In Proc. of the International Conference on Runtime Verification (RV), Istanbul, Turkey, 2012.
Revised version.

Path-aware Time-trigged Runtime Verification

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

[1] Department of Electrical and Computer Engineering, University of Waterloo
Email: {snavabpo,sfischme}@uwaterloo.ca
[2] School of Computer Science, University of Waterloo
Email: borzoo@cs.uwaterloo.ca

Abstract. Time-triggered runtime verification aims at establishing a framework that provides bounded overhead and predictable behavior at run time. A time-triggered monitor runs in parallel with the program under inspection and periodically samples the program state to evaluate a set of properties. However, a time-triggered monitor working with a fixed sampling frequency often suffers from *redundant* sampling, which results in excessive overhead. In this paper, we propose an effective approach to reduce redundant sampling. Our approach calculates the sampling frequency with respect to the program behavior at run time. We further advance this approach to dynamically adjust the sampling frequency at run time by predicting the program behavior using symbolic execution. Our approach is fully implemented in a tool chain. Experiments on the SNU benchmark suit show that our approach reduces the sampling frequency, runtime overhead, and the number of redundant samples by up to 3.5 times, 69%, and 86%, respectively.

1 Introduction

In a computing system, *correctness* refers to the assertion that the system satisfies its specification at all times. Achieving system correctness is a major problem for today's large software systems. A recent NIST report estimates that 59.6 billion dollars are lost every year because of software errors [11]. Verification and testing are arguably the two most common approaches to ensure program correctness. However, verification may suffer from the state explosion problem, and testing may not be able to cover all possible execution scenarios of the system. These limitations argue for *runtime verification* [3, 9, 12], where a *monitor* inspects a program to evaluate a set of properties at run time.

Most monitoring approaches in runtime verification are *event-triggered*. In these approaches, the occurrence of an event of interest triggers the monitor for property evaluation. This technique leads to defects such as *unpredictable monitoring overhead* and potentially *bursts of monitoring invocation* at run time. Such defects can cause unpredictable behaviour from the system at run time; especially in real-time embedded safety/mission-critical systems, where it can result in catastrophic consequences. To tackle these drawbacks, in [6], we proposed *time-triggered* runtime verification, where the monitor runs in parallel with the program and samples the program state at *fixed* time intervals (i.e., the sampling period) to evaluate a set of properties. A challenge in implementing such a monitor is to compute the *longest sampling period*, such that all events of interest are observed. The approach in [6] computes the *fixed* longest sampling period with respect to all events of interests of a program by using the control-flow graph (CFG) of the program.

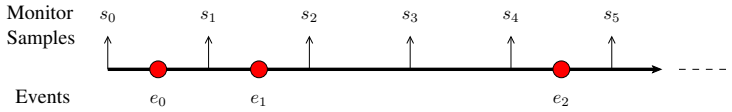


Fig. 1. Redundant monitoring intervention with fixed sampling period.

Although employing a time-triggered monitor results in observing bounded monitoring overhead and predictable monitoring invocation at run time [6], applying the fixed longest sampling period may result in unnecessary monitoring invocations. For example, consider the execution time line in Figure 1. Events e_0, e_1, and e_2 occur on the execution path and the monitor samples the program execution with the fixed longest sampling period at points $s_0, s_1, \ldots, s_5$. It is straightforward to observe that samples s_3 and s_4 are redundant, as no events of interest occur from sample s_2 until sample s_4. A large number of redundant samples cause the monitor to impose excessive overhead at run time. Thus, it is highly desirable to design a time-triggered monitor that can adjust the sampling period at run time based upon the characteristics of the program execution path.

With this motivation, in this paper, we propose the notion of *path-aware time-triggered monitoring*. To this end, we present an approach which leverages symbolic execution [16] to predict the execution path of the program with respect to the values of its inputs. Hence, to compute the *path-aware* longest sampling period, our approach only considers the events of interest in the portion of the program realized (i.e., executed) by the predicted execution path.

Moreover, we introduce a method that allows a time-triggered monitor to *adapt* its sampling period at run time, based on the events of interest to be executed in the near future. In particular, our method partitions the predicted execution path of the program into *code regions* and computes the path-aware longest sampling period of each region. Hence, when the program execution enters a new region at run time, the monitor adapts the path-aware longest sampling period of that region, thus avoiding redundant samples. Our method also ensures that the overhead of adapting the sampling period at run time imposes low overhead.

Finally, we present a fully automated tool chain that implements both path-aware and adaptive path-aware monitoring approaches. We study the effect of both approaches by using the SNU benchmark [1]. The experiments show highly promising results; i.e., our approach reduces the sampling frequency, runtime overhead, and the number of redundant samples by up to 3.5 times, 69%, and 86%, respectively.

Organization Section 2 presents the background concepts. In Section 3, we introduce the path-aware time-triggered monitoring approach. Section 4 is dedicated to adaptive path-aware time-triggered monitoring, while in Section 5 we describe our implementation and tool chain. Section 6 presents the experimental results. Related work is discussed in Section 7. Finally, in Section 8 we make concluding remarks and discuss future work.

2 Background

In time-trigged runtime verification (TTRV) a monitor samples the *state* of a program at regular time intervals to evaluate a set of properties. The state of the program is defined by a set of *variables of interest* that affect the evaluation of the set of properties. A time-triggered monitor exhibits the following characteristics which make it suitable for monitoring time-sensitive systems: (1) bounded overhead (i.e., the monitor imposes approximately the same amount of overhead at each sample), and (2) predictable invocation (i.e., the monitor samples the program execution at constant intervals, called the *sampling period*).

The main challenge with time-triggered monitoring is achieving *sound state reconstruction*. That is, the monitor must sample the program state with a frequency such that it observes all state changes (i.e., change in values of variables of interest) to evaluate the correctness of the execution path. Our previous work [6] showed that the monitor achieves sound state reconstruction by setting its sampling period to the shortest time interval between two consecutive program state changes as follows.

Let P be a program and Π be a logical property, where P is expected to satisfy Π. Let $\mathcal{V}_\Pi$ be the set of variables that change the valuation of Π. We leverage control-flow analysis to estimate the time intervals between consecutive state changes with respect to $\mathcal{V}_\Pi$.

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 *(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.* □

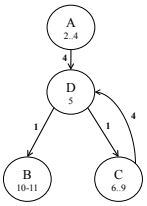
For example, consider the Fibonacci function shown in Figure 2(a) from the fibcall program of the SNU benchmark [1]. Assuming that the BCET of each instruction is one time unit, the resulting CFG is shown in Figure 2(b), where each vertex is annotated with the corresponding line numbers of the program. In order to calculate the longest sampling period (LSP), we modify CFG_P in two steps:

Step 1 (Extracting the Critical Vertices)
In this step, we modify the CFG_P such that each *critical instruction* (i.e., an instruction that updates the value of a variable in $\mathcal{V}_\Pi$) resides in a vertex by itself. We refer to such a vertex as a *critical vertex*. For example, if $\mathcal{V}_\Pi = \{\mathsf{Fnew}, \mathsf{Fold}, \mathsf{ans}\}$, instructions 7, 8 and, 10 are critical instructions in Fibonacci and the evolved CFG_P is shown in Figure 2(c). We call this graph a *critical control-flow graph (CFG)*.

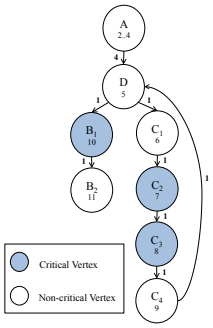
Step 2 (Calculating the Longest Sampling Period)

```
1. fib(int n) {
2.     int  i, Fnew, Fold, temp,ans;
3.     Fnew = 1;  Fold = 0;
4.     i = 2;
5.     while( i <= n ) {
6.         temp = Fnew;
7.*        Fnew = Fnew + Fold;
8.*        Fold = temp;
9.         i++; }
10.*   ans = Fnew;
11.    return ans;}
```

(a) Fibonacci Function



(b) CFG



(c) Critical CFG

Fig. 2. Fibonacci and its CFG and critical CFG.

As mentioned earlier, the main challenge in using TTRV is accurate state reconstruction. To preserve all state changes, the monitor must sample at a sampling period such it does not overlook any state changes that could occur in P at run time. This sampling period is called the *longest sampling period* (*LSP*).

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

For example, *LSP* of Fibonacci is 1 time unit. In this paper, we refer to such a sampling period as *fixed LSP*.

In [6], we observed that a time-triggered monitor with fixed *LSP* imposes 170% overhead on average. This is due to *redundant sampling*; i.e., the monitor may take samples even when the program does not execute a critical instruction since the last sample. To reduce the number of redundant samples, one can employ *auxiliary memory* to build a *history* of state changes between consecutive samples. In other words, let v be a critical vertex in a critical CFG, where the critical instruction *Inst* updates the value of a variable x in vertex v. The following graph transformation [6] results in another critical CFG with a greater *LSP*: it (1) removes v, (2) merges the incoming and outgoing arcs of v, and (3) adds an instruction $Inst' : x' \leftarrow x$ after the instruction *Inst* in the program source code, where x' is an auxiliary memory location. For example, applying this transformation to vertex C_2 in Figure 2(c) results in a graph where C_2 and all its incoming and outgoing arcs are removed, and a new arc from C_1 to C_3 with weight 2 is added. In the obtained graph, we have fixed $LSP = 2$.

3 Path-aware Time-triggered Monitoring

Although the experimental results from [6] show that by using auxiliary memory, the monitoring overhead can be reduced on average by 60%, the overhead still remains larger than the overhead of event-triggered runtime verification frameworks [8,13,15]. This is because the method in [6] uses the CFG of the program to compute the fixed *LSP* and not the realized execution paths at run time. Thus, the fixed *LSP* tends to be conservative; i.e., the monitor may take redundant samples.

To clarify, consider the critical CFG in Figure 2(c), where the fixed $LSP = 1$. This value is optimal when the program executes the instruction sequence $\langle v_A, v_D, v_{C_1}, v_{C_2}, v_{C_3}, v_{C_4}, v_D, v_{B_1}, v_{B_2}\rangle$ at run time (e.g., $n = 2$ as input). On the contrary, when the program executes the instruction sequence $\langle v_A, v_D, v_{B_1}, v_{B_2}\rangle$ (e.g., $n = 0$ as input), the fixed LSP is conservative. In the latter case, the monitor can extract all values of variables of interest with $LSP = 5$. In this case, the monitor takes 85% less redundant samples compared to using the fixed $LSP = 1$.

The above example motivates the idea to compute LSP with respect to the execution path at run time. This is achieved before program execution by the following two steps:

1. Predict the execution path that the program will take at run time with respect to the program's input values.
2. Using the predicted path from Step 1, compute LSP by only considering the sequence of critical instructions within the execution path.

Step 1 (Path Prediction)
Let P be a program, $CFG_P = \langle V, v^0, A, w\rangle$ be its control-flow graph, and $\mathcal{I}_P$ be the *input domain* of P. The input domain is the set of all values that the environment (e.g., a user) can provide as input to P.

Definition 3. *An* execution path *is a sequence of the form* $\gamma = \langle (v_0, \omega_0, v_1), (v_1, \omega_1, v_2), ...\rangle$*, 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* P *is a terminating program, 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.* □

For instance, in Fibonacci, the input value n=0 leads to the execution of path $\gamma_1 = \langle (v_A, 4, v_D), (v_D, 1, v_{B_1}), (v_{B_1}, 1, v_{B_2})\rangle$. In this paper, we only focus on *possible* execution paths; i.e., an execution path for which there exists some input in $\mathcal{I}_P$ that enables P to take the path at run time. We denote the set of all (possible) execution paths of program P as $\mathcal{P}_P$.

To *predict* the execution path(s) of P, we require a mechanism that takes the value-set of the inputs of P and returns the set of execution paths of P. We refer to this mechanism as the *path prediction* function.

Definition 4. *Let* P *be a program. The* path prediction *function* $\psi_P : \mathcal{I}_P \rightarrow 2^{\mathcal{P}_P}$*, maps an input from the input domain of* P *to a subset of execution paths of* P. □

Note that in a deterministic program, ψ_P maps an input to one and only one execution path. In practice, ψ_P can be implemented using symbolic execution [16] before the actual program execution. In particular, symbolic execution creates a bijection from each execution path γ of a program to a *path constraint*. A path constraint projects the conditions (e.g., in if-then-else and loop structures) that need to be satisfied in order for the program to execute γ at run time.

Notation: For each path γ, $PC(\gamma)$ denotes the path constraint of γ. For instance, for execution path $\gamma_1 = \langle (v_A, 4, v_D), (v_D, 1, v_{B_1}), (v_{B_1}, 1, v_{B_2}) \rangle$ of Fibonacci, we have $PC(\gamma_1) = (n < 2)$. Thus, ψ_P in fact, uses the input values (e.g., $n = 0$) and $PC(\gamma)$ to find the path constraint(s) satisfied by the input values, and extracts the set of associated execution path(s).

Step 2 (Computing the Sampling Period)
Given a predicted execution path γ from Step 1, the longest sampling period of γ is computed as follows. We refer to the following sampling period as *path-aware longest sampling period (paLSP)*.

Definition 5. *Let $x \in \mathcal{I}_P$ be an input and $\psi_P(x) = \{\gamma\}$. The* path-aware longest sampling period *paLSP for γ is the minimum subpath length between two critical vertices.* □

Now, consider a program that includes a loop structure. It is likely that an execution path γ of the program has multiple occurrences of a subpath of the form $\langle (v_i, \omega_i, v_{i+1}), (v_{i+1}, \omega_{i+1}, v_{i+2}), ..., (v_n, \omega_n, v_i) \rangle$. We refer to such a subpath as a *loop sequence*. Observe that multiple occurrences of a loop sequence in γ does not affect the value of *paLSP*. Hence, before computing *paLSP*, our approach transforms γ, such that each of its loop sequences occurs only once. We refer to the resulting execution path as the *unique* version of γ.

Definition 6. *Let γ be an execution path. The* unique execution path *of γ, denoted γ^{unq}, is a path, where every loop sequence in γ occurs only once in γ^{unq}.* □

Thus, our approach computes *paLSP* of γ using γ^{unq}. For a program P, a simple algorithm for computing *paLSP* of a unique execution path γ^{unq} takes the following steps:

1. Extract the subgraph CFG'_P of CFG_P that only includes path γ^{unq}.
2. Create the critical control-flow graph of CFG'_P.
3. Compute *paLSP* according to Definition 5.

We refer to a TTRV framework that uses *paLSP* as *path-aware TTRV* (pa-TTRV). In Section 6, we will show that pa-TTRV is quite effective in practice to reduce the runtime overhead.

4 Adaptive Path-aware Time-triggered Monitoring

Although pa-TTRV can effectively reduce the number of redundant samples, it still imposes excessive redundant samples when only a small fraction of the execution path needs to be sampled using the conservatively computed *paLSP*. For instance, if the Fibonacci function (Figure 2(c)) takes the hypothetical execution path $\gamma_2 = \langle v_A, v_D, v_{B_1}, v_{B_2}, v_A, v_D, v_{C_1}, v_{C_2}, v_{C_3}, v_{C_4}, v_D, v_{B_1}, v_{B_2} \rangle$, then $paLSP = 1$. However, if we apply $paLSP = 5$ up until vertex v_{C_2} (which can sample all critical events) and adjust is to $paLSP = 1$ afterwards, then the number of samples drops by 62%.

4.1 Code Regions

Intuitively, our idea to reduce redundant samples in an execution path is to dynamically change *paLSP* according to the *code region* of the execution path.

Definition 7. *Let γ be a unique execution path. A* code region *is a set of subpaths of γ with the same paLSP, where each subpath is* maximal. *That is, for each code region, if a subpath is extended, it no longer belongs to that code region.* □

Since each subpath has a unique *paLSP*, each code region is an equivalence class.

To sample an execution path with an *adaptive paLSP*, our approach needs to somehow *regionalize* the path based on Definition 7. In this case, when the program starts executing a subpath of a code region, the monitor *adapts* to *paLSP* of that code region at run time. Clearly, the regionalization can partition an execution path in various ways. Our objective is to regionalize an execution path such that adapting *paLSP* at run time does not add excessive overhead. We break down our objective as follows:

1. Reducing the number of code regions; i.e., since change of code region and, hence, sampling period at run time incurs some overhead.
2. Reducing the number of samples taken on the execution path.
3. Maintaining the absolute jitter (i.e., the difference between the minimum and maximum *paLSP* of code regions) below a predefined threshold Δ_{paLSP} provided by the designer. Note that this objective ensures *predictable invocation* of the monitor.

We refer to a TTRV framework that uses this method as *adaptive pa-TTRV*.

4.2 A Regionalization Algorithm

The algorithm Regionalize addresses the above objectives (see Algorithm 1). It takes as input (1) the bound Δ_{LSP} on the absolute jitter, (2) the overhead of changing the sampling period O_{LSP}, and (3) a unique execution path γ. Its output is a regionalization. The intuitive idea is that the algorithm creates all possible regionalizations and chooses the one with the least switching and sampling overhead.

The algorithm creates all possible regionalizations using three nested loops: (1) the for-loop (Lines 4-38), (2) the while-loop (Lines 8-37), and (3) the for-loop (Lines 15-35). Each iteration of each loop creates a new regionalization, where each regionalization is different from the other (created in the same loop) by one vertex. Notice that the first for-loop adds/removes vertices from subpath $\langle(v_0, w_0, v_1), \ldots, (v_i, w_i, v_{i+1})\rangle$, the while-loop adds/removes vertices from subpath $\langle(v_{i+1}, w_{i+1}, v_{i+2}), \ldots, (v_{base-1}, w_{base-1}, v_{base})\rangle$, and the second for-loop adds/removes vertices from subpath $\langle(v_{base}, w_{base}, v_{base+1}), \ldots, (v_{n-1}, w_{n-1}, v_n)\rangle$.

When a regionalization $temp_{reg}$ is created (Line 23), the algorithm computes the monitoring overhead of that regionalization. To this end, the monitor computes *paLSP* (line 26), and the BCET of each code region of the regionalization (line 27). Respectively, it computes the monitoring overhead by considering the number of samples taken in each code region and the cost of changing code regions (line 28). If the new regionalization has a lower monitoring overhead compared to the previously chosen regionalization, and the absolute jitter of its *paLSP*s is bounded by Δ_{LSP}, the algorithm chooses the new regionalization as the solution (lines 31- 34).

Algorithm 1 Regionalize

Input: Δ_{LSP}:bound on absolute jitter of $paLSP$s, O_{LSP}: overhead of changing code regions, γ: a unique execution path
Output: A regionalization

```
1: regionalization ← ∅
2: Overhead_reg ← ∞
3: n ← Length(γ)
   /* iterate over the vertices of γ */
4: for i = 0 to n − 1 do
5:     temp_reg ← ∅ /* new regionalization */
6:     reg ← ⟨(v_0, γ_0, v_1), ..., (v_i, γ_i, v_{i+1})⟩
       /* regionalization of remainder of γ */
7:     base ← i + 1
8:     while (base ≤ n − 1) do
9:         temp_reg ← temp_reg ∪ reg
           /* put vertex in γ up to base into separate regions */
10:        for m = i + 1 to base − 1 do
11:            reg′ ← ⟨(v_m, γ_m, v_{m+1})⟩
12:            temp_reg ← temp_reg ∪ reg′
13:        end for
14:        reg″ ← ⟨(v_base, γ_base, v_{base+1})⟩
15:        for m = base to n − 1 do
16:            for j = base + 1 to m do
17:                reg″ ← append(reg″, (v_j, γ_j, v_{j+1}))
18:            end for
19:            temp_reg ← temp_reg ∪ reg″
               /* put the remainder of γ in separate regions */
20:            for q = m + 1 to n − 1 do
21:                reg‴ ← ⟨(v_q, γ_q, v_{q+1})⟩
22:                temp_reg ← temp_reg ∪ reg‴
23:            end for
               /* calculate overhead for new regionalization */
24:            Overhead ← 0
25:            for all reg ∈ temp_reg do
26:                Compute paLSP_reg based on Definition 5
27:                total ← The sum of weights of arcs in reg
28:                Overhead ← Overhead + paLSP_reg / total + O_LSP
29:            end for
30:            Δ_reg ← Absolute jitter of paLSPs of temp_reg
               /* Update the best regionalization */
31:            if Overhead < Overhead_reg and Δ_reg ≤ Δ_LSP
               then
32:                regionalization ← temp_reg
33:                Overhead_reg ← Overhead
34:            end if
35:        end for
36:        base ← base + 1
37:    end while
38: end for
39: return regionalization
```

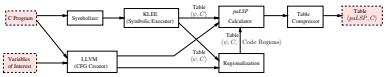


Fig. 3. The tool chain

4.3 General Code Regionalization

Observe that Definition 7 and Algorithm 1 identify a regionalization for an execution path. Hence, if two execution paths in a program share a common subpath, the sampling periods computed by Algorithm 1 for the subpath are not necessarily equal. For instance, consider the following two execution paths: $\gamma_1 =$

$\langle(v_0, 5, v_1), (v_1, 10, v_2), (v_2, 15, v_3)\rangle$ and $\gamma_2 = \langle(v_0, 5, v_1), (v_1, 1, v_5), (v_5, 2, v_6)\rangle$, where $\Delta_{LSP} = O_{LSP} = 5$. Algorithm Regionalize computes the following two code regions for γ_1: (1) $reg_1 = \{\langle(v_0, 5, v_1)\rangle\}$, where $paLSP_{reg_1} = 5$, and (2) $reg_2 = \{\langle(v_1, 10, v_2), (v_2, 15, v_3)\rangle\}$, where $paLSP_{reg_2} = 10$. On the contrary, for γ_2, the algorithm computes a single code region $reg_{\gamma_2} = \{\gamma_2\}$, where $paLSP_{reg_{\gamma_2}} = 1$. Hence, subpath $\langle(v_0, 5, v_1)\rangle$ resides in different regions with different *paLSP*s for execution paths γ_1 and γ_2. Thus, in environments where a unique regionalization among all execution paths of the program is desirable, we generalize the regionalization process as follows.

Definition 8. *Let $CFG = \langle V, v^0, A, w\rangle$ be a control-flow graph. In a* general regionalization*, each arc $(u, v) \in A$ appears in one and only one code region.* □

In this case, the monitor *adapts* the *paLSP* of a code region *reg* at runtime when (1) the program initiates the execution of a subpath in *reg*, and (2) *reg* differs from the code region of the previously executed subpath. Obtaining a general regionalization that optimally satisfies the three objectives mentioned in Subsection 4.1 has exponential complexity. In Section 5, we present an efficient approach to implement general regionalization.

5 Implementation

In this Section, we present the tool chain that computes *paLSP* and adaptive *paLSP* and the implementation of the time-triggered monitor.

5.1 Tool Chain

We have implemented our technique for computing *paLSP* and adaptive *paLSP* in a tool chain (see Figure 4). The tool takes a C program and a set of variables of interest as input. First, it extracts the CFG and subsequently the critical CFG of the program. In order to implement function ψ_P (see Definition 4) (i.e., predicting the execution path), the tool chain first symbolizes the input variables of the program. Then, it feeds the symbolized program to the tool Klee [7]. As a result, Klee creates a mapping table from each unique execution path of the program to its path constraint. We modified Klee using a patch, such that it converts an execution path to its unique version (see Definition 6). In case there exists duplicate unique paths, our Klee patch only keeps the path with the weakest path constraint.

To compute the adaptive *paLSP*, the tool chain regionalizes the program using general regionalization (see Definition 8). To this end, the tool chain considers all the arcs in between two consecutive conditional statements in the CFG of the program as one code region. Consequently, the tool chain maps each execution path to the set of its code regions. The *paLSP* calculator uses the critical CFG to compute *paLSP* and adaptive *paLSP* of each execution path in the mapping table. For adaptive *paLSP*, the *paLSP* calculator computes *paLSP* of each region of each execution path.

In general, the size of the (execution path to path constraint) mapping table may grow exponentially with respect to the number of execution paths. Hence, looking up the mapping table at run time imposes a large overhead. Thus, it is desirable to construct a smaller version of the table to be used at run time. To this end, the tool chain applies two techniques to eliminate entries:

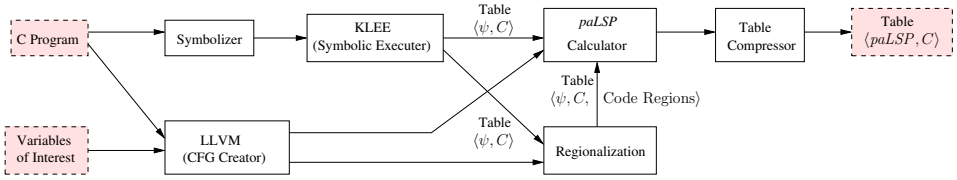


Fig. 4. The tool chain.

1. *Implication Reduction*: This technique groups the execution paths whose *paLSP* is defined by the same arc (u, v) in the critical CFG. For each group, it extracts the path constraint whose satisfaction leads to the execution of (u, v). Then, it represents the execution paths in the group with a table entry, that maps the extracted path constraint to *paLSP* of the execution paths in the group. This table entry also incorporates the *union* of the set of code regions of the execution paths in the group.
2. *paLSP Reduction*: This technique removes all entries from the mapping table, where *paLSP* of the execution path is similar to the fixed *LSP*.

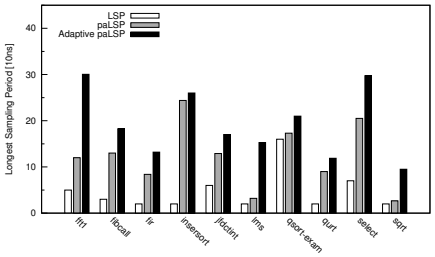
The final mapping table maps a path constraint to a *paLSP* and a set of code regions along with their *paLSP*. These techniques, on average, reduce the mapping table of SNU programs by 78% without loss of precision. In other words, *paLSP* and adaptive *paLSP* of an execution path does not change. When the input values do not satisfy a path constraint, the satisfiable path constraint is removed by *paLSP* reduction, hence, the monitor sets its sampling period to the fixed *LSP*.

There are cases where KLEE can not process all execution paths because of its limitations. Hence, the tool chain takes two conservative approaches: (1) when there is unanalyzed code, it assumes that this code gets executed at all times and, hence, appends it to all execution paths, and (2) when the analysis of an execution path γ is incomplete, the tool chain finds all possible unique subpaths that can be executed after γ and, hence, creates a new path by appending these subpaths to γ.

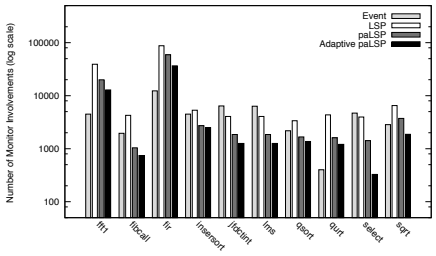
5.2 Implementing a Path-aware Time-triggered Monitor

The time-triggered monitor is a C program which runs in parallel with the input program (under scrutiny). The monitor has read-access to the memory location of the variables of interest. The monitor runs in three modes: *fixed*, *path-aware*, and *adaptive*. In the fixed mode, the monitor samples the program using the fixed *LSP*. In the path-aware mode, at each point at run time, where the program receives a new input from the environment[3], the monitor looks up the mapping table and evaluates path constraints to identify the proper *paLSP*. Then, the monitor adjusts the sampling period accordingly. In the adaptive mode, the monitor applies all features of the path-aware mode. In addition, the program is instrumented, so that when it reaches a new code region, it notifies the monitor to adjust its sampling period accordingly.

[3] Examples include executing instructions such as scanf, read, fscanf, etc.



(a) Longest Sampling Period



(b) Redundant Samples of fixed *LSP*, *paLSP*, and Adaptive *paLSP*

Fig. 5. Sampling period and redundant samples.

6 Experimental Results

In this Section, we present the experimental results. We use a selected set of programs from the SNU benchmark [1] to evaluate our techniques. Programs not discussed in this section exhibit similar behavior. The experimental setting is as follows. In each program, the main function runs 100 times, where at each iteration the main function receives new input values from the environment. We devised the input values, such that each unique execution path of the program executes at least once. The program and the time-triggered monitor run on an MCB1700 board with RTX real-time operating system. The time-triggered monitor for each program runs in four modes: (1) fixed *LSP*, (2) path-aware *LSP*, (3) adaptive *paLSP*, where Δ_{LSP} and O_{LSP} are 50ns, (4) program augmented with history (cf. Section 2) and the sampling period of $50 \times$ fixed *LSP*, $50 \times$ *paLSP*, and $50 \times$ adaptive *paLSP*. We measure the following metrics for evaluating our methods:

1. The values of the fixed *LSP*, *paLSP*, and adaptive *paLSP*.
2. The number of redundant samples taken at run time by the monitor.
3. The execution time of the monitored program. This value projects the amount of monitoring overhead.

6.1 Impact of pa-TTRV and Adaptive pa-TTRV on Sampling Period

Figure 5(a) shows the fixed *LSP*, *paLSP*, and adaptive *paLSP* of each program. The *paLSP* of each program is the average *paLSP* over all unique execution paths of the program. As for the adaptive *paLSP*, for each unique execution path, we consider the average *paLSP* over all the code regions of the execution path. Respectively, the adaptive *paLSP* of each program is the average adaptive *paLSP* over all unique execution paths of the program. The results show that the average *paLSP* and adaptive *paLSP* of all programs are 2.4 and 3.34 times greater than their fixed *LSP*.

Observe that in some programs, *paLSP* is considerably greater than its fixed *LSP* (e.g., in insertsort this is 12.2 times). Our studies show that such programs have at least one of the two following characteristics:

– The majority of the execution paths do not incorporate critical instructions and, hence, do not require monitoring. For instance, 66.66% of the execution paths of insertsort do not require monitoring.

– In the majority of the execution paths, the critical instructions are sparsely distributed and, hence, the sampling period of the monitor is greater than the fixed *LSP*. For instance, for 50% of the execution paths of select, *paLSP* is 130ns while the fixed *LSP* of select is 70ns.

On the contrary, in programs such as sqrt and lms, *paLSP* is moderately larger than the fixed *LSP*. Our studies show that such programs have at least one of the two following characteristics:

– The majority of the execution paths execute the two consecutive critical instructions that define the fixed *LSP* of the program and, hence, their *paLSP* is equal to the fixed *LSP*. For instance, for 75% of sqrt's execution paths, *paLSP* is 20ns which is the same as sqrt's fixed *LSP*.
– In the majority of the execution paths, the critical instructions are densely distributed. For instance, 54% of the execution paths of lms have *paLSP* of 40ns while lms's fixed *LSP* is 20ns.

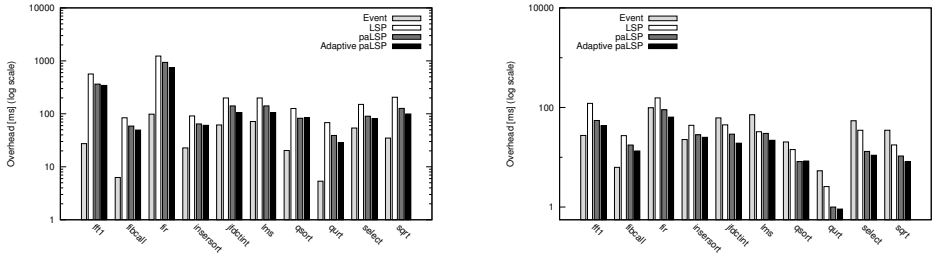
In addition, in programs such as select, adaptive *paLSP* is considerably greater than *paLSP*. In the execution path of such programs, the critical instructions are densely concentrated in a small fraction of the execution path, while in the remaining of the path the critical instructions are sparse. For instance, in select, the critical instructions *only* reside in the function SWAP, where *paLSP* of an execution path executing SWAP is as low as 70ns. Hence, the adaptive *paLSP* of such execution paths is 48.75% larger than their *paLSP*.

6.2 Impact of pa-TTRV and Adaptive pa-TTRV on Redundant Samples

Figure 5(b) shows the number of redundant samples of a time-triggered monitor when using fixed *LSP*, *paLSP*, and adaptive *paLSP*. Note that the y-axis is in log scale. Each *Event* bar shows the number of critical instructions executed throughout the program run. Bars *LSP*, *paLSP*, and adaptive *paLSP* show the number of redundant samples in the first three monitoring modes. The number of redundant samples is the difference between the total number of taken samples and the number of executed critical instructions. On average, when using *paLSP*, redundant samples decrease by 44.87%, and when using adaptive *paLSP*, redundant samples decrease by 64.04%. Our analysis shows that in programs such as qurt and select, if the execution of paths with *paLSP* greater than the fixed *LSP* dominate the execution at run time, then using *paLSP* results in larger reductions in the number of redundant samples. On the contrary, for programs such as sqrt and fir, we see small reduction in the number of redundant samples, since the majority of the executed paths at run time have *paLSP* equal to the fixed *LSP*. Note that a large percentage of execution paths with *paLSP* equal to the fixed *LSP* does not imply that the program execution at run time is dominated by these paths.

6.3 Impact of pa-TTRV and Adaptive pa-TTRV on Monitoring Overhead

Figure 6(a) shows the monitoring overhead of a time-triggered monitor when using fixed *LSP*, *paLSP*, and adaptive *paLSP*. Each *Event* bar shows the execution time of the monitored program when using an event-triggered monitor. Bars *LSP*, *paLSP*, and adaptive *paLSP* show the execution time of the monitored program



(a) Overhead of fixed *LSP*, *paLSP*, and Adaptive *paLSP*

(b) Overhead of 50 × *LSP*, 50 × *paLSP*, and 50 × adaptive*paLSP*

Fig. 6. Monitoring overhead.

in the first three monitoring modes. On average, monitoring overhead decreases by 39.34% when using *paLSP*, and by 51.28% when using adaptive *paLSP*. In programs such as qurt and select, when using *paLSP*, the monitoring overhead does not decrease in the same proportion as the redundant samples. For instance, in select, the number of redundant samples deceases by 72.04%, while the monitoring overhead decreases 51.83%. This is because, the overhead caused by the monitor to find the satisfied path constraint (using the lookup table discussed in Section 5) and adjust its sampling period, is large. Hence, we see less reduction in the monitoring overhead.

The same side affect is seen when using adaptive *paLSP* in programs such as select and fibcall. The overall overhead of looking up the mapping table in the adaptive path-aware monitor is larger compared to the path-aware monitor, since the monitor looks up the table more frequently (i.e., at each entry to a new code region). In some cases, the overall look up overhead is such that the monitoring overhead of the adaptive path-aware monitor exceeds the monitoring overhead of the path-aware monitor, although the adaptive path-aware monitor reduces more redundant samples. For instance, in qsort, the monitoring overhead of the adaptive path-aware monitor is 84.388ms, and the monitoring overhead of the path-aware monitor is 82.477ms, while the adaptive path-aware monitor removes 18.22% more redundant samples.

To further reduce redundant samples and monitoring overhead, we augment each program with history (see Section 2). We increase the fixed *LSP*, *paLSP*, and adaptive *paLSP* by a factor of 50. Experimental results show that for the SNU programs, the sampling period of 50 × fixed *LSP*, 50 × *paLSP*, and 50 × adaptive *paLSP* cause zero redundant samples. Furthermore, Figure 6(b) shows that in 66% of the programs, the overhead of the path-aware monitor is less than the overhead of the event-triggered monitor, and in 75% of the programs, the overhead of the adaptive path-aware monitor is less than the overhead of the event-triggered monitor.

7 Related Work

In general, runtime verification frameworks [8, 13, 15] use event-triggered monitoring. These frameworks are not suitable for time-sensitive systems. On the other

hand, [6] uses time-triggered monitoring. In particular, the approach in [6] calculates the longest sampling period which ensures sound program state reconstruction. This method may impose a large monitoring overhead. In [6] and [18], the authors propose using auxiliary memory to increase the sampling period which in turn reduces the runtime overhead.

Regardless of the type of monitor, runtime verification frameworks must impose low monitoring overhead. [3] reduces the overhead by rewriting safety properties such that the evaluation of properties requires the least information regarding the program execution. [5] reduces the number of instrumentation, by determining locations in the program which do not affect property evaluation. [4] distributes the instrumentation cost among multiple users. [14] controls the overhead by temporarily disabling monitoring of selected data, by using supervisory control theory of discrete event systems and PID-control theory of discrete time systems. [10] extracts only a subset of the data required to evaluate program properties, by removing/adding instrumentation relevant to the program state at runtime.

[2] is the closest work to our path-aware and adaptive path-aware frameworks. This approach discards instrumentation with respect to the execution path of the program. Our frameworks surpass [2] in the following aspects. To our knowledge, the approach in [2] manually extracts the path constraints for each path and, hence, it can only handle very small programs. On the contrary, our frameworks are fully automated and can handle medium to large size programs. Moreover, unlike our rigorous analysis of multiple case studies, [2] only presents the manual analyses of two small case studies and lacks strong evidence on the effectiveness of their method. In addition, [2] is only applicable when all input values are known a priori. On the other hand, our frameworks can handle inputs provided dynamically throughout the program run. Finally, [2] does not intelligently discard instrumentation at run time. On the contrary, the adaptive path-aware framework dynamically adjusts the sampling period at runtime to further reduce monitoring overhead.

8 Conclusion

In this paper, we presented an effective method for reducing the overhead of time-triggered runtime verification (TTRV). In TTRV, a monitor runs in parallel with a program under inspection and samples the program state periodically to evaluate a set of properties [17]. The main drawback of TTRV is its excessive runtime overhead due to redundant sampling, where the monitor may take samples from the program even if no new event for monitoring has occurred. Our proposed method in this paper leverages symbolic execution in order to predict the program's execution path, in order to intelligently choose the longest sampling period. In particular, we proposed *path-aware* TTRV, where the monitor adjusts its sampling period based on the given input values to the program under inspection. We also introduced *adaptive* path-aware TTRV, where the monitor adjusts its sampling period at runtime based on the density of occurrence of events that need to be monitored in a code region. Our techniques are implemented in a tool chain and the result of experiments show that adaptive path-aware TTRV reduces the runtime overhead and the number of redundant samples by up to 69% and 86%, respectively.

There are several future research directions. An open problem is merging rigorous execution time analysis with our method to accurately measure and incorporate

the cost of switching the sampling period in the algorithm presented in Section 4. Another interesting problem is to combine symbolic execution in our approach with (1) dynamic analysis techniques such as employing PID controllers [14], or (2) learning techniques such state estimation [19] to adjust the sampling period at runtime.

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. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'04, pages 44–57, 2004.
4. 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.
5. E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07, pages 525–549, 2007.
6. B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Sampling-based runtime verification. In *Formal Methods (FM)*, pages 88–102, 2011.
7. 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.
8. 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.
9. S. Colin and L. Mariani. *Run-Time Verification*, chapter 18. Springer-Verlag LNCS 3472, 2005.
10. 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.
11. M. Gallaher and B. Kropp. The economic impacts of inadequate infrastructure for software testing. In *National Institute of Standards & Technology Planning Report 02–03*, 2002.
12. K. Havelund and A. Goldberg. Verify your Runs. pages 374–383, 2008.
13. K. Havelund and G. Roşu. An overview of the runtime verification tool java pathexplorer. *Form. Methods Syst. Des.*, 24(2):189–215, 2004.
14. X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok. Software monitoring with controllable overhead. *Software tools for technology transfer (STTT)*, 14(3):327–347, 2012.
15. 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.
16. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

17. S. Navabpour, B. Bonakdarpour, and S. Fischmeister. Software debugging and testing using the abstract diagnosis theory. In *Languages, compilers, and tools for embedded systems (LCTES)*, pages 111–120, 2011.
18. 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)*, pages 208–222, 2011.
19. S. D. Stoller, E. Bartocci, J. Seyster, R. Grosu, K. Havelund, S. A. Smolka, and E. Zadok. Runtime verification with state estimation. In *Runtime Verification (RV)*, pages 193–207, 2011.