In Proc. of the 2nd International Conference on Runtime Verification, San Francisco, USA, 2011.
Revised version.

Efficient Techniques for Near-optimal Instrumentation in Time-triggered Runtime Verification

Samaneh Navabpour[1], Chun Wah Wallace Wu[1], Borzoo Bonakdarpour[2], and Sebastian Fischmeister[1]

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

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

Abstract. *Time-triggered* runtime verification aims at tackling two defects associated with runtime overhead normally incurred in event-triggered approaches: *unboundedness* and *unpredictability*. In the time-triggered approach, a monitor runs in parallel with the program and periodically samples the program state to evaluate a set of properties. In our previous work, we showed that to increase the sampling period of the monitor (and hence decrease involvement of the monitor), one can employ auxiliary memory to build a history of state changes between subsequent samples. We also showed that the problem of optimization of the size of history and sampling period is NP-complete.
In this paper, we propose a set of heuristics that find near-optimal solutions to the problem. Our experiments show that by employing negligible extra memory at run time, we can solve the optimization problem significantly faster, while maintaining a similar level of overhead as the optimal solution. We conclude from our experiments that the NP-completeness of the optimization problem is not an obstacle when applying time-triggered runtime verification in practice.

Keywords: Runtime monitoring, instrumentation, optimization, verification, time-triggered, predictability.

1 Introduction

Runtime verification [1, 2, 4, 7, 9, 13] refers to a technique where a system under inspection is continually checked by a *monitor* at run time with respect to its

specification. Runtime verification complements exhaustive verification methods, such as model checking and theorem proving, as well as incomplete solutions, such as testing and debugging. This is because exhaustive verification often requires developing a rigorous abstract model of the system and suffers from the state-explosion problem. Testing and debugging, on the other hand, provide us with under-approximated confidence about the correctness of a system, as these methods only check for the presence of defects under specific conditions.

In the literature, deploying runtime verification involves instrumenting the program under inspection, so that upon occurrence of events (e.g., change of value of a variable) that may change the truthfulness of a property, the monitor is called to re-evaluate the property. We call this method *event-triggered* runtime verification, because each change prompts a re-evaluation. Event-triggered runtime verification suffers from two drawbacks: (1) *unpredictable* overhead, and (2) possible *bursts* of events at run time. These defects can lead to undesirable transient overload situations in time-sensitive systems such as real-time embedded safety-critical systems. To address these issues, in [3], we introduced the notion of *time-triggered* runtime verification, where a monitor runs in parallel with the program and samples the program state periodically to evaluate a set of system properties.

The main challenge in time-triggered runtime verification is to guarantee accurate program state reconstruction at sampling time. To this end, we introduced an optimization problem where the objective is to find the minimum number of critical events that need to be buffered for a given sampling period [3]. Consequently, the time-triggered monitor can successfully reconstruct the state of the program between two successive samples. We showed that this optimization problem is NP-complete and proposed a transformation of this problem to an *integer linear program* (ILP). This transformation enables us to employ powerful ILP-solvers to identify the minimum buffer size and instrumentation instructions for state reconstruction. It is possible to solve the corresponding ILP model for some applications, but for larger applications, the exponential complexity poses a serious stumbling block.

With this motivation, in this paper, we focus on developing polynomial-time algorithms that find near-optimal solutions to the optimization problem. Our algorithms are inspired by an observation made in [3]. Figure 1, taken from [3], shows the decrease in execution time and increase in total memory usage of a program (y-axis) when the sampling period (denoted *MSP* in Figure 1) is increased by factors of 20, 50, 70, and 100 (x-axis). Increasing the sampling period requires storing more events, and hence, requiring larger buffers. However, Figure 1 shows that when we increase the sampling period even by a factor of 100, the increase in memory usage is only 4%. In other words, the impact of increasing the sampling period on memory usage is negligible. Our experiments on other programs exhibit the same behavior. This observation suggests that nearly optimal solutions to the optimization problem are likely to be sufficiently effective.

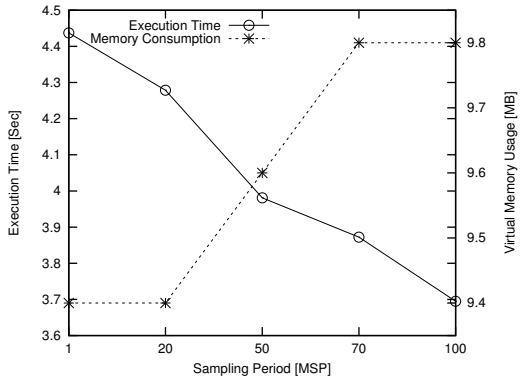


Fig. 1. Memory usage vs. sampling period [3].

We propose three polynomial-time heuristics. All heuristics are over-approximations and, hence, sound (they do not cause overlooking of events to be monitored). The first heuristic is a greedy algorithm that aims at instrumenting variables that participate in many execution branches. The second heuristic is based on a 2-approximation algorithm for solving the minimum vertex cover problem. Intuitively, this heuristic instruments variables that are likely to cover all cases where variable updates occur within time intervals less than the sampling period. The third heuristic uses genetic algorithms, where the population generation aims at minimizing the number of variables that need to be instrumented and buffered.

The results of our experiments show that our heuristics are significantly faster than the ILP-based solution proposed in [3]. More importantly, the solutions returned by all three algorithms lead to a negligible increase in instrumentation overhead and total memory usage at run time as well as negligible increase in the total execution time of the monitored program. We also observe that in general, extra instrumentation instructions are evenly distributed between samples. Moreover, our genetic algorithm generally produces instrumentation schemes closest to the optimal solution as compared to the other heuristics. Based on the results of our experiments, we conclude that the NP-completeness of the optimization problem is not an obstacle when applying time-triggered runtime verification in practice.

Organization. The rest of the paper is organized as follows. In Section 2, we review the concept of time-triggered runtime verification. The first two heuristics are presented in Section 3. Section 4 is dedicated to our genetic algorithm. We analyze the results of experiments in Section 5. Finally, we make our concluding remarks and discuss future work in Section 6.

2 Preliminaries

Time-triggered runtime verification [3] consists of a monitor and an application program under inspection. The monitor runs in parallel with the application program and interrupts the program execution at regular time intervals to observe the state of the program. The state of the program is determined by evaluating the value of a set of variables being monitored. The key advantage of this technique is *bounded* and *predictable* overhead incurred during program execution. This overhead is inversely proportional to the sampling period at which the monitor samples the program.

Formally, let P be a program and Π be a logical property (e.g., in LTL), where P is expected to satisfy Π. Let $\mathcal{V}_\Pi$ denote the set of variables that participate in Π. In time-triggered runtime verification, a monitor reads the value of variables in $\mathcal{V}_\Pi$ at certain time intervals and evaluates Π. The main challenge in this mechanism is accurate reconstruction of states of P between two consecutive samples; i.e., if the value of a variable in $\mathcal{V}_\Pi$ changes more than once between two consecutive samples, then the monitor may fail to detect violations of Π. Control flow analysis helps us to reconstruct the states of P. To reason about the control-flow of programs at run time, we utilize the notion of *control-flow graphs* (CFG).

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 in-degree 0, which represents the initial basic block of P.*
- *A: is a set of* arcs *(u, v), where $u, v \in V$. An arc (u, v) exists in A, if and only if the execution of basic block u immediately leads to the execution of basic block v.*
- *w: is a function $w : A \to \mathbb{N}$, which defines a* weight *for each arc in A. The weight of an arc is the* best-case execution time *(BCET) of the source basic block.* □

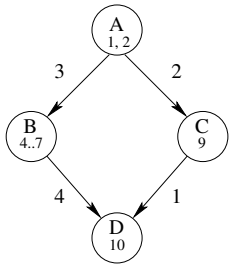
For example, consider the C program in Figure 2(a) (taken from [3]). If each instruction takes one time unit to execute in the best case, then the resulting control-flow graph is the one shown in Figure 2(b). Vertices of the graph in Figure 2(b) are annotated by the corresponding line numbers of the C program in Figure 2(a).

In order to accurately reconstruct program states between two samples, we modify CFG_P in three steps.

Step 1: Identifying Critical Vertices
We ensure that each *critical instruction* (i.e., an instruction that modifies a variable in $\mathcal{V}_\Pi$) is in a basic block that contains no other instructions. We refer to such a basic block as a *critical basic block* or *critical vertex*. For example, in

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

(a) A simple C program



(b) Control-flow graph

Fig. 2. A C program and its control-flow graph.

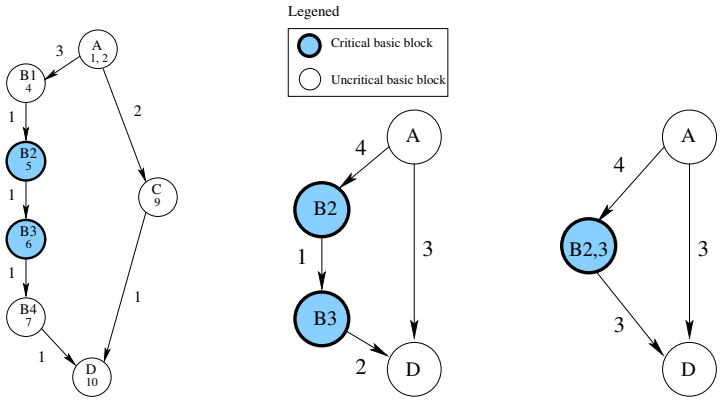
Figure 2(a), if variables `b` and `c` are of interest for verification of a property at run time, then instructions 5 and 6 will be critical and we will obtain the control-flow graph shown in Figure 3(a).

Step 2: Calculating the Minimum Sampling Period
Since uncritical vertices play no role in determining the sampling period, in the second step, we collapse uncritical vertices as follows. Let $CFG = \langle V, v^0, A, w\rangle$ be a control-flow graph. *Transformation* $T(CFG, v)$, where $v \in V \backslash \{v^0\}$ and the out-degree of v is positive, obtains $CFG' = \langle V', v^0, A', w'\rangle$ via the following ordered steps:

1. Let A'' be the set $A \cup \{(u_1, u_2) \mid (u_1, v), (v, u_2) \in A\}$. Observe that if an arc (u_1, u_2) already exists in A, then A'' will contain parallel arcs (such arcs can be distinguished by a simple indexing or renaming scheme). We eliminate the additional arcs in Step 3.
2. For each arc $(u_1, u_2) \in A''$,
$$w'(u_1, u_2) = \begin{cases} w(u_1, u_2) & \text{if } (u_1, u_2) \in A \\ w(u_1, v) + w(v, u_2) & \text{if } (u_1, u_2) \in A'' \backslash A \end{cases}$$
3. If there exist parallel arcs from vertex u_1 to u_2, we will only include the one with minimum weight in A''.
4. Finally, $A' = A'' \backslash \{(u_1, v), (v, u_2) \mid u_1, u_2 \in V\}$ and $V' = V \setminus \{v\}$.

We clarify a special case of the above transformation, where u and v are two uncritical vertices with arcs (u, v) and (v, u) between them. Deleting one of the vertices, say u, results in a self-loop (v, v), which we can safely remove. This is simply because a loop that contains no critical instructions does not affect the sampling period.

We apply the above transformation on all uncritical vertices. We call the result a *critical control-flow graph*. Such a graph includes (1) an uncritical initial basic block, (2) possibly an uncritical vertex with out-degree 0 (if the program is terminating), and (3) a set of critical vertices. Figure 3(b) shows the critical control-flow graph of the graph in Figure 3(a).



(a) Step 1 (b) Step 2 (c) Step 3

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

Definition 2. *Let* $CFG = \langle V, v^0, A, w\rangle$ *be a critical control-flow graph. The* minimum sampling period *for* CFG *is* $MSP_{CFG} = \min\{w(v_1, v_2) \mid (v_1, v_2) \in A \wedge v_1$ *is a critical vertex*$\}$. □

Intuitively, the minimum sampling period is the minimum timespan between two successive changes to any two variables in $\mathcal{V}_\Pi$. For example, the minimum sampling period of the control-flow graph in Figure 3(b) is $MSP = 1$. By applying this sampling period, all property violations can be detected [3].

Step 3: Increasing the Sampling Period using Auxiliary Memory
To increase the sampling period (and, hence, the involvement of the monitor), we use auxiliary memory to build a history of critical state changes between consecutive samples. More specifically, let (u, v) be an arc and v be a critical vertex in a critical control-flow graph CFG, where critical instruction *inst* changes the value of a variable a. We apply transformation $T(CFG, v)$ and append an instruction $inst' : a' \leftarrow a$, where a' is an auxiliary memory location, to the sequence of instructions in vertex u. We call this process *instrumenting transformation* and denote it by $IT(CFG, v)$. Observe that deleting a critical vertex v results in incorporating an additional memory location.

Unlike uncritical vertices, the issue of loops involving critical vertices needs to be handled differently. Suppose u and v are two critical vertices with arcs (u, v) and (v, u) between them and we intend to delete u. This results in a self-loop (v, v), where $w(v, v) = w(u, v) + w(v, u)$. Since we do not know how many times the loop may iterate at run time, it is impossible to determine the upper bound on the size of auxiliary memory needed to collapse vertex v. Hence, to ensure correctness, we do not allow applying the transformation IT on critical vertices that have self-loops.

Given a critical control-flow graph, our goal is to optimize two factors through a set of IT transformations: (1) minimizing auxiliary memory, and (2) max-

imizing sampling period. In [3], we showed that this optimization problem is NP-complete.

3 Heuristics for Optimizing Instrumentation and Auxiliary Memory

In order to tackle the exponential complexity of our optimization problem, in [3], we proposed a mapping from our optimization problem to ILP. This mapping enables us to utilize powerful ILP-solvers to solve our problem. However, the exponential complexity of the problem can still be a stumbling block for large programs.

An interesting observation from the experiments conducted in [3] is that increasing the sampling period even by a factor 100 resulted in at most a 4% increase in total memory usage for tested programs. This observation strongly suggests that for a fixed sampling period, even nearly optimal solutions to the problem (in terms of the size of auxiliary memory) are likely to be quite acceptable. With this intuition, in this section, we propose two polynomial-time heuristics. Both heuristics take a control-flow graph G and a desired sampling period SP as input and return a set U of vertices to be deleted as prescribed by Step 3 (i.e., $IT(CFG, v)$) in Section 2. This set identifies the extra memory locations and the corresponding instrumentation instructions.

3.1 Heuristic 1

Our first heuristic is a simple greedy algorithm (see Heuristic 1):

- First, it prunes the input control-flow graph G (Line 2). That is, it removes all vertices where the weights of all its incoming and outgoing arcs are greater than or equal to SP. Obviously, such vertices need not be deleted from the graph, because they leave the minimal sampling period unaffected.
- Next, it explores G to find the vertex incident to the maximum number of incoming and outgoing arcs whose weights are strictly less than SP (Line 4). Our intuition is that deleting such a vertex results in removing a high number of arcs whose weights are less than the desired sampling period.
- Then, it collapses vertex v identified on Line 4. This operation (Line 5) results in merging incoming arcs to v with outgoing arcs from v in the fashion described in Step 3 in Section 2.
- Obviously, basic block v contains a critical instruction for which we add an auxiliary memory location to build history of this instruction. Thus, we add v to U (Line 6).
- We repeat Lines 3-7 until the minimum arc weight of G is greater than or equal to SP (the while-loop condition in Line 3).
- If the graph cannot be collapsed further (i.e., all vertices are collapsed), then the graph's structure will not permit increasing the sampling period to SP and the algorithm declares failure.

Heuristic 1 Greedy

Input: A critical control-flow graph $G = \langle V, v^0, A, w\rangle$ and desired sampling period SP.

Output: A set U of vertices to be deleted from G.

```
1: U := {};
2: G := PruneCFG(G, SP);

3: while (MW(G) < SP ∧ U ≠ V) do
4:     v := GreedySearch(G);
5:     G := CollapseVertex(G, v);
6:     U := U ∪ {v};
7: end while

8: if (U = V) then declare failure;
9: return U;
```

Heuristic 2 Vertex Cover Based

Input: A critical control-flow graph $G = \langle V, v^0, A, w\rangle$ and desired sampling period SP.

Output: A set U of vertices to be deleted from G.

```
1: U := {};
2: G := PruneCFG(G, SP);

3: while (MW(G) < SP ∧ U ≠ V) do
4:     vc := Approximate-Vertex-
       Cover(G);
5:     for each vertex v ∈ vc do
6:         G := CollapseNode(G, v);
7:         U := U ∪ {v};
8:     end for
9: end while

10: if (U = V) then declare failure;
11: return U;
```

3.2 Heuristic 2

Our second heuristic is an algorithm based on a solution to the *minimum vertex cover* problem: Given a (directed or undirected) graph $G = \langle V, E\rangle$, our goal is to find the minimum set $U \subseteq V$, such that each edge in E is incident to at least one vertex in U. The minimum vertex cover problem is NP-complete, but there exists several approximation algorithms that find nearly optimal solutions (e.g., the 2-approximation in [5]).

Our algorithm (see Heuristic 2) works as follows:

- First, it prunes G (Line 2). That is, it removes all vertices where the weights of all its incoming and outgoing arcs are greater than or equal to SP. Obviously, such vertices can remain in the graph.
- Next, we compute an approximate vertex cover of graph G (Line 4), denoted as vc. Our intuition is that since the graph is pruned and the vertex cover vc covers all arcs of the graph, collapsing all vertices in vc may result in removing all arcs whose weights are strictly less than SP. We note that the approximation algorithm in [5] is a non-deterministic randomized algorithm and may produce different covers for the same input graph. To improve our solution, we run Line 4 multiple times and select the smallest vertex cover. This is abstracted away from the pseudo-code.
- Then, similar to Heuristic 1, we collapse each vertex $v \in vc$ (Lines 5-7). This operation (Lines 5-7) results in merging incoming arcs to v with outgoing

arcs from v in the fashion described in Step 3 in Section 2. Basic block v contains a critical instruction for which we add an auxiliary memory location to build history of this instruction. Thus, we add v to U (Line 7).
- We repeat Lines 3-8 until the minimum arc weights of G are greater than or equal to SP (the while-loop condition in Line 3).
- If the graph cannot be collapsed further (i.e., all vertices are collapsed), then the graph's structure will not permit increasing the sampling period to SP and the algorithm declares failure.

4 Optimization Using a Genetic Algorithm

As another practical approach to the heuristics described in Section 3, we employ a genetic algorithm (GA) to approximate the solution of our optimization problem. In our genetic model, we define a desirable sampling period SP and aim at collapsing a minimum number of vertices in a given critical control-flow graph G, so that we achieve a sampling period of at least SP.

We map our optimization problem to the following genetic model and will describe it in detail in the following subsections:

1. *Chromosomes*: Each chromosome represents the list of vertices in a critical control-flow graph, G. Each vertex in a chromosome is flagged by either the value *true* or *false*. The value *true* represents the condition where the vertex has been chosen to be collapsed in G.
2. *Fitness Function*: The fitness function of a chromosome is the number of collapsed vertices represented by the chromosome.
3. *Reproduction*: To create a new generation of chromosomes, we use both mutation and crossover.
4. *Termination*: The genetic algorithm terminates when a chromosome with the optimal number of collapsed vertices is found, or the upper limit on creating new generations is reached.

4.1 The Chromosomes

Let $G = \langle V, v^0, A, w\rangle$ be a critical control-flow graph. Each chromosome in the genetic model has a static length of $|V|$. Each entry of the chromosome is a tuple $\langle$vertex id, min-SP, value$\rangle$ that represents a vertex in G. *Vertex id* is the vertex identifier, *min-SP* is the minimum weight of the incoming and outgoing arcs of the vertex and *value* indicates whether the vertex is collapsed in G. If $value = true$ for a vertex v, then v is collapsed and we add an auxiliary memory location to build a history of the instruction in v. The sampling period of the control-flow graph resulting from the collapsed vertices identified by the chromosome must always be at least SP. We refer to the sampling period of the resulting control-flow graph as the chromosome's sampling period.

Upon initialization, we create the initial generation. First, We choose the size $|\mathcal{G}|$ (i.e., number of chromosomes) of the generations. Second, we randomly create

$|\mathcal{G}|$ chromosomes for the initial generation. To create a chromosome, we randomly collapse a set of vertices resulting in a control-flow graph with a sampling period of at least SP. Our genetic algorithm executes the following steps to generate such a chromosome:

- First, it finds the set of vertices, SV, in G where *min-SP* is less than SP for each vertex in SV.
- Second, it randomly chooses a vertex $v \in SV$ and collapses v from G and produces a new control-flow graph $G' = T(G, v)$.
- Third, it calculates the sampling period of G'. If the sampling period is less than SP, it returns back to the first step and chooses the next vertex to collapse.

4.2 Selection/Fitness Function

Since we aim at increasing the sampling period to SP with the least number of collapsed vertices, the chromosome is more fit when the number of collapsed vertices in the chromosome is closer to the optimal number of collapsed vertices. Hence, we define the fitness function as: $\mathcal{F} = \mathcal{C}_{chr}$, where $\mathcal{C}_{chr}$ is the number of collapsed vertices in chromosome chr. Consequently, if $\mathcal{F}$ is smaller, then the chromosome will be more fit.

4.3 Reproduction

We use both *mutation* and *crossover* to evolve the current generation into a new generation. First, we use a *one-point* crossover to create new chromosomes for the next generation. The choice of parents is random. In the crossover, we cut the two parents into half and create two children by swapping halves between the parents. We check both children to see if their sampling period is at least SP. If so, the child will be added to the set of chromosomes of the next generation; if not, the child will be passed on to the mutation step.

Second, the mutation process takes the children passed over by the crossover process and processes each child by the following steps:

1. It finds the set of vertices, SV, where *min-SP* is less than SP for each vertex in SV.
2. It randomly chooses a vertex $v \in SV$ to collapse by using $T(G, v)$.
3. It finds the set of collapsed vertices, PV, in the child chromosome for vertices where *min-SP* is larger than SP.
4. It randomly chooses a vertex $u \in PV$ to un-collapse, meaning that u is restored to the control-flow graph represented by the child chromosome.
5. It will check if the minimum sampling period of the new child chromosome is at least SP. If the sampling period is less than SP, it will return to the first step and repeat the steps again, until the sampling period of the child chromosome is at least SP or when it exhausts the limit we set for the number of times a chromosome can be mutated.

6. If a new child chromosome with a sampling period of at least SP is reached at step five, it is added to the next generation.

Sometimes the crossover and mutation processes fails to create $|\mathcal{G}|$ chromosomes to populate the next generation, since fewer than $|\mathcal{G}|$ children satisfy the sampling period restriction for chromosomes. In this case, our genetic algorithm chooses the most fit chromosomes from the current generation and adds them to the next generation to create a population of $|\mathcal{G}|$ chromosomes. In the case that duplicates chromosomes appear in this process, it discards the duplicate and randomly creates new chromosomes as described in Section 4.1.

4.4 Termination

Two conditions can terminate the process of creating a new generation: (1) when we find a chromosome with a sampling period of at least SP and has collapsed the same number of vertices as the optimal solution; (2) when we reach an upper bound on the number of generations. In the second case, we choose from all generations the chromosome with the lowest fitness value $\mathcal{F}$.

5 Experimental Results

In this section, we present the results of our experiments. Our tool chain consists of the following: We generate the control-flow graph of a given C program using the tool CIL [12]. Next, we generate the critical control-flow graph and either transform it into an ILP model using the method in [3] and solve the model using lp_solve [11] or we feed the critical control-flow graph into our heuristics. In either case, we obtain the set of instructions and variables in the program that need to be instrumented using auxiliary memory. We use the breakpoint mechanism of gdb [6] to implement time-triggered monitors. Finally, a Python script controls gdb. Our case studies are from the MiBench [8] benchmark suite. We fix a sampling period of $40 \times MSP$, where MSP is the minimum sampling period of the program (see Definition 2). All experiments in this section are conducted on a personal computer with a 2.26 GHz Intel Core 2 Duo processor and 6 GB of main memory.

5.1 Performance of Heuristics

Table 1 compares the performance of the ILP-based solution [3] with the heuristics presented in Section 3 and the genetic algorithm proposed in Section 4 for different programs from MiBench. The first column shows the size of the critical control-flow graph of programs in terms of the number of vertices. With each approach, we record the time spent to solve the optimization problem (in seconds) and the suboptimal factor (SOF). SOF is defined as $\frac{sol}{opt}$, where sol and opt are the number of vertices requiring instrumentation returned by a heuristic and the ILP-based solution (i.e., the optimal solution), respectively.

	CFG	*ILP*		*Heuristic 1 (Greedy)*		*Heuristic 2 (VC)*		*Genetic Algorithm*	
	Size($\|V\|$)	**time (s)**	**SOF**	**time (s)**	**SOF**	**time (s)**	**SOF**	**time (s)**	**SOF**
Blowfish	177	5316	–	0.0363	7.8	0.8875	8	383	2.5
CRC	13	0.35	–	0.0002	3.5	0.0852	3	0.254	1.5
Dijkstra	48	1808	–	0.0064	1.2	0.1400	1.2	116	1.7
FFT	47	269	–	0.0042	1.7	0.1737	1.8	74	1.1
Patricia	49	2084	–	0.0054	1.4	0.1369	1.6	140	1.5
Rijndael	70	3096	–	0.0060	1.6	0.2557	2.1	370	1.9
SHA	40	124	–	0.0039	2.2	0.1545	2.2	46	1.3
Susan	20 259	∞	–	3 181	N/A	26 211	N/A	923	N/A

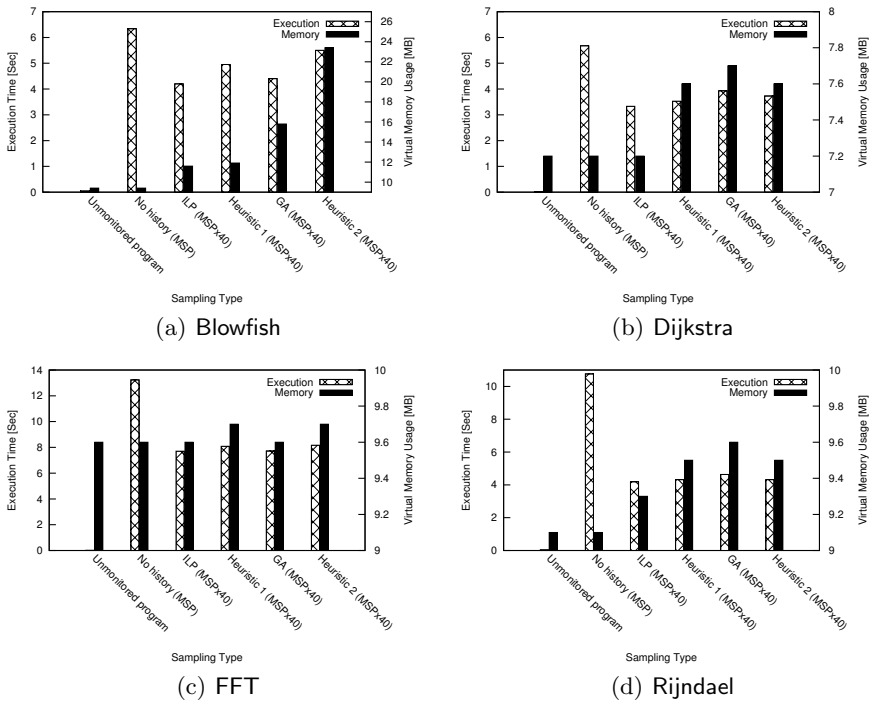
Table 1. Performance of different optimization techniques.

Clearly from Table 1, all three heuristic algorithms perform substantially faster than solving the exact ILP problem. On average, Heuristic 1, Heuristic 2, and the genetic algorithm yield in speedups of 200 000, 7 000, and 9, respectively, where the speedup is defined as the ratio between the execution time required to solve the ILP problem and the time required to generate an approximate solution using one of the heuristics. The execution times of Heuristic 2 are based on running Approximate-Vertex-Cover 500 times to cope with the randomized vertex cover algorithm (see Line 4 in Heuristic 2). Table 1 shows that for large programs, such as Susan, solving for the optimal solution becomes infeasible because of the problem's intractability. However, we see that all three heuristics are able to generate some approximate solution that can be used to instrument the program for time-triggered runtime verification.

In general, the genetic algorithm produces results that are closer to the optimal solution than Heuristics 1 and 2. The spread of the SOFs for the conducted experiments is small for the genetic algorithm. For the conducted experiments, the worst SOF for the genetic algorithm is 2.5 (i.e., for Blowfish), which indicates that this solution will collapse 2.5 times more vertices in the critical control-flow graph than the optimal solution. With the exception of Blowfish, Heuristics 1 and 2 also perform well, where the SOF ranges from 1.2 to 3.5. We cannot conclude that the performance of Heuristics 1 and 2 suffers as the size of the problem increases because for Susan, Heuristics 1 and 2 indicate that SP may be satisfied by collapsing 104 and 180 vertices, respectively, while the genetic algorithm produces a solution where 222 vertices must be collapsed. The SOFs for Dijkstra also indicate an anomaly in the overall trend. Therefore, the performance of the heuristics likely depends on the structure of the critical control-flow graph. For Susan, the number of vertices being collapsed is approximately 0.5% to 1% of $|V|$, which indicates that the instrumentation overhead should be small.

5.2 Analysis of Instrumentation Overhead

We also collected the execution times and memory usage of the instrumented benchmark programs during experimentation. Figure 4 shows the execution times and memory usage of four of the eight benchmark programs (for reasons of



(a) Blowfish (b) Dijkstra

(c) FFT (d) Rijndael

Fig. 4. The impact of different instrumentation schemes on memory usage and total execution time.

space) we used for our experiments. Each plot in Figure 4 contains the execution times and memory usage for the unmonitored program, the program monitored with a sampling period of MSP, and the program monitored at $40 \times MSP$ with the inserted instrumentation points indicated by the optimal and heuristic solutions. The benchmark program results not shown in Figure 4 exhibit similar trends as Figure 4(c).

Based on Figure 4, we observe that instrumented benchmark programs with no history always run slower than the programs instrumented with $SP = 40 \times MSP$. This is expected because the external monitor requires more processing resources when it samples at higher frequencies.

We also observe that the variation of the execution times of programs instrumented based on the optimal and heuristic solutions (i.e., ILP, Heuristics 1 and 2, GA) are negligible. Therefore, using suboptimal instrumentation schemes do not greatly affect the execution time of the program as compared to the execution time of optimally instrumented program.

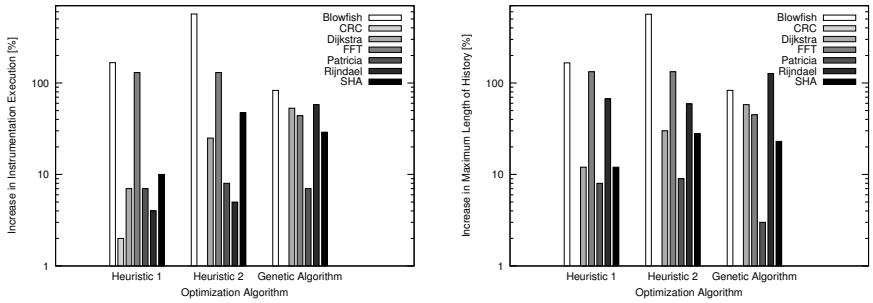
From Figure 4, we observe that utilizing the instrumentation schemes returned by solving the ILP or running the heuristics result in an increase in the memory usage during program execution. This is expected because to increase the sampling period of the monitor, some program state history must be retained

to ensure that the program can be properly verified at run time. With the exception of Blowfish, the memory usage increase is negligible for the benchmark programs.

Using the instrumentation schemes generated by the heuristics, the increase in memory usage is negligible during program execution with respect to the optimally instrumented program, except for Blowfish. The variation of memory usage for all benchmark programs except for Blowfish generally spans from 0 MB to 0.1 MB. Even though the memory usage of Blowfish instrumented with the schemes produced by Heuristic 2 and the genetic algorithm is relatively larger than the optimal scheme, an increase of 15 MB of virtual memory is still negligible to the amount of memory that is generally available on the machine used to verify a program. Of the three heuristics, we cannot conclude which heuristic generally produces the best instrumentation scheme, as each heuristic behaves differently for different types of control-flow graphs.

Figure 5 shows the percentage increase in the number of instrumentation instructions executed and the percentage increase in the maximum size of history between two consecutive samples with respect to the optimally instrumented benchmark programs. Note that logarithmic scales are used in the charts in Figure 5. Observe that Susan is not shown in the figure, because solving for the optimal solution is infeasible. Blowfish performed the poorest with respect to the two measures when the instrumentation schemes generated by Heuristics 1 and 2 were used. In most cases, the percentage increase in the number of instrumentation instructions that are executed and the maximize size of history are below 50% if we remove the two largest percentages from each set. If we ignore a few more outliers, then most of the percentage increases for both measures will be below 20%. We also observe that the percentage increase in the number of instrumentation instructions executed is proportional to the increase in the maximum size of the history between two consecutive samples. This implies that the extra instrumentation instructions (as compared to the optimal solution) are evenly distributed among sampling points.

Recall that the collapsed vertices during the transformation IT (see Section 2) determine the instrumentation instructions added to the program under inspection. These instructions in turn store changes in critical variables to the history. Although one may argue that auxiliary memory usage at run time must be in direct relationship with the number of collapsed vertices (i.e., instrumentation instructions), this is not necessarily true. This is because the number of added instrumentation instructions differs in different execution paths. For example, one execution path may include no instrumentation instruction and another path may include all such instructions. In this case, the first path will build no history and the second will consume the maximum possible auxiliary memory. This observation also holds in our analysis on other types of overheads as well as the total execution time. This is why in Table 1, the genetic algorithm does the best job of optimizing the Blowfish benchmark for the fewest critical instructions, but in Figure 4(a), the benchmark uses substantially more memory than the greedy heuristic. This is also why in Figure 5, the amount of auxil-



(a) Increase in the number of execution of instrumentation instructions.

(b) Increase in the maximum size of history between two samples.

Fig. 5. The impact of sub-optimal solutions on execution of instructions to build history and its maximum size.

iary memory used by a monitored program is not proportional to the number of instrumented critical instructions.

> *We conclude from our experiments that the NP-completeness of the optimization problem is not an obstacle when applying time-triggered runtime verification in practice.*

6 Conclusion

In this paper, we proposed three efficient algorithms to address the NP-complete problem of optimizing the instrumentation of programs in the context of time-triggered runtime verification [3]. This instrumentation is needed for constructing history to record events between two consecutive samples at run time. Our algorithms are inspired by different techniques, such as greedy heuristics, finding the minimum vertex cover, and biological evolution. We rigorously benchmarked eight different programs using our algorithms. The results show that the solutions returned by all three algorithms lead to negligible increase in instrumentation overhead and total memory usage at run time as well as the total execution time of monitored program. Moreover, we found our genetic algorithm more efficient and robust than the other two. In summary, we conclude from our experiments that the NP-completeness of the optimization problem is not an obstacle when applying time-triggered runtime verification in practice.

In the future, we plan to develop more sophisticated heuristics that specifically aim at distributing instrumentation instructions between sampling points evenly. We are also working on other polynomial-time techniques, such as ILP relaxation, for solving the instrumentation optimization problem. Other research directions include developing adaptive methods for overhead control (e.g., by incorporating the method in [10]), where the monitor adapts its sampling period based upon the structure of the input program.

7 Acknowledgement

This research was supported in part by NSERC DG 357121-2008, ORF RE03-045, ORE RE04-036, ORF-RE04-039, ISOP IS09-06-037, APCPJ 386797-09, and CFI 20314 with CMC.

References

1. A. Bauer, M. Leucker, and C. Schallhart. Runtime Verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2009. in press.
2. A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL Semantics for Runtime Verification. *Journal of Logic and Computation*, 20(3):651–674, 2010.
3. B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Sampling-based runtime verification. In *Formal Methods (FM)*, 2011. To appear.
4. S. Colin and L. Mariani. *Run-Time Verification*, chapter 18. Springer-Verlag LNCS 3472, 2005.
5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
6. GNU debugger. `http://www.gnu.org/software/gdb/`.
7. D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Automated Software Engineering (ASE)*, pages 412–416, 2001.
8. M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE International Workshop on In Workload Characterization (WWC)*, pages 3–14, 2001.
9. K. Havelund and A. Goldberg. Verify your Runs. pages 374–383, 2008.
10. X. Huang, J. Seyster, S. Callanan, K. Dixit, R. Grosu, S. A. Smolka, S. D. Stoller, and E. Zadok. Software monitoring with controllable overhead. *Software tools for technology transfer (STTT)*, 2011. To appear.
11. ILP solver lp_solve. `http://lpsolve.sourceforge.net/5.5/`.
12. G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. Proceedings of Conference on Compilier Construction, 2002.
13. A. Pnueli and A. Zaks. PSL Model Checking and Run-Time Verification via Testers. In *Symposium on Formal Methods (FM)*, pages 573–586, 2006.