimental results show that self-monitored programs perform significantly faster than externally monitored programs. This is simply due to elimination of the cost of synchronization and context switching for programs monitored by an external process.

We note that instrumenting a program to add self-checking instructions is a commonly applied exercise by system designers and developers. Examples include assertion instructions, exception handling, and even simple conditional statements to check the state of the program. The technique proposed in this paper ensures that such instructions are executed within a certain time period at run time, and that the number of inserted instructions is minimum.

Organization. The rest of the paper is organized as follows. We present the preliminary concepts on time-triggered runtime verification and execution monitoring in Section II. In Section III, we formally define the notion of time-triggered self-monitoring and analyze the complexity of identifying the minimum number of instrumentation points for enabling self-monitoring. Section IV is dedicated to our SAT-based solution and the heuristic as efficient solutions to the optimization problem. The results of experiments are analyzed in Section V. Related work is discussed in Section VI. Finally, in Section VII, we make concluding remarks and discuss future work.

II. Background

Time-triggered runtime monitoring [7], [8] consist of a monitor and an application program under inspection. The monitor runs in parallel with the application program and interrupts the program execution at regular time intervals to observe the state of the program. This state could be formed by some variable values, stack values, register values, etc. The key advantage of this technique is obtaining bounded and predictable overhead incurred on the program execution. This overhead is inversely proportional to the sampling period at which the monitor samples the state of the program.

In this section, we review two existing techniques for time-triggered execution monitoring [8] and runtime verification [7] in Subsections II-A and II-B, respectively.

```
A:    if (x < 5) {
B:        x++;
          goto A
      }
      else {
C:        x -= 10;
          goto A;
      }
```

Figure 1. A simple C program.

Both methods employ the notion of *control-flow graph* (CFG) in order to reason about program execution and its timing characteristics.

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

- V is a set of *vertices*, each representing a basic block of P. Each basic block consists of a sequence of instructions in P.
- v^0 is the *initial vertex* with indegree 0, which represents the initial basic block of P.
- A is a set of *arcs* of the form (u, v), where $u, v \in V$. An arc (u, v) exists in A, if and only if the execution of basic block u can immediately lead 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.

For example, Figure 1 shows a simple C program with three basic blocks labeled A, B, and C. Figure 2(i) shows the control-flow graph of the program.

A. Time-triggered Execution Monitoring

In execution monitoring, the objective is to take periodic samples in such a way that the monitor can re-construct execution paths. To this end, the monitor has to execute at the speed of the minimum best-case execution time of branching statements. For example, in Figure 2(i) the monitor needs to execute at the speed of shortest best-case execution time of $A + B$ or $A + C$; otherwise, the re-construction of the execution path will not be possible. Figure 2(ii) shows the timing diagram for the example. It demonstrates that, assuming all basic blocks take an execution time of 1 time unit, after two time units, it will be impossible to decide whether the program took the path $A \to B \to A$ or $A \to C \to A$. Thereby, the sampling period for the program needs to be $SP = 2$.

To increase the sampling period and, hence, decrease the monitor intervention in execution of the program, we introduce *markers* to the program. A marker is a simple

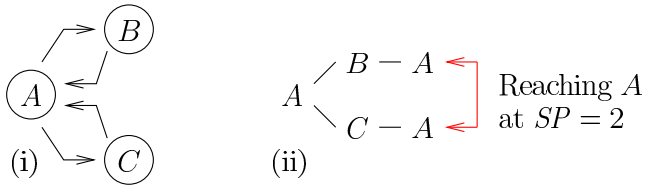

Figure 2. Example of path re-construction by applying the minimum sampling period for the program in Figure 1.

variable that can be manipulated in a basic block to distinguish different paths and, hence, obtaining in a larger sampling period. In our example, we introduce marker m_1 and instrument vertex C (see Figure 3(i)). Vertex C manipulates the value of marker m_1 by incrementing it. Thus, the monitor can re-store the basic block *id* (vertex A, B, or C), the current value of m_1, and a time stamp. The timing diagram in Figure 3(ii) shows that introducing the marker increases the sampling period to $SP = 4$, because only after four time units the program will have two or more paths with the same number of increments of m_1 and the same basic block *id*s.

B. Time-triggered Runtime Verification

Let P be a program and Π be a logical property (e.g., in LTL), where P is expected to satisfy Π. Let $\mathcal{V}_\Pi$ denote the set of variables that participate in Π. In our time-triggered runtime verification technique, the monitor reads the value of variables in $\mathcal{V}_\Pi$ in a periodic fashion in order to evaluate property Π. The main challenge in this mechanism is accurate re-construction of the state of P between two

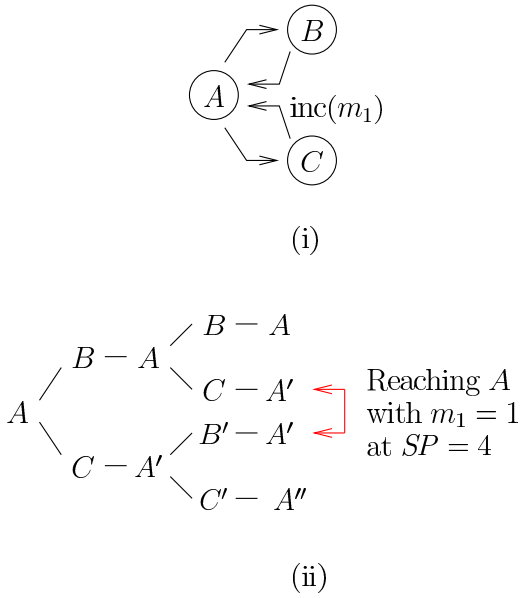

Figure 3. Example of a single instrumentation to extend SP for the program in Figure 1.

samples; i.e., if the value of a variable in $\mathcal{V}_\Pi$ changes more than once between two samples, then the monitor may fail to detect violation of Π. For instance, in the program of Figure 1, if we are to verify the safety property $\Pi \equiv \Box(-5 \leq x \leq 5)$ (read as 'it is always the case that the value of x is between -5 and 5'), then the monitor requires a fresh value of variable `x` without overlooking any changes. Thus, the sampling period for the program needs to be $SP = 2$. Notice that although there are similarities, execution monitoring and runtime verification focus on different issues: the former concentrates on execution paths and the latter on state variable changes.

To increase the sampling period, one can introduce *history variables* to the program. For example, in Figure 1, we introduce history variables `x1` and `x2` and add instrumentation instructions `x1 := x` and `x2 := x` to basic blocks B and C, respectively. Thus, if the execution of each instrumentation instruction takes 1 time unit, then we can increase the sampling period to $SP = 5$. This is due to the fact that only after six time units the value of `x1` or `x2` will be over written. Thus, sampling period $SP = 5$ allows the monitor to fully re-construct the state of the program using history variables when it takes a sample.

III. Time-triggered Self-monitoring with Minimum Instrumentation

A. Problem Description

Deploying time-triggered program execution monitoring and runtime verification as discussed in Section II requires a separate process or thread that continually interrupts the program execution to take samples and evaluate a set of properties. Although time-triggered monitors with interruptions have widely been used, they suffer from two main drawbacks:

- time-triggered interruptions introduce a large number of context switches to the system,
- if the program under inspection is blocked (e.g., waiting for I/O) the monitor keeps waking up to take samples from the program, and
- incorporating external monitors requires communication between at least two processes and synchronization data structures require the underlying operating system to provide kernel-level system call primitives and inter-process communication features.

Clearly, these issues introduce additional but unnecessary overhead to the system. In addition, in the latter case, such primitives may possibly be unreliable in real-time settings. Moreover, some embedded environments such as TinyOS do not provide such primitives at all.

In order to eliminate the aforementioned overheads, we introduce the concept of time-triggered *self-monitoring*. Our idea is to remove the external time-triggered monitor and instrument the program under inspection by augmenting the program with instructions that *self-sample* the state of the program periodically for inspection without using an internal timer. Specifically, these instructions are intended to read the state of the program (e.g., a set of variables, registers, path markers, stack contents, etc) and call a function within the program for monitoring purposes. Moreover, the program execution time between each two successive samples must be at most the desired sampling period, given as an input parameter. For instance, for the program in Figure 1, where the sampling period is $SP = 2$, the goal is to augment the program with instructions that take a sample from the value of variable `x`, such that the execution time between each two successive samples is *at most* 2 time units.

We emphasize that we assume that the sampling period is given as input to an instrumentation algorithm for self-monitoring. As illustrated in the example in Section II, the given sampling period can be the minimum sampling period (e.g., $SP = 2$) or an increased sampling period using markers (e.g., $SP = 4$) or history variables (e.g., $SP = 5$). Thus, the process of obtaining a sampling period is irrelevant to the algorithms that generate instrumentation schemes for enabling self-monitoring for a program. In other words, such algorithms only take a control-flow graph and a desired sampling period as input and return a set of vertices of the control-flow graph that need to be instrumented.

A naive solution to instrument a program for self-monitoring is as follows. One can insert self-sampling instructions at every vertex of the CFG, that take samples within the sampling period. However, in order to minimize the impact of instrumentation, it is desirable to insert the minimum number of self-sampling instructions in the program. Next, we formalize an optimization problem that captures minimum instrumentation that enables self-monitoring in a program for a given sampling period.

B. Complexity Analysis

Let $G = \langle V, v^0, A, w\rangle$ be the control-flow graph of a program and SP be the sampling period at which the program has to be sampled. Let $\Pi_{v,v'}$ denote the set of all paths between two vertices v and v' in V and Π be the set of all paths in G. The length of a path π (denoted $Length(\pi)$) is calculated as the sum of the weights of arcs present on π using the function w.

Now, let $\mathcal{V} : \Pi \rightarrow 2^V$ be the function that obtains the set of vertices on a path except the source and end vertices. Our goal is to find the minimum set of vertices $V' \subseteq V$, such that for any two vertices $v, v' \in V'$, the length of the longest path from v to v' that does not pass through a vertex $v'' \in V'$, where $v'' \neq v, v'$, is at most SP. The initial vertex v^0 is by default always present in V'. The set V' identifies the basic blocks that need to be instrumented to augment the program with self-sampling instructions. We now show that this minimization problem is NP-complete.

Instance. A control-flow graph $G = \langle V, v^0, A, w\rangle$, a sampling period SP, and a positive integer k, where $k \leq |V|$.

Self-monitoring instrumentation decision problem (SMI). Does there exist a set $V' \subseteq V$ of vertices such that:

- $|V'| \leq k$
- $v^0 \in V'$
- $\forall v, v' \in V' : \forall \pi \in \Pi_{v,v'} :$
$$\nexists v'' \in (\mathcal{V}(\pi) \cap V') : Length(\pi) \leq SP.$$

Lemma 1: SMI is in NP.

Proof: We need to show that given a solution to the problem, one can verify its correctness in polynomial-time. Given an instance of SMI and a set V' of vertices as a certificate, we verify whether or not V' solves the decision problem as follows. The first two conditions of SMI can be verified trivially. For the third condition, for every vertex $v \in V'$, we construct an SP-depth first tree (SPDFT) as follows. An SP-depth first tree of a vertex v is a spanning tree rooted at v obtained by applying depth-first search exploration on G, such that the length of any path of the tree that starts from v and ends at a leaf is at most SP. Also, all forward and cross edges are preserved when a depth first search exploration is applied on G to obtain the SPDFT of a vertex v.

Now, given a SPDFT rooted at vertex $v \in V'$, we check if there exists a path from v to one of the leaves, such that this path does not include a vertex $v' \in V'$, where $v' \neq v$. If so, it means that after self-sampling in basic block v, there exists an execution path of the program where self-sampling does not occur within the given sampling period SP. Hence, the answer to the verification question is negative. Otherwise, starting from v, in all execution paths the program self-samples within SP time units.

We repeat this procedure for all vertices in V'. If all vertices pass this verification successfully, the answer to the verification problem is affirmative. The complexity of the algorithms is $O(V^2)$ and, hence, SMI is a member of the class NP. ■

Lemma 2: SMI is NP-hard.

Proof: Now, we show that SMI is NP-hard. To this end, we reduce the *Minimum Vertex Cover Problem* (VC) to SMI. The minimum vertex cover problem is as follows [11]. Given a (directed or undirected) graph $G = \langle V, E\rangle$ and a positive integer K, the problem is to find a set $U \subseteq V$, such that $|U| \leq K$ and each edge in E is incident to at least one vertex in U.

Mapping. Let directed graph $G_{vc} = \langle V_{vc}, E_{vc}\rangle$ and a positive integer K be an instance of VC. We assume that the graph has a vertex v^0_{vc} with indegree zero. This assumption does not change the complexity of VC. We can obtain an instance of SMI as follows:

- Graph $G_{smi} = \langle V_{smi}, v^0_{smi}, A_{smi}, w\rangle$, where
  - $V_{smi} = V_{vc}$
  - $v^0_{smi} = v^0_{vc}$
  - $A_{smi} = A_{vc}$
  - $w(a) = 1$ for all $a \in A_{smi}$
- Sampling period $SP = 2$, and
- $k = K$.

Reduction. We now prove that a solution to VC exists if and only if a solution to the obtained instance of SMI as prescribed above exists:

- ($\Rightarrow$) Let $U \subseteq V$ be a solution to VC for graph $G_{vc} = \langle V_{vc}, E_{vc}\rangle$, such that $|U| \leq K$. Let V' identical to U be the solution to SMI for graph $G_{smi} = \langle V_{smi}, v^0_{smi}, A_{smi}, w\rangle$ and sampling period $SP = 2$. Thus, self-sampling instructions are added to the end of basic blocks in U. We now show that this solution is valid for SMI. First, we have $|U| \leq k$, as $k = K$ and $|U| \leq K$. Secondly, for any edge $e = (u, v) \in E$, either one of the incident vertices u or v must belong to U or both u and v belong to U. In the former case, the number of edges on a path from either u or v to another vertex $v' \in V'$ will be 2. In the latter case, the number of edges on a path from either u or v to another vertex in V' will be 1 since that would either be v or u, respectively. Likewise, the number of edges between any two vertices in V' is at most 2 by applying the same analogy to all edges in E. Hence, the set U is an answer to the instance of SMI.

- ($\Leftarrow$) Let V' be a solution to the instance of SMI (i.e., the graph $G_{smi} = \langle V_{smi}, v^0_{smi}, A_{smi}, w\rangle$ and sampling period $SP = 2$). We show that U identical to V' is a solution to VC for the graph $G_{vc} = \langle V_{vc}, E_{vc}\rangle$. First, notice that we have $|V'| \leq K$, as $K = k$ and $|V'| \leq k$. Secondly, for any two vertices $v, v' \in V'$, the number of arcs on a path from v to v' that does not include a third vertex $v'' \in V'$ must be is at most 2. Otherwise U is a not a valid solution to SMI. For the case of length 2, one of the edges will be incident to v and the other will be incident to v'. Since $v^0_{smi} \in V'$, it can be seen that all the arcs in E_{vc} will be incident to at least one of the vertices in U using the same analogy applied to all vertices in U taken as pairs. Hence, U is a vertex cover for graph G_{vc}.

■

Theorem 1: SMI is NP-complete.

Proof: The proof of the theorem trivially follows from Lemmas 1 and 2. ■

IV. Coping with the Exponential Complexity

As we showed in Section III, the problem of identifying minimum set of instrumentation for a program to enable

self-monitoring is NP-complete. To remedy the exponential complexity, in this section, we propose a SAT-based solution that finds an optimal solution to our problem and a greedy algorithm. These solutions are presented in Subsections IV-A and IV-B, respectively.

A. A SAT-based Solution

In this subsection, we propose a transformation from the optimization problem presented in Section III (SMI) into the *Boolean satisfiability problem* (SAT); i.e., the problem of assigning truth values to variables of a given Boolean formula to make the formula evaluate to logical $true$.

Let $G = \langle V, v^0, A, w\rangle$ be a control-flow graph and SP be a desired sampling period. We construct a SAT formula as follows. The set of Boolean variables in our SAT formula is:

$$X = \{x_v \mid v \in V\}.$$

Our intention is that, if $x_v = true$, then basic block v in G will be instrumented with self-sampling instructions. Otherwise, basic block v remains unchanged.

We now identify the constraints of the SAT formula. Let v be a vertex in V with outdegree greater than 0. We construct the SP-depth first tree as described in the proof of Lemma 1 (i.e., a spanning tree rooted at v obtained by applying depth first search exploration on graph G while preserving all forward and cross edges, such that the length of any path from v to its leaves is at most SP). Let $Ch : V \rightarrow 2^V$ denote a function that computes the set of child vertices of a vertex. The Boolean formula representing vertex v is of the form:

$$\mathcal{F}_v = (x_v \Rightarrow \bigwedge_{u \in Ch(v)} \mathcal{G}_u^{SP-1}) \tag{1}$$

Intuitively, $\mathcal{F}_v$ captures the constraint that if basic block v is instrumented, then in all execution branches, a basic block within $SP-1$ steps needs to be instrumented as well. The latter proposition is specified by the conjunction of $\mathcal{G}_u^{SP-1}$ formulas. For example, consider the 3-depth tree rooted at vertex v_1 in Figure 4. For this tree, by applying Constraint 1, we have $\mathcal{F}_{v_1} = (x_{v_1} \Rightarrow \mathcal{G}_{v_2}^2 \wedge \mathcal{G}_{v_3}^2)$.

Formula $\mathcal{G}$ is recursively defined as follows:

$$\mathcal{G}_u^i = (x_u \vee \bigwedge_{w \in Ch(u)} \mathcal{G}_w^{i-1}) \tag{2}$$

i.e., either basic block u is instrumented or in all execution branches starting from u, a basic block within $i-1$ steps is instrumented. The termination condition of this formula is $\mathcal{G}_w^0 = x_w$. For example, by applying Constraint 2, we have $\mathcal{G}_{v_2}^2 = x_{v_2} \vee [(x_{v_4} \vee (x_{v_7} \wedge x_{v_8})) \wedge (x_{v_5} \vee x_{v_9})]$ and $\mathcal{G}_{v_3}^2 = x_{v_3} \vee x_{v_6}$.

We add identical constraints for each vertex in the control-flow graph. Thus, our complete SAT formula is the following:

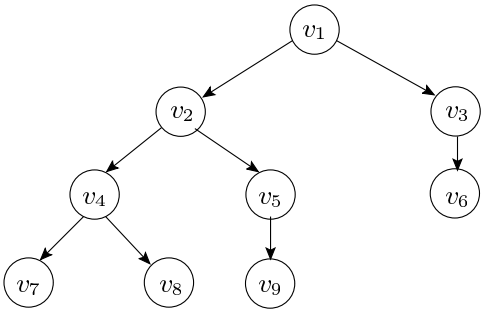

Figure 4. Example of a 3-depth first tree.

$$\mathcal{F} = \bigwedge_{v \in V} \mathcal{F}_v \tag{3}$$

Finally, our objective to minimize the number of instrumentations for self-sampling is the following (assuming that logical $true$ has integer value 1 and logical $false$ has integer value 0):

$$\text{minimize} \sum_{v \in V} x_v \tag{4}$$

i.e., by finding the minimum number of Boolean variables (respectively, vertices) whose truthfulness (respectively, instrumentation) makes the SAT formula $true$ (respectively, enables self-monitoring in the control-flow graph). We note that although Constraint 4 is not a normal SAT constraint, one can implement such a constraint in modern solvers for satisfiability modulo theory (SMT) efficiently using a simple binary search algorithm.

Special Case. A special case occurs when the the required sampling period is greater than the length of the longest execution path of the program. In such a case, both $SAT-based$ and $Greedy$ algorithms instrument only the root and the leaves of the program's CFG. This results in a possibility that there could exist an execution path with length greater than the required sampling period if the CFG has any cycles or loops. The main reason for such a case to occur is due to the fact that the loop bound is not taken into consideration in the building of the CFG of the program. This special case is solved by instrumenting at least one vertex that lies on the execution path that constitutes the loop. This is done for all the loops that exist in the program.

B. A Greedy Algorithm

In addition to our SAT-based solution, we also propose a simple greedy algorithm for instrumenting a given control-flow graph $G = \langle V, v^0, A, w\rangle$ with sampling period SP. The algorithm works as follows:

1) Initially, we let $U = \{v^0\}$ and $W = \{v^0\}$.

Case Study	Sampling Period (ns)	Size of CFG (Vertices)	*SAT-based Solution*		*Greedy Algorithm*	
			No. of Vertices	**% of Vertices**	**No. of Vertices**	**% of Vertices**
CNT	1000	28	5	17.85	7	25
InsertSort	1000	11	4	36.36	4	36.36
MATMULT	1000	29	7	24.13	9	31.03
FIBCALL	25	9	7	77.77	7	77.77
QURT	100	30	8	26.67	20	66.67
ADPCM	1000	158	29	18.35	70	44.3

Table I
COMPARING THE NUMBER AND PERCENTAGE OF VERTICES CHOSEN FOR INSTRUMENTATION FOR SAT-BASED AND GREEDY TECHNIQUES.

2) We construct SP-depth first trees (SPDFT) rooted at vertices in U and set $U = \emptyset$.
3) Let T be the set of all vertices and R be the leaves of the tree constructed in Step 2.
4) We instrument the vertices in R and we let $U = U \cup R$ and $W = W \cup T$.
5) We repeat Steps 2 and 3 until $W = V$.

The only case where this algorithm may not instrument correctly is when a control-flow graph has cycles. To deal with cycles, one can find back-arcs and instrument vertices on cycles. The special case discussed in Subsection IV-A also applies for the heuristic presented in this subsection.

V. EXPERIMENTAL RESULTS

In this section, we present the results of our experiments to evaluate the effectiveness and efficiency of time-triggered self-monitoring. In Subsection V-A, we describe our experimental settings. Then, in Subsection V-B, we analyze the results of experiments.

A. Experimental Settings

Our tool chain consists of the following. Given a C program, we first convert it to a program in assembly language. Then, we generate the control-flow graph from the assembler code. Next, we either transform the control-flow graph into a SAT formula using the method described in Subsection IV-A or feed the control-flow graph into our heuristic in Subsection IV-B. We utilize the SMT-solver Yices [12] to solve our SAT formulae. In either case, we obtain the set of vertices of the control-flow graph that need to be instrumented for self-sampling. The instrumentation instructions include storing the value of the most frequently used variable to an array to emulate a property verification task.

In order to compare our self-monitoring technique with external time-triggered monitoring, we deploy a time-triggered external monitor as follows. We use shared memory for the program and the monitor to exchange data between them. This data is basically the value of variables that change the truthfulness of a property in the program under inspection. The program is instrumented, so that it writes the value of the most frequently used variable to the shared memory whenever the variable is modified. The monitor periodically reads the shared memory and stores the values in an array, performing the same monitoring task as for the self-monitored program.

Our case studies are drawn from the Mälardalen [13] benchmark suite. All experiments in this section are conducted on a Dual-core ARM Cortex-A9 MPCore with Symmetric Multiprocessing (SMP) at 1 GHz each and 1 GB low-power DDR2 RAM under Ubuntu Linux with the default scheduling policy. Each case study is run in a loop of 1000 iterations for measurements and we ran each experiment 50 times to ensure that the collected data is sound and exhibits reliable confidence intervals.

B. Results and Analysis

1) Performance of the SAT-based and Greedy Techniques: First, we analyze the performance of our SAT-based (i.e., optimal) and greedy solutions in terms of their capability in handling input control-flow graphs. Table I shows the chosen sampling periods and the size of the input control-flow graph in terms of the number of vertices. We note that the sampling periods are chosen based on two criteria, namely, the internal structure of the case studies and the requirement that the programs should be sampled at least once. It can be observed that on average the size of the solution set obtained by using the greedy algorithm is nearly double the size of the solution set obtained by using the SAT-based method.

The time spent in obtaining the solution sets using the SAT-based approach and the greedy algorithm (not shown in Table I) are comparable in most cases. The only exception was ADPCM with sampling period 1000ns. We ran the SAT-based method on ADPCM with sampling period 1000ns and it took nearly 4 hours to obtain a solution. This special case highlights the fact that in some cases, obtaining a

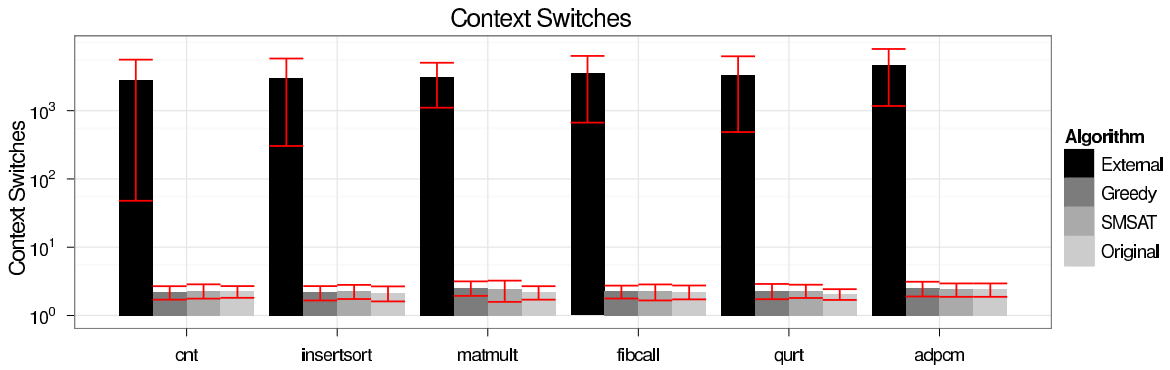

Figure 5. Results on reduction of context switching (in log scale).

near-optimal solution is more feasible than the optimal solution. For the cases where the data (number of vertices instrumented) for different sampling periods are equal, the corresponding row is omitted in Table I. Obtaining equal data for different sampling periods is due to the fact that the sampling period is greater than the execution time of the program. In this case, only the root, leaves, and at least one vertex for each loop that lies on the execution path that constitutes the loop, of the program's CFG would be instrumented. Thus, increasing the sampling period results in obtaining the same size of vertices for instrumentation.

2) Analysis of Self-monitoring Overhead: We now analyze the impact of instrumentation on performance in terms of the number of context switches and execution time of programs under inspection. Figure 5 compares the number of context switches in execution of our case studies using external monitor and self-monitored programs instrumented by the SAT-based approach and the greedy algorithm. Note that the bar chart in Figure 5 is in *logarithmic scale*. As can be seen the number of context switches incurred using external monitoring is higher than self-monitoring in orders of magnitude. This result simply shows that self-monitoring is highly preferred in a real-time setting, where non-determinism is not desirable. Also, the number of context switches incurred in the original unmonitored program an self-monitored programs instrumented by SAT-based and greedy approaches are very close. In other words, self-monitoring can significantly assist in preserving the predictability of the program under inspection. Note that in Figure 5 the error bar shows reliable confidence interval of 95%.

Figure 6 compares the total execution time of our case studies (in microseconds) for the original unmonitored program and three different monitoring techniques: (1) using an external monitor, (2) self-monitoring program instrumented by the SAT-based approach (SMSAT), and (3) self-monitoring program instrumented by our greedy algorithm. Note that the bar chart in Figure 6 is also in logarithmic scale. As can be seen in Figure 6, the total execution time of programs monitored by an external process is significantly higher than the execution of their self-monitored counterparts. More specifically, self-monitored programs instrumented using the SAT-based method are on average 2 times faster than externally monitored programs. And, self-monitored programs instrumented using the greedy approach are on average 1.6 times faster than externally monitored versions. It can also be observed that self-monitored programs instrumented using the SAT-based method run on an average 2 times slower than their unmonitored counterparts. Also, note that in Figure 6 the error bar shows reliable confidence interval of 95%.

VI. Related Work

In classic runtime verification [2], 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. To the best of our knowledge, in the literature of runtime verification, monitors are event-triggered [14] in the sense that every change in the state of the system invokes the monitor for analysis.

From the logical and language points of view, runtime verification has mostly been studied in the context of Linear Temporal Logic (LTL) [3], [6], [15]–[18] and in particular safety properties [19], [20]. Other languages and frameworks have also been developed for facilitating specification of temporal properties [21]–[23]. Runtime verification of ω-languages was considered in [24]. In [25], the authors address runtime verification of safety-progress [26], [27] properties.

In [7]–[9], the authors introduce a time-triggered execution monitoring and runtime verification techniques. They propose a framework that allows quantitative reasoning about issues involved in time-triggered techniques. They also discuss how to optimally instrument a program by a set

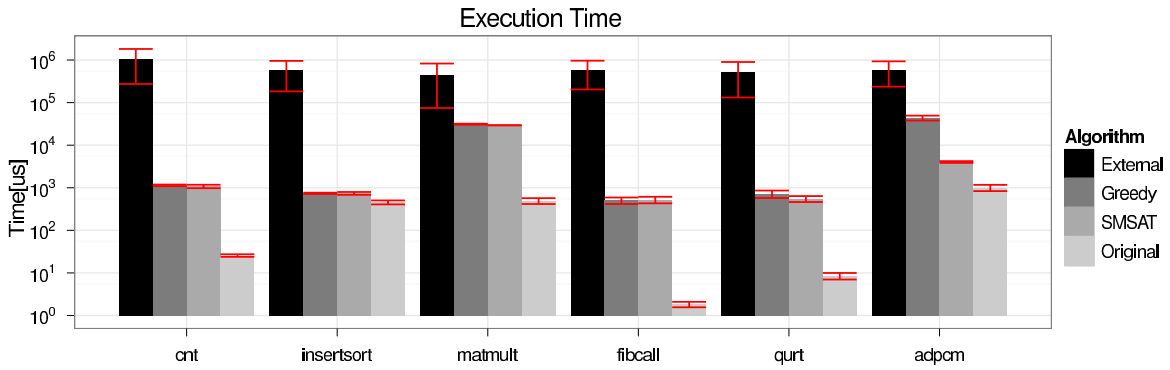

Figure 6. Results on reduction of execution time (in log scale).

of markers and history variables, such that different execution paths reachable from the same state or state changes are distinguishable. Several polynomial-time heuristics are proposed in [10] to tackle the exponential complexity of the approach in [7]. In the same context, in [28], [29], the authors propose the language Copilot for developing hard real-time monitors. The aim of this language is to develop programs where the monitor (1) does not change the functionality and schedule of the program, and (2) adds minimal overhead to the program.

Finally, in [30], 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 [31], 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.

VII. Conclusion

In this paper, we proposed a new technique for runtime monitoring called time-triggered *self-monitoring*. This technique aims at reducing the overheads incurred at time-triggered monitoring using an external monitor process. In time-triggered external monitoring, the monitor runs in parallel with the program and samples the program state periodically to evaluate a set of properties. Incorporating such an external process increases the overhead due to inter-process communication and context switching costs. Moreover, self-monitoring remedies the tight dependency between the program and monitor at run time, making it more resilient to faults and unreliability of kernel-level synchronization system calls in real-time settings. Furthermore, self-monitoring can be deployed in embedded environments, where multi-tasking features are not necessarily assumed (e.g., in TinyOS). Moreover, since time-triggered monitoring provides us with bounded and predictable overhead, it is suitable for time-sensitive platforms, where violation of timing constraints may lead to catastrophic consequences.

Our self-monitoring technique instruments a program under scrutiny with instructions in such a way that the program *self-samples* itself periodically. Moreover, self-sampling instrumentation ensures that (1) the time interval between two successive self-sampling points does not exceed the desired sampling period, and (2) minimum number of self-sampling points is introduced to the program. We showed that solving this minimization problem is NP-complete in the size of the program's control-flow graph. We subsequently proposed a SAT-based method that finds optimal solutions and a polynomial-time greedy algorithm that finds near-optimal solutions. Our experiments show that self-monitored programs perform significantly faster than their counterparts monitored by an external monitor. Moreover, the binary code size and the number of context switches occurred in self-monitored programs are substantially less than externally monitored programs.

For future work, we are considering several research directions. An important direction is self-monitoring in the context of concurrent programs. Our current method cannot handle concurrent programs, as identifying self-sampling points in the program require causal relation analysis of the program's threads and processes. Another direction is to strengthen our optimization problem such that self-sampling instructions use their time budget optimally (see the minimum deviation criterion in Section I); i.e., the instructions execute as close as possible to the intended sampling points. Developing more sophisticated heuristics to tackle the exponential complexity of our optimization problem is also an interesting research problem.

VIII. Acknowledgement

This work is partially sponsored by Canada NSERC

Discovery Grant 418396-2012, NSERC DG 357121-2008, ORF RE03-045, ORE RE-04-036, ORF-RE04-039, APCPJ 386797-09, CFI 20314 and CMC, and ISOP IS09-06-037 grants.

REFERENCES

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

[2] A. Pnueli and A. Zaks, "PSL Model Checking and Run-Time Verification via Testers," in *Symposium on Formal Methods (FM)*, 2006, pp. 573–586.

[3] A. Bauer, M. Leucker, and C. Schallhart, "Runtime Verification for LTL and TLTL," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2009, in press.

[4] ——, "Comparing LTL Semantics for Runtime Verification," *Journal of Logic and Computation*, vol. 20, no. 3, pp. 651–674, 2010.

[5] K. Havelund and A. Goldberg, "Verify your Runs," *Verified Software: Theories, Tools, Experiments (VSTTE)*, pp. 374–383, 2008.

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

[7] B. Bonakdarpour, S. Navabpour, and S. Fischmeister, "Sampling-based runtime verification," in *Formal Methods (FM)*, 2011, pp. 88–102.

[8] S. Fischmeister and Y. Ba, "Sampling-based Program Execution Monitoring," in *ACM International conference on Languages, compilers, and tools for embedded systems (LCTES)*, 2010, pp. 133–142.

[9] J. J. Thomas, S. Fischmeister, and D. Kumar, "Lowering overhead in sampling-based execution monitoring and tracing," in *Languages, compilers, and tools for embedded systems (LCTES)*, 2011, pp. 101–110.

[10] S. Navabpour, C. W. Wu, B. Bonakdarpour, and S. Fischmeister, "Efficient techniques for near-optimal instrumentation in time-triggered runtime verification," in *International Conference on Runtime Verification (RV)*, 2011, pp. 208–222.

[11] R. M. Karp, "Reducibility Among Combinatorial Problems," in *Symposium on Complexity of Computer Computations*, 1972, pp. 85–103.

[12] "Yices: An SMT Solver," http://yices.csl.sri.com.

[13] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future," in *WCET2010*, 2010, pp. 137–147.

[14] O. Kupferman and M. Y. Vardi, "Model Checking of Safety Properties," in *Computer Aided Verification (CAV)*, 1999, pp. 172–183.

[15] K. Havelund and G. Rosu, "Monitoring Programs Using Rewriting," in *Automated Software Engineering (ASE)*, 2001, pp. 135–143.

[16] ——, "Monitoring Java Programs with Java PathExplorer," *Electronic Notes in Theoretical. Computer Science*, vol. 55, no. 2, 2001.

[17] ——, "Synthesizing Monitors for Safety Properties," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2002, pp. 342–356.

[18] V. Stolz and E. Bodden, "Temporal Assertions using Aspectj," *Electronic Notes in Theoretical Computer Science*, vol. 144, no. 4, 2006.

[19] G. Rosu, F. Chen, and T. Ball, "Synthesizing Monitors for Safety Properties: This Time with Calls and Returns," in *Runtime Verification (RV)*, 2008, pp. 51–68.

[20] K. Havelund and G. Rosu, "Efficient Monitoring of Safety Properties," *Software Tools and Technology Transfer (STTT)*, vol. 6, no. 2, pp. 158–173, 2004.

[21] W. Zhou, O. Sokolsky, B. T. Loo, and I. Lee, "*MaC*: Distributed Monitoring and Checking," in *Runtime Verification (RV)*, 2009, pp. 184–201.

[22] 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*, vol. 70, no. 4, 2002.

[23] 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)*, vol. 24, no. 2, pp. 129–155, 2004.

[24] M. d'Amorim and G. Rosu, "Efficient Monitoring of omega-Languages," in *Computer Aided Verification (CAV)*, 2005, pp. 364–378.

[25] Y. Falcone, J.-C. Fernandez, and L. Mounier, "Runtime Verification of Safety-Progress Properties," in *Runtime Verification (RV)*, 2009, pp. 40–59.

[26] Z. Manna and A. Pnueli, "A Hierarchy of Temporal Properties," in *Principles of Distributed Computing (PODC)*, 1990, pp. 377–410.

[27] E. Y. Chang, Z. Manna, and A. Pnueli, "Characterization of Temporal Property Classes," in *Automata, Languages and Programming (ICALP)*, 1992, pp. 474–486.

[28] L. Pike, A. Goodloe, R. Morisset, and S. Niller, "Copilot: A Hard Real-Time Runtime Monitor," in *Runtime Verification (RV)*, 2010, 345-359.

[29] L. Pike, S. Niller, and N. Wegmann, "Runtime verification for ultra-critical systems," in *Runtime Verification (RV)*, 2011, pp. 310–324.

[30] 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)*, vol. 14, no. 3, pp. 327–347, 2012.

[31] S. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. Smolka, and E. Zadok, "Runtime verification with state estimation," in *Runtime Verification (RV)*, 2011, pp. 193–207.