Time-triggered Runtime Verification of Component-Based Multi-core Systems*

Samaneh Navabpour[1], Borzoo Bonakdarpour[2], and Sebastian Fischmeister[3]

[1] TD Bank, Canada, Email: navabs2@td.com
[2] Department of Computing and Software
McMaster University, Canada
Email: borzoo@mcmaster.ca
[3] Department of Electrical and Computer Engineering
University of Waterloo, Canada
Email: sfischeme@uwaterloo.ca

Abstract. Runtime verification is a complementary technique to exhaustive verification and traditional testing, where a monitor inspects the correctness of a system's behavior at run time with respect to a set of logical properties. In order to tackle non-uniform intervention of monitoring in embedded real-time systems, in *time-triggered* runtime verification (TTRV), the monitor runs in parallel with the system under inspection and reads the system state at a fixed frequency for property evaluation. Such a monitor ensures bounded overhead and time-predictable intervention in the system execution. In this paper, we characterize and solve the problem of augmenting a component-based system with TTRV, where different components are expected to run on different computing cores with minimum monitoring overhead at run time. In such a system, the mapping of components to the computing cores adds an additional level of complexity to the optimization problem, which is known to be NP-complete in a single component setting. We present an optimization technique that calculates (1) the mapping of components to computing cores, and (2) the monitoring frequency, such that TTRV's runtime overhead is minimized. Although dealing with runtime overhead of concurrent systems is a challenging problem due to their inherent complex nature, our experiments show that our optimization technique is robust and reduces the monitoring overhead by 34%, as compared to various near-optimal monitoring patterns of the components at run time.

1 Introduction

Embedded systems are engineering artifacts that involve computations subject to physical world constraints [12]. These systems interact with the physical world as well as execute on physical platforms. Since embedded systems are normally deployed in safety/mission-critical systems, assurance about their *correctness* is significantly vital. In addition, as embedded applications are increasingly being deployed on multi-core platforms, due to their inherent complex nature, the

* This is an extended version of the paper appeared in RV'15.

need for guaranteeing their correctness is further amplified. Consequently, it is essential to augment such systems with *runtime verification* technology, where a monitor inspects the system's correctness at run time.

The conventional monitoring approach in runtime verification has been *event-triggered*, where the occurrence of an event of interest triggers the monitor to evaluate properties. This technique leads to unpredictable monitoring overhead and potentially bursts of monitoring invocations at run time, which may cause undesirable behavior and, hence, catastrophic consequences in real-time embedded systems. To tackle this problem, the notion of *time-triggered* runtime verification (TTRV) is introduced [5], where the monitor runs in parallel with the program and reads the program state at fixed time intervals (called the *polling period*) to evaluate a set of logical properties. Although a time-triggered monitor exhibits bounded overhead and predictable monitoring invocations, the approach in [5] falls short in handling multi-core applications when several computing components execute concurrently.

In this paper, we focus on extending the notion of TTRV to the context of component-based multi-core embedded systems. The main challenge in this context is to identify the polling period of the monitors and a mapping from components and monitors to a set of computing cores, such that the cumulative monitoring overhead is minimized. To further describe this problem consider a system with four components C_1, C_2, C_3, and C_4 that are executed only once. The system runs on two interconnected and identical computing cores P_1 and P_2, where each core hosts one time-triggered monitor. Each monitor has a fixed polling period and monitors all the components running on its host computing core. Table 1 shows the results of an experiment (see Subsection 5.3 for the settings) which measures the monitoring overhead (in milliseconds) imposed by a time-triggered monitor onto a component for different polling periods.

To demonstrate the importance of the mapping of components to computing cores, we randomly map the components in two ways:

- $\{C_1, C_2, C_3\}$ runs on P_1 and $\{C_4\}$ runs on P_2, and
- $\{C_2, C_3, C_4\}$ runs on P_1 and $\{C_1\}$ runs on P_2.

In the first mapping, on P_1, the polling period of 20 cycles and on P_2, the polling period of 90 cycles achieve the optimal cumulative monitoring overhead (i.e., overall overhead on P_1 and P_2) which is 352ms. In the second mapping, on P_1, the polling period of 70 cycles and on P_2, the polling period of 20 cycles achieve the optimal overall monitoring overhead which is 337ms. Hence,

	Polling Period [CPU cycles]								
	10	20	30	40	50	60	70	80	90
C_1	81	136	140	140	145	142	140	138	137
C_2	120	70	84	91	94	92	63	74	79
C_3	120	70	88	95	99	91	61	76	77
C_4	120	83	86	93	91	80	77	78	76

Table 1. Example of monitoring overhead [ms].

the second mapping imposes 20% less monitoring overhead. This simple experiment indicates that the mapping of components to the computing cores affects the monitoring overhead and since embedded systems are usually resource constrained, it is highly desirable to find the mapping which results in the *least* monitoring overhead throughout the system run.

With this motivation, in this paper, we propose an approach for optimizing the monitoring overhead of TTRV in component-based multi-core embedded systems. That is, given a set of components, computing cores, and a set of allowed polling periods, the goal is to identify (1) the mapping of components to computing cores, and (2) the polling period of each monitor, such that the monitoring overhead of TTRV is minimized. To achieve this goal, first, we formalize the notion of monitoring overhead associated with terminating and non-terminating components. This notion incorporates different sources of runtime monitoring overhead such as monitor invocation, event buffering, and execution of instrumentation instructions. Since the optimization problem is known to be intractable even for single component uni-core systems [5], we introduce a mapping from our problem to Integer Linear Programming (ILP). In order to incorporate the runtime characteristics of each component in our ILP model, we employ symbolic execution techniques [15].

Our approach is fully implemented within a tool chain and we report the results of rigorous experiments using the SNU [1] benchmark suite. Experimental results show that on average, our approach can reduce the monitoring overhead of TTRV by 34% as compared to various near-optimal monitoring patterns of the components at run time.

Organization The rest of the paper is organized as follows. Section 2 presents the background concepts. Section 3 formally states our optimization problem, while Section 4 presents the mapping of our problem to ILP. Section 5 presents the tool chain and the experimental results. Section 6 discusses related work. Finally, in Section 7, we make our concluding remarks.

2 Preliminaries

In this section, we define our notion of software components and time-triggered runtime verification.

2.1 Software Components

There are various definitions for a software component. In this paper, we adapt the general view of [19], where a software component is a binary unit of independent production, acquisition, and deployment that interacts with other components to form a system. Our view of a software component is in terms of a *control-flow graph*.

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

– *V: is a set of* vertices*, each representing a basic block of C. Each basic block consists of a sequence of instructions in C.*
– *v^0: is the* initial vertex *with indegree 0, which represents the initial basic block of C.*
– *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.* □

Two components interact with each other using conventional methods such as shared variables, message passing, etc. For example, consider the two components in Figures 1(a) and 1(c) which interact with each other via the shared variable alert. Component1 reads a temperature sensor every 5 time units and when the temperature exceeds 100 Celsius, it will set alert to 1. Component2 reads a pressure sensor every 10 time units and when the pressure exceeds 20 Pascals while alert is equal to 1, it will start checking the pressure every 3 time units. In this example, we assume that the BCET of each instruction in both components is 1 time unit. As a result, Figures 1(b) and 1(d) show the CFG of Component1 and Component2, respectively. Each vertex is annotated with the corresponding line numbers in the code. Note that since we focus on BCET of instructions, we consider the least delay (3 time units) for the wait instruction (line 5) in Component2.

2.2 Time-triggered Runtime Verification (TTRV)

The main challenge in implementing TTRV is to compute the *longest* polling period (LPP), such that the monitor polls (i.e., observes) all the changes in the value of the *variables of interest*. The following recaps the procedure to calculate LPP [5].

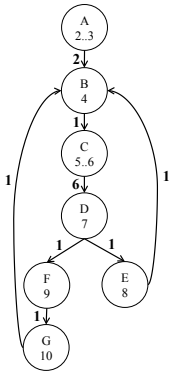
Let C be a component and Π be a logical property, where C is expected to satisfy Π. Let $\mathcal{V}_\Pi$ be the set of variables whose values can change the valuation of Π (i.e., variables of interest) and CFG_C be the control-flow graph of C. We use control-flow analysis to estimate the time intervals between consecutive state changes of C with respect to the variables in $\mathcal{V}_\Pi$. In order to calculate LPP, we modify CFG_C in two steps.

Step 1 (Identifying critical vertices) We modify CFG_C, such that each *critical instruction* (i.e., an instruction that updates the value of a variable in $\mathcal{V}_\Pi$) resides in one and only one vertex by itself. We refer to such a vertex as a *critical vertex*. For example, if $\mathcal{V}_\Pi = \{\mathsf{pressure}, \mathsf{delay}\}$ in Component2, then the critical instructions are at lines 2, 6, 8, and 10. Figure 2(a) shows the transformed CFG of Component2. We call this graph a *critical CFG*.

Step 2 (Calculating LPP) The polling period of the monitor must be such that the monitor does not overlook any state changes that could occur in C at run time. Such a polling period is defined as follows.

```
1. __task void component1 (void) {
2.     int delay = 5;
3.     os_tsk_prio (2);
4.     while (1) {
5.       os_dly_wait(delay);
6.*      temperature = read_temp();
7.       if(temperature > 100){
8.            alert = 1;}
9.       else {
10.           alert = 0;}
11.   }}
```

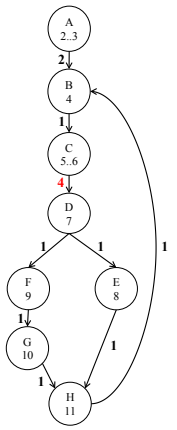
(a) Component1



(b) CFG of Component1

```
1. __task void component2 (void) {
2.*    int delay = 10;
3.     os_tsk_prio (2);
4.     while (1) {
5.       os_dly_wait(delay);
6.*      pressure = read_pressure();
7.       if(alert == 1 && pressure > 20){
8.*           delay = 3;
9.       else {
10.*          delay = 10;}
11.      update_report();
12.    }}
```

(c) Component2



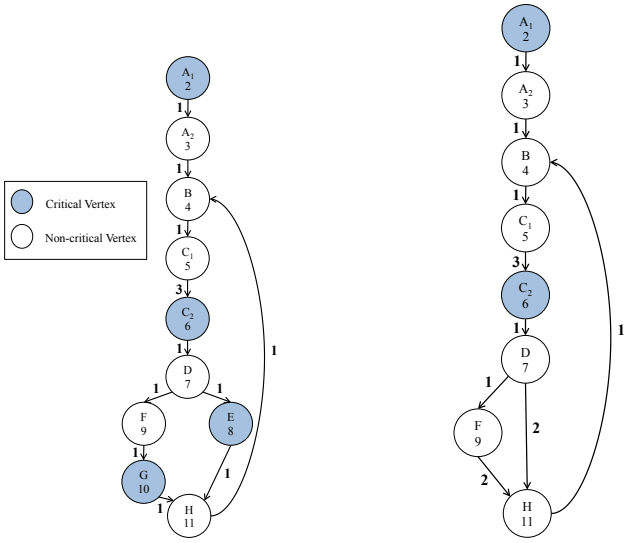
(d) CFG of Component2

Fig. 1. Component1 and Component2, along with their CFGs.

Definition 2. *Let* $CCFG = \langle V, v^0, A, w\rangle$ *be a critical control-flow graph and* $V_c \subseteq V$ *be the set of critical vertices. The* longest polling period *(LPP) for CCFG is the minimum-length shortest path between two vertices in* V_c. □

For example, *LPP* of Component2 is 2 time units.

To reduce the overhead of time-triggered monitoring by increasing *LPP*, in [5], the authors propose employing *auxiliary memory* to build a *history* of state changes between consecutive monitoring polls. More specifically, let v be a critical vertex in a critical CFG, where the critical instruction I updates the value of a variable x. The following graph transformation results in a new critical CFG with a larger *LPP*: it (1) removes v, (2) merges the incoming and outgoing arcs of v, and (3) adds an instruction $I' : x' \leftarrow x$ after instruction I in the component's source code, where x' is an auxiliary memory location. For example, the graph in Figure 2(b) is the result of applying this transformation to vertices E and G



(a) Critical CFG of Component2.

(b) Transformed Critical CFG of Component2.

Fig. 2. Critical CFG.

of Component2 (see Figure 2(a)), where vertices E and G are removed, and two new arcs (D, H) and (F, H) with weights of $1+1=2$ are added. The new graph has $LPP = 6$ due to the length of the path between A_1 and C_2.

3 Optimal Monitoring of Component-based Systems

As discussed in the introduction, the mapping of system components to the available computing cores can significantly affect the monitoring overhead imposed onto the system. Thus, we aim at finding the mapping that results in the *minimum* monitoring overhead. In this section, we present our underlying objective, the optimization problem, and its complexity analysis.

3.1 System Architecture

We follow an abstract view towards the underlying architecture of the system which is independent of the hardware, operating system, network protocol, etc. That is, a component-based system runs on a set of interconnected and potentially heterogeneous computing cores. However, we make the following assumptions:

- A component is either nonterminating or is invoked infinitely often throughout the system run.
- Given a set $\mathcal{C} = \{C_1 \cdots C_n\}$ of components and a set $\mathcal{P} = \{P_1 \cdots P_m\}$ of computing cores, where $m \leq n$, a function $\mathcal{F} : \mathcal{P} \rightarrow 2^{\mathcal{C}}$ maps each core in $\mathcal{P}$

to a unique subset of components of $\mathcal{C}$. Moreover, for any two distinct cores $P_1, P_2 \in \mathcal{P}$, we have $\mathcal{F}(P_1) \cap \mathcal{F}(P_2) = \{\}$. This function remains unchanged throughout the system execution.
- We assume a fully preemptive scheduler.
- The components can be invoked *aperiodically*.
- When a timing property involving a set of components requires verification, this set of components must run on the same core, so the property is soundly verified.
- We assume time-triggered monitors $\mathcal{M} = \{M_1 \cdots M_m\}$, where monitor M_i is deployed on core P_i, for all $i \in \{1 \cdots m\}$. Each monitor observes and verifies *all* the components in $\mathcal{F}(P_i)$.
- No two components share a *variable of interest*; i.e., shared variables cannot be monitored.

3.2 Underlying Objective

The overhead imposed by the monitor is tightly coupled with the *execution path* of a component at run time.

Definition 3. *An* execution path *of a component C with $CFG_C = \langle V, v^0, A, w\rangle$ is a sequence of the form $\gamma = \langle (v_0, \omega_0, v_1), (v_1, \omega_1, v_2), \ldots\rangle$, where:*
- *$v_0 = v^0$.*
- *For all $i \geq 0$, $v_i \in V$.*
- *For all (v_i, ω_i, v_{i+1}), where $i \geq 0$, there exists an arc (v_i, v_{i+1}) in A.*
- *For all $i \geq 0$, $\omega_i = w(v_i)$.*
- *If C is a terminating component, then $\gamma = \langle (v_0, \omega_0, v_1),\ \ldots, (v_{n-1}, \omega_{n-1}, v_n)\rangle$ is finite and v_n is a vertex in V with outdegree of zero.* □

When it is clear from the context, we abbreviate an execution path $\gamma = \langle (v_0, \omega_0, v_1), (v_1, \omega_1, v_2), ...\rangle$ by the sequence of its vertices $\gamma = \langle v_0, v_1, v_2, \ldots\rangle$. Moreover, for an infinite execution path γ, we represent the finite sub-execution path $\langle v_0, v_1, v_2, \ldots, v_n\rangle$ with γ^n, where $n \geq 0$.

For a finite path γ^n, monitored with a polling period δ, we identify the following types of *time-related* overheads:
- $\mathcal{O}_c^\delta(\gamma^n)$ denotes the cumulative time spent for invoking the monitor throughout γ^n.
- $\mathcal{O}_r^\delta(\gamma^n)$ denotes the cumulative time spent for executing the monitoring code throughout γ^n (i.e., reading variables of interest and auxiliary memory).
- $\mathcal{O}_i^\delta(\gamma^n)$ denotes the cumulative time spent for executing the instrumentation instructions in γ^n. Recall that (from Subsection 2.2) instrumentation is used to increase *LPP* of a component.

Hence, the *time-related* monitoring overhead for γ^n is

$$\mathcal{O}_T^\delta(\gamma^n) = \mathcal{O}_c^\delta(\gamma^n) + \mathcal{O}_r^\delta(\gamma^n) + \mathcal{O}_i^\delta(\gamma^n)$$

Moreover, we identify the *memory-related* overhead, denoted by $\mathcal{O}_M^\delta(\gamma^n)$, which represents the auxiliary memory required to increase *LPP* (see Subsection 2.2).

To this end, we consider both the time-related and memory-related overheads to represent the overhead associated with time-triggered monitoring. Thus, we present *monitoring overhead* as the pair $\mathcal{O}^{\delta}(\gamma^n) = \langle \mathcal{O}^{\delta}_T(\gamma^n), \mathcal{O}^{\delta}_M(\gamma^n) \rangle$.

Observe that the polling period δ of a monitor considerably affects $\mathcal{O}^{\delta}(\gamma^n)$. That is, increasing δ results in decreasing the monitor invocations (i.e., $\mathcal{O}^{\delta}_c(\gamma^n)$), and increasing instrumentation instructions and the memory consumption (i.e., $\mathcal{O}^{\delta}_i(\gamma^n)$ and $\mathcal{O}^{\delta}_M(\gamma^n)$). For instance, in Component2, consider execution path $\gamma_1 = \langle A, (B, C, D, E, H)^{\omega} \rangle$, where ω denotes the infinite execution of a sequence of basic blocks. Assuming that an instrumentation instruction takes 2 CPU cycles, for $\delta = 2$, where vertices E and G are instrumented, we have $\mathcal{O}^{\delta}_i(\gamma^n_1) = \frac{2n}{5}$. Note that for each polling period δ, there is one and only one set of associated overheads and, hence, a unique $\mathcal{O}^{\delta}(\gamma^n)$.

Clearly, for a finite path γ^n and two polling periods δ_1 and δ_2, the time-related overhead incurred by δ_1 is better than the time-related overhead incurred by δ_2 iff $\mathcal{O}^{\delta_1}_T(\gamma^n) < \mathcal{O}^{\delta_2}_T(\gamma^n)$. Accordingly, for an infinite path γ, the overhead incurred by δ_1 is better than the overhead incurred by δ_2 iff

$$\lim_{n \to \infty} \frac{\mathcal{O}^{\delta_1}_T(\gamma^n)}{\mathcal{O}^{\delta_2}_T(\gamma^n)} < 1 \tag{1}$$

For instance, for execution path γ_1 and $\delta_1 = 2$, the monitor is invoked $\frac{8n}{5 \times 2}$ times where 8 is the BCET of $\langle B, C, D, E, H \rangle$. Assuming that the monitoring code and monitor invocation each takes 5 CPU cycles, $\mathcal{O}^{\delta_1}_c(\gamma^n_1) = \mathcal{O}^{\delta_1}_r(\gamma^n_1) = 5 \times \frac{8n}{5 \times 2}$. For $\delta_2 = 6$, where once again vertices E and G are instrumented, $\mathcal{O}^{\delta_2}_c(\gamma^n_1) = \mathcal{O}^{\delta_2}_r(\gamma^n_1) = 5 \times \frac{8n}{5 \times 6}$, and $\mathcal{O}^{\delta_2}_i(\gamma^n_1) = \frac{2n}{5}$. As a result, δ_2 imposes less time-related overhead since:

$$\lim_{n \to \infty} \frac{\mathcal{O}^{\delta_2}_T(\gamma^n_1) = 2(5 \times \frac{8n}{5 \times 6}) + \frac{2n}{5}}{\mathcal{O}^{\delta_1}_T(\gamma^n_1) = 2(5 \times \frac{8n}{5 \times 2}) + \frac{2n}{5}} < 1$$

Our goal is to minimize the monitoring overhead associated with a component. Thus, for a finite set of possible polling periods PP, and a component C with set of execution paths Γ, we want to find the polling period $\Delta \in PP$ such that[4]:

$$\forall \delta \in PP : \lim_{n \to \infty} \frac{\mathcal{O}^{\Delta}_T(C) = \sum_{\gamma \in \Gamma} \mathcal{O}^{\Delta}_T(\gamma^n)}{\mathcal{O}^{\delta}_T(C) = \sum_{\gamma \in \Gamma} \mathcal{O}^{\delta}_T(\gamma^n)} \leq 1 \tag{2}$$

As discussed, the internal structure of a component determines the polling period Δ. Moreover, the features of the computing cores define the possible polling periods in PP which in practice is always finite.

In the general case, where a monitor M inspects a set of components $\mathcal{C}$, our underlying objective is to minimize the time-related overhead over all compo-

[4] Section 4.1 describes how to deal with finite paths by artificially making them infinite.

nents. In other words, our objective is to identify Δ such that:

$$\forall \delta \in PP : \lim_{n \to \infty} \frac{\mathcal{O}_T^{\Delta}(M) = \sum_{C \in \mathcal{C}} \mathcal{O}_T^{\Delta}(C)}{\mathcal{O}_T^{\delta}(M) = \sum_{C \in \mathcal{C}} \mathcal{O}_T^{\delta}(C)} \leq 1 \tag{3}$$

One can develop the corresponding equations identical to Equations 1 – 3 for the memory-related overhead, and, hence, generalize these equations for the monitoring overhead.

3.3 Optimization Problem

In a system with multiple components that run on multiple computing cores, to minimize the *overall* monitoring overhead, in addition to calculating Δ (see Equation 3), one has to also identify an efficient mapping from components to cores. Thus, our optimization problem is as follows:

Problem statement. Given a set of components $\mathcal{C}$, a set of computing cores $\mathcal{P}$, where $|\mathcal{C}| \geq |\mathcal{P}|$, and a set of polling periods PP, identify function $\mathcal{F} : \mathcal{P} \rightarrow 2^{\mathcal{C}}$ and polling period Δ_P for each $\mathcal{F}(P)$, where $P \in \mathcal{P}$, such that

- the following is minimized:

$$\sum \{\mathcal{O}^{\Delta_P}(M_P) \mid P \in \mathcal{P}\}$$

- $\Delta_P \in PP$ (i.e., the polling period of monitor M_P for components in $\mathcal{F}(P)$) satisfies Equation 3 with respect to the monitoring overhead,
- for any two computing cores $P_1, P_2 \in \mathcal{P}$, we have $\mathcal{F}(P_1) \cap \mathcal{F}(P_2) = \{\}$.

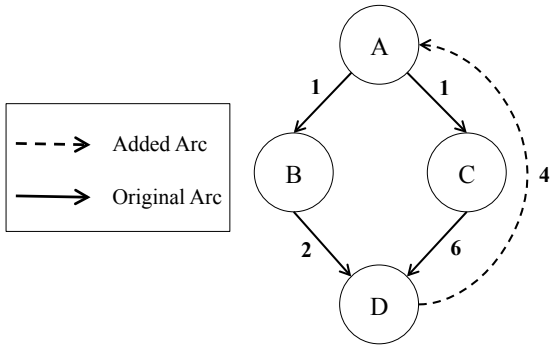
In [5], the authors show that optimizing the memory-related overhead and the polling period even for one component is NP-complete. Thus, the NP-hardness of our optimization problem immediately follows. To tackle this obstacle, we propose a mapping from our optimization problem to integer linear programming (ILP).

4 Mapping to Integer Linear Programming

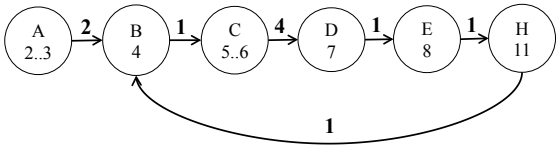
In this section, we introduce our mapping from the optimization problem in Subsection 3.3 to integer linear programming (ILP). Subsection 4.1 describes our technique to estimate the monitoring overhead associated with each component. Then, Subsection 4.2 presents the mapping to ILP.

4.1 Calculating Overhead

To solve the optimization problem, we first calculate the monitoring overhead associated with each component for every possible polling period. Recall that



(a) Creating a nonterminating component.



(b) Sub-CFG of Component2.

Fig. 3. Calculating monitoring overhead.

we focus on components that are non-terminating or are invoked infinitely often. Hence, for our analysis, we first convert all terminating components into non-terminating ones. To this end, for each component C and vertex v^t in the CFG of C, where v^t has no outgoing arcs, we add an arc $\langle v^t, w_{v^t}, v^0 \rangle$ to the CFG, where w_{v^t} is the BCET of v^t, and adjust C's source code accordingly. We refer to the new component as *adjusted component*[5]. For instance, Figure 3(a) shows the CFG of a terminating component with $v^0 = A$ and $v^t = D$. To adjust this component, we add arc $(D, w_D = 4, A)$ to the CFG.

We now describe our approach for characterizing the monitoring overhead associated with a non-terminating component. Recall that to calculate the monitoring overhead, we require the set of all execution paths of the component Γ (see Equation 2). To properly calculate the monitoring overhead, Γ must satisfy the following requirement:

CFG Coverage The execution paths in Γ must completely cover the CFG of the component. In other words, let CFG_γ be the CFG realized by a path $\gamma \in \Gamma$. Then,

$$\bigcup_{\gamma \in \Gamma} CFG_\gamma = CFG$$

Moreover, to calculate the monitoring overhead for each execution path $\gamma \in \Gamma$, we take the following steps. Consider path $\gamma = \langle A, (B, C, D, E, H)^\omega \rangle$ of Compo-

[5] The adjusted component is only used for pre-run analysis and does not replace the original component at run time.

nent2. To calculate its monitoring overhead, we extract a sub-execution path γ^n, such that the following requirements are met:

Path Coverage The length of γ^n should be long enough to cover the vertices in γ. This requirement is formalized as follows:

1. Let V_γ be the set of vertices that appear in γ. We require that for every vertex $v \in V_\gamma$, we have $v \in V_{\gamma^n}$. For instance, for any $n \geq 6$, $V_\gamma = V_{\gamma^n} = \{A, B, C, D, E, H\}$.
2. Let CFG_γ be the CFG realized by execution path γ. We require that $CFG_\gamma = CFG_{\gamma^n}$. For instance, the set of vertices in γ and γ^{1000} induce the same CFG in Figure 3(b). Observe that for path $\gamma^5 = \langle A, B, C, D, E, H \rangle$, $CFG_\gamma \neq CFG_{\gamma^5}$, since CFG_{γ^5} does not include arc (H, B).

Frequency Coverage The length of γ^n should be long enough such that the vertices which reside in an infinite loop are distinguishable from vertices that do not. This requirement is formalized as follows:

1. Let $f_\gamma(v)$ denote the number of times that vertex v appears in execution path γ. We require that for any two vertices v_1 and v_2 in V_γ, we have:

$$(\frac{f_\gamma(v_1)}{f_\gamma(v_2)} = \infty) \Rightarrow (\frac{f_{\gamma^n}(v_1)}{f_{\gamma^n}(v_2)} \approx n)$$

   For instance, for vertices B and A in γ, if $n = 1000$, then

$$\frac{f_{\gamma^{1000}}(B)}{f_\gamma^{1000}(A)} = 994 \approx 1000$$

2. We also require that for two vertices v_1 and v_2 in V_γ, we have:

$$(\frac{f_\gamma(v_1)}{f_\gamma(v_2)} = k) \Rightarrow (\frac{f_{\gamma^n}(v_1)}{f_{\gamma^n}(v_2)} = k)$$

   where k is a constant. For example, in path $\gamma' = \langle (A, B, C, B, C)^\omega \rangle$, for vertices A and B, $k = 2$. Thus, n has to be a multiple of 5. This coverage ensure that each vertex contributes the same ratio of execution time in γ^n as it does in γ.

By converting infinite execution paths to finite paths, the execution time of the paths may differ from one another. Hence, the monitoring overhead calculated for each path is based on a different scale (i.e., execution range). For example, consider paths γ_1^n and γ_2^n with BCET of 100 and 80 CPU cycles, and time-related overhead $\mathcal{O}_T^\delta$ of 50 CPU cycles. Evidently, in the long run, $\mathcal{O}_T^\delta(\gamma_2^n)$ will be larger than $\mathcal{O}_T^\delta(\gamma_1^n)$. Thus, to create comparable monitoring overheads, for each path γ^n, we *normalize* the monitoring overhead onto a scale of $[0, 1]$ as follows:

$$\mathcal{O}_\eta^\Delta(\gamma^n) = \frac{\mathcal{O}^\Delta(\gamma^n) - \mathcal{O}_{\min}(\gamma^n)}{\mathcal{O}_{\max}(\gamma^n) - \mathcal{O}_{\min}(\gamma^n)} \tag{4}$$

where $\mathcal{O}_{\min}$ is the least time-related overhead and $\mathcal{O}_{\max}$ is the largest time-related overhead that can be associated with γ^n. We consider $\mathcal{O}_{\min}(\gamma^n) = 0$

and $\mathcal{O}_{\max}(\gamma^n)$ = BCET of γ^n. We refer to $\mathcal{O}_{\eta}^{\Delta}$ as the *normalized monitoring overhead*.

Algorithm 1 calculates the normalized monitoring overhead $\mathcal{O}_{\eta}^{\delta}(C)$ associated with a component C as follows:

- For each polling period δ, the algorithm optimally instruments C with respect to δ (Line 2), in the same fashion described in Subsection 2.2.
- Then, the algorithm extracts the set of finite execution paths Γ_{γ^n} of the instrumented component C' (Line 3).
- In Lines 7 – 10, for each path $\gamma^n \in \Gamma_{\gamma^n}$, the algorithm adds the overhead of each instrumentation instruction to $\mathcal{O}_{i}^{\delta}(\gamma^n)$. Function Ω_{inst} returns the BCET (in CPU cycles) of running the instrumentation.
- Then, Algorithm 1 calculates the memory-related overhead using function $get\_mem(\gamma^n, \delta)$ (Line 12). Note that at each poll, the monitor flushes out the auxiliary memory. Hence, *get_mem* creates a window of size δ and slides it through γ^n to find the maximum amount of data (in bytes) that a set of instrumentation residing in the window can store in the auxiliary memory, and considers it as the memory-related overhead.
- Next, the algorithm sets $\mathcal{O}_{\max}(\gamma^n)$ to the BCET of γ^n (Line 15).
- In Line 16, it calculates the normalized monitoring overhead $\mathcal{O}_{\eta}^{\delta}(\gamma^n)$ of γ^n and at line 17, it adds the normalized overhead to the overall monitoring overhead of the component $\mathcal{O}_{\sum}^{\delta}(C)$.
- Finally, the algorithm calculates the normalized monitoring overhead of the component $\mathcal{O}_{\eta}^{\delta}(C)$ by setting it to the average normalized monitoring overhead of the paths in Γ_{γ^n} (Line 19). Note that, since we do not know the exact frequency at which each path will be executed at run time, we consider the average normalized monitoring overhead among the paths. In the case where the probability distribution of the execution paths is known, we employ the *weighted* average.

For a path γ, the monitor invocation overhead $\mathcal{O}_{c}^{\delta}(\gamma)$ depends on the number of times the monitor is invoked throughout the execution of γ which is tightly coupled with the execution time of γ. Since statically computing the execution time of a path is unrealistic, it is equally impractical to estimate the number of monitor invocations. Thus, instead of calculating $\mathcal{O}_{c}^{\delta}(\gamma)$, we indirectly represent $\mathcal{O}_{c}^{\delta}(\gamma)$ using the value of the polling period. It is straightforward to see that for a path γ, a larger polling period results in less monitoring invocations. Hence, for a component C and polling periods δ_1 and δ_2, we have

$$(\frac{\delta_1}{\delta_2} = k) \Rightarrow (\frac{\mathcal{O}_{c}^{\delta_1}(C)}{\mathcal{O}_{c}^{\delta_2}(C)} = k)$$

where k is a constant. Thus, to minimize $\mathcal{O}_{c}^{\delta}(C)$, one should increase the polling period δ. The overhead of running the monitoring code $\mathcal{O}_{r}^{\delta}(C)$ suffers from the same issue. We circumvent the problem in a similar fashion.

Algorithm 1 Calculating Normalized Monitoring Overhead

Input: PP: set of polling periods, C: a component
Output: $\Theta^C_{\mathcal{O}} = \{\mathcal{O}^{\delta_1}_{\eta}(C), \ldots, \mathcal{O}^{\delta_k}_{\eta}(C)\}$: set of monitoring overheads

1: **for** $\delta \in PP$ **do**
2: $C' \leftarrow opt\_instrument(C, \delta)$ /*instrument component*/
3: $\Gamma_{\gamma^n} \leftarrow get\_paths(C')$ /*get all finite execution paths*/
4: **for each** path $\gamma^n \in \Gamma_{\gamma^n}$ **do**
5: $\mathcal{O}^{\delta}_{i}(\gamma^n) \leftarrow 0$; $\mathcal{O}^{\delta}_{m}(\gamma^n) \leftarrow 0$; $\mathcal{O}^{\delta}_{\sum}(C) \leftarrow 0$
6: **for** vertex $v \in V_{\gamma^n}$ **do**
7: **if** v is instrumented **then**
8: /*adding instrumentation overhead*/
9: $\mathcal{O}^{\delta}_{i}(\gamma^n) \leftarrow \mathcal{O}^{\delta}_{i}(\gamma^n) + \Omega_{inst}(v)$
10: **end if**
11: **end for**
12: $\mathcal{O}^{\delta}_{M}(\gamma^n) \leftarrow get\_mem(\gamma^n, \delta)$
13: $\mathcal{O}^{\delta}_{T}(\gamma^n) \leftarrow \mathcal{O}^{\delta}_{i}(\gamma^n)$
14: $\mathcal{O}^{\delta}(\gamma^n) \leftarrow \langle \mathcal{O}^{\delta}_{T}(\gamma^n), \mathcal{O}^{\delta}_{M}(\gamma^n) \rangle$
15: $\mathcal{O}_{\max}(\gamma^n) \leftarrow$ BCET of γ^n
16: $\mathcal{O}^{\delta}_{\eta}(\gamma^n) \leftarrow \frac{\mathcal{O}^{\delta}(\gamma^n)}{\mathcal{O}_{\max}(\gamma^n)}$
17: $\mathcal{O}^{\delta}_{\sum}(C) \leftarrow \mathcal{O}^{\delta}_{\sum}(C) + \mathcal{O}^{\delta}_{\eta}(\gamma^n)$
18: **end for**
19: $\mathcal{O}^{\delta}_{\eta}(C) \leftarrow \frac{\mathcal{O}^{\delta}_{\sum}(C)}{|\mathcal{S}_{\gamma^n}|}$
20: $\Theta^C_{\mathcal{O}} \leftarrow \Theta^C_{\mathcal{O}} \cup \mathcal{O}^{\delta}_{\eta}(C)$ /*storing overhead for δ*/
21: **end for**
22: **return** $\Theta^C_{\mathcal{O}}$

4.2 ILP Model

In order to cope with the exponential complexity of the optimization problem described in Subsection 3.3, we transform it into *Integer Linear Programming* (ILP). ILP is a well-studied optimization problem and there exist numerous efficient ILP solvers. The problem is of the form:

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

where A (a rational $m \times n$ matrix), c (a rational n-vector), and b (a rational m-vector) are given, and, z is an n-vector of integers to be determined. In other words, the objective is to find the minimum of a linear function over a feasible set defined by a finite number of linear constraints.

Our mapping takes as input the set $\mathcal{C}$ of components, the set $\mathcal{M}$ of monitors, the set PP of polling periods, and the list of normalized monitoring overheads $\Theta^c_{\mathcal{O}}$ calculated by Algorithm 1 for each component $c \in \mathcal{C}$.

Variables. Our ILP model employs the following sets of variables:

1. $\mathbf{p} = \{p_m \mid m \in \mathcal{M}\}$, where each integer variable p_m has a value in PP, representing the polling period of monitor m. This set of variables targets to find the optimal polling periods Δ, as stated in the formal problem statement in Subsection 3.3. We emphasize that the solution to our ILP model gives the optimal polling period Δ for each monitor, if overhead calculations are accurate. Section 5 discusses how we leverage symbolic execution for overhead calculation.
2. $\mathbf{x} = \{x_m^c \mid m \in \mathcal{M} \wedge c \in \mathcal{C}\}$, where each variable x_m^c represents the normalized monitoring overhead imposed on component c by monitor m. If m does not monitor c, then $x_m^c = 0$.
3. $\mathbf{y} = \{y_m^c, {y'}_m^c \mid m \in \mathcal{M} \wedge c \in \mathcal{C}\}$, where y_m^c and ${y'}_m^c$ are called *choice variables*. The application of this set is described later in this section.
4. $\mathbf{ot} = \{ot_m^c \mid m \in \mathcal{M} \wedge c \in \mathcal{C}\}$, where each variable ot_m^c presents the time-related overhead (i.e., $\mathcal{O}_T^\delta(c)$ calculated by Algorithm 1) imposed on component c when monitored by m.
5. $\mathbf{om} = \{om_m^c \mid m \in \mathcal{M} \wedge c \in \mathcal{C}\}$, where each variable om_m^c presents the memory-related overhead (i.e., $\mathcal{O}_M^\delta(c)$ calculated by Algorithm 1) imposed on component c when monitored by m.
6. $\mathbf{z} = \{z_m^\delta \mid m \in \mathcal{M} \wedge \delta \in PP\}$, where each z_m^k is a Boolean variable. The application of this set is described later in this section.
7. $\mathbf{ov} = \{ov_m \mid m \in \mathcal{M}\}$, where each variable ov_m represents the cumulative normalized monitoring overhead imposed by m (i.e., $\mathcal{O}_\eta^{p_m}(m)$).

Constrains for polling periods. For every monitor $m \in \mathcal{M}$, we add the following constraint to model all the possible values for a polling period provided by PP:

$$s_m = \sum_{\delta \in PP} \delta \times z_m^\delta$$

Since s_m can have only one value from PP, we also add a constraint that *at most one* variable in $\{z_m^\delta\}_{\delta \in PP}$ can have a non-zero value.

Constrains for components. For each component $c \in \mathcal{C}$, we add the following constraints:

– *Constraint 1.* Each component $c \in \mathcal{C}$ must be monitored by one and only one monitor. Hence, variables $\{x_m^c\}_{m \in \mathcal{M}}$ are such that only *at most one* has a non-zero value and, hence:

$$\sum_{m \in \mathcal{M}} x_m^c \geq 1$$

– *Constraint 2.* When component c is monitored by a monitor m, x_m^c represents the normalized monitoring overhead for c, otherwise, $x_m^c = 0$. To model this behavior, we employ choice variables y_m^c and ${y'}_m^c$. Variables y_m^c and ${y'}_m^c$ are such that one represents the normalized monitoring overhead and the other is zero. When m monitors c, $y_m^c > 0$ and ${y'}_m^c = 0$, otherwise,

$y^c_m = 0$ and $y'^c_m > 0$. To this end, we have the following constraints for every monitor m:

$$x^c_m = y^c_m$$

$$y^c_m + y'^c_m = ot^c_m$$

As seen, y^c_m only represents the time-related overhead and not the memory-related overhead. This is because adding up ot^c_m and om^c_m that have different units (i.e., CPU cycles and bytes) is logically incorrect. Hence, we limit the memory-related overhead by defining an upper limit MEM.

$$\sum_{c \in \mathcal{C}} \sum_{m \in \mathcal{M}} om^c_m \leq MEM$$

We solve the optimization problem by running the model for different values of MEM and find the optimal solution.

– *Constraint 3.* For each monitor m, we calculate the time-related and memory-related overheads. Recall that $\Theta^c_{\mathcal{O}} = \{\mathcal{O}^\delta_\eta(c)\}_{\delta \in PP}$ is the set of normalized monitoring overheads where $\mathcal{O}^\delta_\eta(c) = \langle \mathcal{O}^\delta_T(c), \mathcal{O}^\delta_M(c) \rangle$ is calculated by Algorithm 1. To this end, we model all possible values of ot^c_m and om^c_m as follows:

$$ot^c_m = \sum_{\delta \in PP} \mathcal{O}^\delta_T(c) \times z^\delta_m, \text{ where } \mathcal{O}^\delta_T(c) \in \Theta^c_{\mathcal{O}}(\delta)$$

$$om^c_m = \sum_{\delta \in PP} \mathcal{O}^\delta_M(c) \times z^\delta_m, \text{ where } \mathcal{O}^\delta_M(c) \in \Theta^c_{\mathcal{O}}(\delta)$$

Constrains for monitors. The overhead imposed by a monitor m is the cumulative normalized monitoring overhead associated with all the components monitored by m. Hence, for each $m \in \mathcal{M}$, we have the following constraint:

$$ov_m = \sum_{c \in \mathcal{C}} x^c_m$$

Optimization objective. In general, our objective is to map the components to monitors such that the cumulative normalized monitoring overhead over all monitors is minimized and also the polling periods of the monitors is maximized. Thus, our ILP objectives are the following:

$$\min \sum_{m \in \mathcal{M}} ov_m$$

$$\sum_{m \in \mathcal{M}} s_m = MAX \tag{5}$$

where MAX is given as an input parameter to the ILP solver. Since off-the-shelf ILP solvers only support single objectives, we run the ILP model for all possible combination of values for s_m by setting MAX from 1 to $|PP| \times |\mathcal{M}|$ to find the optimal solution.

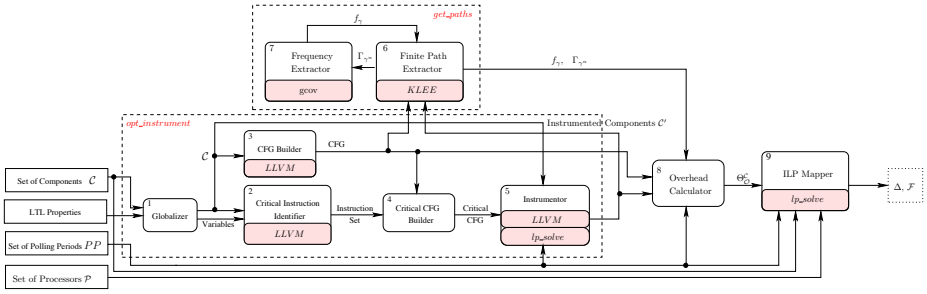


Fig. 4. Tool chain.

5 Implementation and Experimental Results

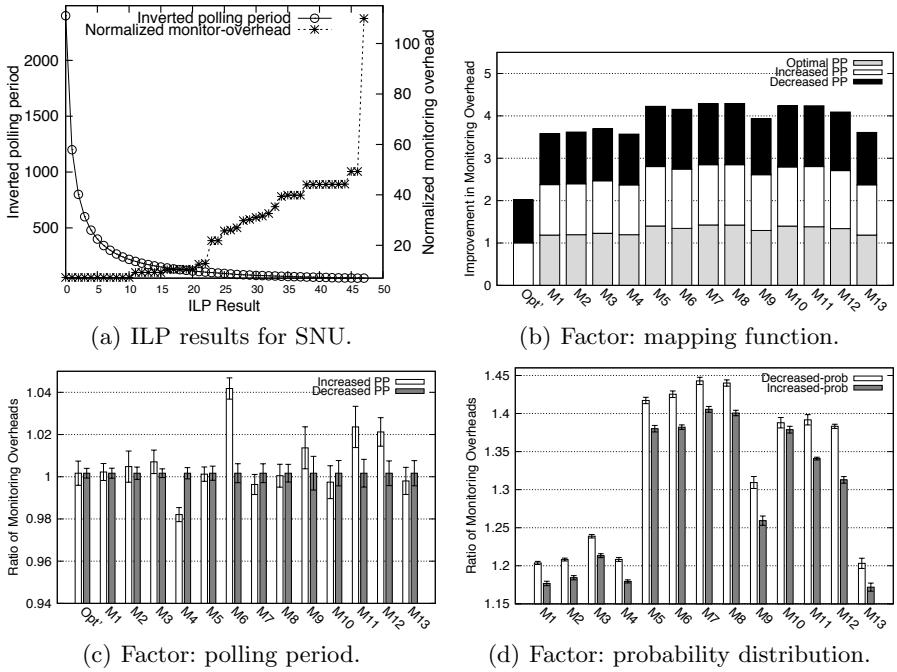
In this section we present our tool chain and the evaluation of our optimization approach.

5.1 Implementation

Figure 4 presents the modules of our tool chain that implements our solution for solving the optimization problem described in Subsection 3.3.

opt_instrument This module uses the techniques from [5] to optimally instrument the source code of the set of components $\mathcal{C}$ for each polling period in PP. It takes as input the source code of each component in $\mathcal{C}$, a set of Linear Temporal Logic (LTL) properties, and the set PP of possible polling periods. *Globalizer* extracts the set of variables of interest from the LTL properties and prepares the components in $\mathcal{C}$ for monitoring. *CFG Builder* extracts the CFG of each component in $\mathcal{C}$ and *Critical Instruction Identifier* finds the set of critical instructions of each component in $\mathcal{C}$. *Critical CFG Builder* uses the set of critical instructions and the set of CFGs to create the set of critical CFGs. *Instrumentor* uses the set of critical CFGs to optimally instrument the components in $\mathcal{C}$ for each polling period in PP by leveraging the ILP solver lp_solve and LLVM [16].

get_paths This module extracts the set of finite execution paths of each instrumented component in $\mathcal{C}'$ (see Subsection 4.1). *Finite Path Extractor (FPE)* leverages the symbolic execution tool KLEE [6]. FPE initially *adjusts* the terminating components in $\mathcal{C}'$ (see Subsection 4.1) and runs KLEE to extract the set of execution paths Γ_{γ^n} of the components in $\mathcal{C}'$. Since the components in $\mathcal{C}'$ are non-terminating, we set an upper bound on the analysis time of KLEE. If all the paths in Γ_{γ^n} achieve path and CFG coverage (see Subsection 4.1), FPE sends Γ_{γ^n} to *Frequency Extractor*, otherwise, FPE increases the analysis time and restarts KLEE. When CFG coverage is not satisfied, FPE checks whether the uncovered CFG is dead code. If so, FPE flags CFG coverage as unnecessary.



(a) ILP results for SNU.

(b) Factor: mapping function.

(c) Factor: polling period.

(d) Factor: probability distribution.

Fig. 5. Experimental results.

Note that for an execution path, by increasing the analysis time of KLEE, we can potentially increase the length of the path. *Frequency Extractor* uses gcov to extract the execution frequency of each instruction of an execution path in Γ_{γ^n}. If the paths in Γ_{γ^n} do not satisfy frequency coverage (see Subsection 4.1), FPE increases the analysis time of KLEE. When the paths in Γ_{γ^n} satisfy all three coverages, FPE reports Γ_{γ^n} and the set of execution frequencies f_γ.

Overhead Calculator This module implements Algorithm 1 using the characteristics of the computing cores and the overhead associated with each instrumentation instruction, memory read and write, running the monitoring code, etc. This module calls opt_instrument (line 2 of Algorithm 1) and get_paths (line 3 of Algorithm 1) to calculate the normalized monitoring overhead.

ILP Mapper This module solves the optimization problem and returns the polling period of each monitor (i.e., Δ) and the mapping of components to computing cores (i.e., function $\mathcal{F}$). It uses lp_solve to solve the ILP model for all the possible combinations of polling periods in PP (see Equation 5).solution which has the least monitoring overhead and largest polling period.

5.2 Experimental Settings

We use two interconnected MCB1700 boards, Core1 and Core2, both running the RTX operating system. The time-triggered monitor on each board is a task with read access to all the variables of interest. At each poll the monitor writes the variables of interest and the auxiliary memory to an SD card for the verification engine to retrieve. The auxiliary memory on each board is 1600 bytes. We consider the following factors: (1) the mapping function $\mathcal{F}$, (2) the polling period of each monitor, and (3) the probability distribution for executing the components. For evaluation, we run each experiment for one hour and measure the following metrics in 1-minute intervals (i.e., each experiment provides 60 data points): (1) the number of polls, (2) $\mathcal{O}_c^\delta$, $\mathcal{O}_i^\delta$, $\mathcal{O}_r^\delta$, and $\mathcal{O}_M^\delta$.

Our case study leverages the SNU benchmark suite [1] to create a component-based embedded system. Each program is a component that is invoked infinitely often according to a normal distribution. Moreover, the set of possible polling periods is $PP = \{5 \cdots 30\}$.

5.3 Analysis of Experiments

We ran the ILP model for each possible combination of polling periods used by the monitors running on the two boards. Figure 5(a) shows the ILP results (i.e., the optimal normalized monitoring overhead for each combination of polling periods). Since the normalized monitoring overhead is a complex number (see Algorithm 1), in Figure 5(a), normalized monitor-overhead is the distance of the optimal normalized monitoring overhead to the origin of a complex plane. Inverted polling period, in Figure 5(a), presents

$$\frac{1}{\sum_{m \in \mathcal{M}} s_m \times factor}$$

where $\mathcal{M} = \{\mathsf{Core1}, \mathsf{Core2}\}$, s_m is the polling period of the monitor on each core, and *factor* presents the impact that a monitor invocation along with monitoring code execution has on the monitoring overhead in comparison to an instrumentation. *factor* has a value of 100 in our experiments. Figure 5(a) shows that the settings from point 20 is the solution to the optimization problem. Row *Opt* in Table 2 presents the solution. Metric $\mathcal{F}$ is the mapping of components to cores and Δ is the optimal polling period.

Impact of Mapping Function We now evaluate the effectiveness of the mapping function of *Opt*. To this end, we change function $\mathcal{F}$ and recalculate Δ for each monitor using Equation 3. We create 13 different mappings, denoted by $M_\mathcal{F}$. Three mappings from $M_\mathcal{F}$ that have the closest monitoring overhead to *Opt* are shown in Table 2 (for reasons of space). Table 3 shows the experimental results for the mappings in Table 2. All the values are averages over 60 data points. *Polls* is the number of monitoring polls, $\mathcal{O}_T^\Delta\%$ is the percentage of the execution

Setting	Metric	Core1	Core2
Opt	Δ	21 cycles	23 cycles
	$\mathcal{F}$	bs, jfd, ludcmp, sqrt, matmul, minver, qsort, insertsort, select	crc, fibcall, fft, fir, lms, qurt
M_1	Δ	16 cycles	23 cycles
	$\mathcal{F}$	bs, crc, fft, fibcall, jfd, fir	minver, lms, ludcmp, matmul, qsort, select, sqrt
M_2	Δ	16 cycles	23 cycles
	$\mathcal{F}$	bs, fft, lms, ludcmp, minver	crc, fibcall, fir, jfd, matmul, qurt, qsort, select, sqrt
M_3	Δ	16 cycles	23 cycles
	$\mathcal{F}$	bs, crc, fibcall, jfd, matmul, ludcmp	fft, fir, lms, minver, qurt, qsort, select, sqrt

Table 2. Settings for SNU programs.

Setting	Poll	$\mathcal{O}_c^{\Delta}$ [ms]	$\mathcal{O}_i^{\Delta}$ [ms]	$\mathcal{O}_r^{\Delta}$ [ms]	$\mathcal{O}_T^{\Delta}$ [ms]	$\mathcal{O}_M^{\Delta}$ [byte]	$\mathcal{O}_T^{\Delta}$%	$\mathcal{O}_M^{\Delta}$%	$\sigma_{\mathcal{O}_T^{\Delta}}$	$\sigma_{\mathcal{O}_M^{\Delta}}$	$\mathcal{O}^{\Delta}$ [ms]	$\sigma_{\mathcal{O}^{\Delta}}$
Opt	250,617.98	1,700.69	6.26	2,607.79	4,314.75	925.33	7.02	57.83	4.75	24.73	4,412.84	12.30
M_1	361,275.83	2,443.04	5.82	2,727.42	5,176.29	827.2	8.62	51.7	4.48	28.67	5,242.58	10.81
M_2	360,929.61	2,441.17	7.74	2,744.72	5,193.64	984	8.65	61.5	4.68	27.29	5,279.62	13.28
M_3	368,088.71	2,493.52	9.07	2,828.90	5,331.50	1,004.26	8.88	62.73	5.13	31.83	5,425.21	12.84

Table 3. Monitoring overhead of SNU programs.

time consumed by the time-related overhead $\mathcal{O}_T^{\Delta}$, $\mathcal{O}_M^{\Delta}$% is the percentage of the consumed auxiliary memory $\mathcal{O}_M^{\Delta}$, $\mathcal{O}^{\Delta}$ is the absolute value of the monitoring overhead (i.e., the distance to the origin of a complex plane), and $\sigma_{\mathcal{O}_T^{\Delta}}$, $\sigma_{\mathcal{O}_M^{\Delta}}$, and $\sigma_{\mathcal{O}^{\Delta}}$ are the standard deviations of $\mathcal{O}_T^{\Delta}$, $\mathcal{O}_M^{\Delta}$, and $\mathcal{O}^{\Delta}$, respectively.

Table 3 shows that on average *Opt* imposes 20.46% less monitoring overhead in comparison to M_1–M_3. *Opt* imposes 11.86% more memory-related overhead $\mathcal{O}_M^{\Delta}$ in comparison to M_2. On the other hand, M_2 imposes 19.98% more time-related overhead $\mathcal{O}_T^{\Delta}$. Although, M_2 imposes less $\mathcal{O}_M^{\Delta}$, the impact of $\mathcal{O}_T^{\Delta}$ is stronger on $\mathcal{O}^{\Delta}$ (i.e., *factor* is 100 in our experiments). This observation matches with the observations in [5]. The larger $\mathcal{O}_T^{\Delta}$ of M_1–M_3 is caused by the larger number of polls which is the outcome of the smaller polling period on Core1. Moreover, the experiments on all the mappings in $M_{\mathcal{F}}$ show that on average *Opt* imposes 31.1% less monitoring overhead. *Optimal-PP*, in Figure 5(b), shows the ratio of the monitoring overhead associated with each mapping in $M_{\mathcal{F}}$ to the monitoring overhead associated with *Opt*. In addition, the results showed that our optimization approach has an error factor of 0.07. In other words, in one out of 13 mappings, our approach did not accurately reflect the monitoring overhead due to the normalization in Equation 4. The significance test on all these experiments show that the results are statistically significant with a p-value of less than 1^{-100}, hence, our approach can successfully find the mapping that results in the minimum monitoring overhead.

Impact of Polling Period We change the polling period of the monitor on each core for every mapping in $M_{\mathcal{F}}$. We once increase and once decrease the polling period of each monitor by 2 CPU cycles. Figure 5(b) shows the ratio of the monitoring overhead associated with each mapping in $M_{\mathcal{F}}$ to the monitoring overhead associated with *Opt* with respect to the changed polling periods. On average *Opt* imposes 32.81% less monitoring overhead in comparison to the monitoring overhead associated with the mappings in $M_{\mathcal{F}}$. Moreover, in Figure 5(c), for each mapping M, *increased/decreased-PP* presents the ratio of the monitoring overhead associated with mapping M when using the increased/decreased polling periods to the monitoring overhead of mapping M when using the optimal polling period Δ. The error bars reflect the standard deviation. The results show that by employing Δ, the mappings in $M_{\mathcal{F}}$ impose 1.39% less monitoring overhead. The error factor of this set of experiments is 0.19; i.e., in 5 out of 26 experiments, our approach did not correctly calculate Δ.

Robustness Analysis To calculate the normalize monitoring overhead, we assumed that the execution paths of each component and the set of components are executed based on a normal distribution. We change the probability distribution as follows to evaluate the robustness of our optimization approach:

1. We increase the probability distribution of the execution path(s) and component(s) with the highest normalized monitoring overhead, by 10 units. In Figure 5(d), *increased-prob* shows the ratio of the monitoring overhead associated with each mapping in $M_{\mathcal{F}}$ to the monitoring overhead associated with *Opt* with respect to the new probability distribution. The error bars reflect the standard deviation. Figure 5(d) shows that with moderate increases in the probability distribution, our approach is still effective since on average *Opt* imposes 33.14% less monitoring overhead. By also changing the polling period as in Subsection 5.3, on average *Opt* imposes 34.1% less overhead. The experiments show an error factor of 0.38, meaning that in 15 out of 39 experiments, our approach did not either correctly reflect the monitoring overhead or calculate the optimal polling period.
2. Likewise, we decrease the probability distribution of the execution path(s) and component(s). Figure 5(d) shows that with moderate decreases in the probability distribution, our approach is still effective since on average *Opt* imposes 27.27% less monitoring overhead. By also changing the polling period as in Subsection 5.3, on average *Opt* imposes 29.2% less overhead. The experiments show an error factor of 0.25, meaning that in 10 out of 39 experiments, our approach did not either correctly reflect the monitoring overhead or calculate the optimal polling period.

We note that the above error factors are expected, since our overhead calculations are based on normal distribution. Having said that, these experiments shows that changing the probability distribution does not significantly undermine the robustness of our approach. Thus, the insight is when the probability distribution of the components are approximately known, it is advisable to use

the knowledge when calculating the normalized monitoring overheads in Algorithm 1. In addition, in all the aforementioned experiments, the significance test show that the results are statistically significant with a p-value of less than 1^{-100}.

6 Related Work

Most runtime verification frameworks [7, 11, 14] use event-triggered monitoring. These frameworks are not suitable for time-sensitive systems. Time-triggered monitoring [5] ensures periodic monitoring with sound program state reconstruction. To reduce the overhead, the approach in [4, 5, 18] uses auxiliary memory to increase the polling period. The technique in [17] uses symbolic execution to adjust the monitor's polling period at run time according to the execution path. [2] discards instrumentation with respect to the system's execution path in event-based monitoring. [3,8] discards/adds monitoring instrumentation with respect to the properties being monitored.

There are numerous papers on event-triggered runtime monitoring of component-based systems, but to our knowledge, our approach is the first that handles time-triggered monitoring for component-based multi-core systems. [10] uses the BIP modelling language to formally model and introduce runtime verification into software components. [20] uses a software monitor that observes the system's input, output and partial internal states to check a set of assumption-guarantee rules and suffers from high computation runtime overhead.In [9], the authors design a monitor over the Eclipse modelling framework where its monitoring coverage changes at run time to keep the monitoring overhead bounded. The approach in [13] edits the UML diagrams of the component at the design level to embed a monitor inside a component. The majority of current literature on runtime monitoring of component-based systems focus on leveraging existing technology to achieve efficient monitoring and, hence, lose generality. Thus, in contrast to our approach that can be adopted into numerous technologies and architectures, their solutions are limited to their underlying infrastructure.

7 Conclusion

In this paper, we presented an effective optimization approach to minimize the monitoring overhead associated with time-triggered runtime verification (TTRV) of component-based multi-core embedded systems. In TTRV, a monitor runs in parallel with the components under inspection and polls the component's state periodically to evaluate a set of properties. The overhead imposed by TTRV is mainly affected by (1) the mapping of the components to computing cores, and (2) the polling period of the time-triggered monitors. Our proposed approach leverages control-flow analysis and symbolic execution to characterize the monitoring overhead associated with monitoring each component at run time. Then, it transforms the optimization problem (for finding the mapping of components to cores that incurs minimum overhead) to an integer linear program. We evaluated

our approach using the SNU benchmark. Experimental results show that our approach finds the optimal solution with a success rate of 80%. On average, our approach can reduce the monitoring overhead by 34%, as compared to various near-optimal monitoring patterns of the components at run time. In addition, our technique shows resilience towards changes in the probability distribution of invoking the components.

As for future work, we plan to further reduce the monitoring overhead by adjusting the polling period of the monitors at run time using symbolic execution techniques. Another interesting extension is to consider TTRV for distributed applications.

8 Acknowledgment

This work was partially sponsored by Canada NSERC Discovery Grant 418396-2012 and NSERC Strategic Grants 430575-2012 and 463324-2014.

References

1. SNU Real-Time Benchmarks. `http://www.cprover.org/goto-cc/examples/snu.html`.
2. C. Artho, D. Drusinksy, A. Goldberg, K. H. andM. Lowry, C. Pasareanu, G. Roşu, and W. Visser. Experiments with test case generation and runtime analysis. In *Proceedings of the 10th International Conference on Advances in Theory and Practice of Abstract State Machines*, ASM'03, pages 87–108, 2003.
3. E. Bodden, L. Hendren, P. Lam, O. Lhoták, and N. Naeem. Collaborative runtime verification with tracematches. In *Proceedings of the 7th International Conference on Runtime Verification*, RV'07, pages 22–37, 2007.
4. B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Sampling-based runtime verification. In *Formal Methods (FM)*, pages 88–102, 2011.
5. B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Time-triggered runtime verification. *Formal Methods in System Design (FMSD)*, 43(1):29–60, 2013.
6. C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, 2008.
7. F. Chen and G. Roşu. Java-mop: A monitoring oriented programming environment for java. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 546–550, 2005.
8. M. B. Dwyer, A. Kinneer, and S. Elbaum. Adaptive online program analysis. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 220–229, 2007.
9. J. Ehlers and W. Hasselbring. A self-adaptive monitoring framework for component-based software systems. In *Proceedings of the 5th European conference on Software architecture*, ECSA'11, pages 278–286, 2011.
10. Y. Falcone, M. Jaber, T. Nguyen, M. Bozga, and S. Bensalem. Runtime verification of component-based systems. In *Proceedings of the 9th international conference on Software engineering and formal methods*, SEFM'11, pages 204–220, 2011.

11. K. Havelund and G. Roşu. An overview of the runtime verification tool java pathexplorer. *Form. Methods Syst. Des.*, 24(2):189–215, 2004.
12. T. A. Henzinger and J. Sifakis. The embedded systems design challenge. In *Formal Methods (FM)*, pages 1–15, 2006.
13. M. U. A. Khan and M. Zulkernine. Building components with embedded security monitors. In *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures*, QoSA-ISARCS '11, pages 133–142, 2011.
14. M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-mac: A run-time assurance approach for java programs. *Form. Methods Syst. Des.*, 24(2):129–155, 2004.
15. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
16. 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.
17. S. Navabpour, B. Bonakdarpour, and S. Fischmeister. Path-aware time-triggered runtime verification. In *Third International Conference on Runtime Verification (RV)*, pages 199–213, 2012.
18. S. Navabpour, C. W. Wu, B. Bonakdarpour, and S. Fischmeister. Efficient techniques for near-optimal instrumentation in time-triggered runtime verification. In *Runtime Verification (RV)*, pages 208–222, 2011.
19. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2002.
20. M. Zulkernine and R. Seviora. Towards automatic monitoring of component-based software systems. *Journal of Systems and Softwares*, 74(1):15–24.