Runtime Monitoring of Time-sensitive Systems
[Tutorial Supplement]

Borzoo Bonakdarpour and Sebastian Fischmeister

University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada, N2L 3G1
Email:borzoo@cs.uwaterloo.ca and sfischme@uwaterloo.ca

Abstract. This tutorial focuses on issues involved in runtime monitoring of time-sensitive systems, where violation of timing constraints are undesired. Our goal is to describe the challenges in instrumenting, measuring, and monitoring such systems and present our solutions developed in the past few years to deal with these challenges. The tutorial consists of two parts. First, we present challenge problems and corresponding solutions on instrumenting real-time systems so that timing constraints of the system are respected. The second part of the tutorial will focus on *time-triggered* runtime monitoring, where a monitor is invoked at equal time intervals, allowing designers to schedule regular and monitoring tasks hand-in-hand.

1 Overview of Tutorial

In computing systems, *correctness* refers to the assertion that a system satisfies its specification. *Verification* is a technique for checking such an assertion and *runtime verification* refers to a lightweight technique where a *monitor* checks at run time whether the execution of a system under inspection satisfies or violates a given correctness property. Deploying runtime verification involves instrumenting the program under inspection, so that upon occurrence of events (e.g., value changes of a variable) that may change the truthfulness of a property, the monitor will be 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.

The above defects are undesired in the context of real-time embedded systems, where predictability and timing constraints play a central role. This tutorial focuses on describing our solutions to two challenge problems:

- **Time-aware instrumentation.** Instrumentation is a technique to extract information or trigger events in programs under inspection. Instrumentation is a vital step for enabling system monitoring; i.e. the system is augmented with instructions that invokes a monitor when certain events occur. Instrumentation of software programs while preserving logical correctness is an

established field. However, current approaches are inadequate for real-time embedded applications. The key idea behind the *time-aware instrumentation* of a system is to transform the execution-time distribution of the system so as to maximize the coverage of the trace while always staying within the time budget.

– **Time-triggered monitoring.** In *time-triggered* runtime verification, 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. Providing such guarantee results in solving an 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. Consequently, the time-triggered monitor can successfully reconstruct the state of the program between two successive samples.

This tutorial will discuss in detail our techniques developed in the past few years while exploring time-aware instrumentation and time-triggered monitoring. In Section 2, we introduce the problem of timing in real-time embedded systems. Section 3 is dedicated to our techniques on time-aware instrumentation. Time-triggered runtime monitoring is discussed in Section 4. Finally, we present a set of future research directions in Section 5.

2 Real-time Embedded Software Primer

Embedded software is the essence of our modern computerized standard of living. It is omnipresent and controls everything from everyday consumer products such as microwaves and digital cameras to large systems for factory automation, aircraft, and automotive applications. Software enables devices that make life more pleasant (i.e., adaptive cruise control in cars), more acceptable (i.e., implanted insulin infusion pumps), or even possible (i.e., implanted pacemakers and defibrillators). Many companies define innovation in their products through adding new features implemented in software, thus future systems will contain more complex and larger portions of software. For example, the next generation automobile is a highly distributed system expected to run several million lines of code [3].

A special class of embedded software is real-time embedded software. A real-time application is time sensitive and this generally means that delivering a correct value at the wrong point in time—especially too late—can still cause service failures. Thus a real-time system must work correctly in the logic and timing domain. Examples of such system span almost all domains of embedded software including consumer devices (e.g., video decoders), avionics platforms (e.g., flight control in autopilots), automotive (e.g., distance measurement in adaptive cruise control), medical devices (e.g., pacemakers), finance (e.g., high-frequency trading), communications (e.g., software-defined radio), and space (e.g., thrust control).

Table 1 shows the relationship between timing, logic, and correctness of a real-time application for two basic classes. Naturally, more classes have been defined over time. These additional classes refine and extend some of the properties such as firm real-time constraints [13] and imprecise computation [14]. Soft real-time applications have soft timing constraints. This means that if the application sometimes misses deadlines, the application will still function. A typical example of such an application is video decoding; dropping a frame sometimes will remain unnoticed by the viewer, however, frequent drops in frames will degrade the experience. The threshold for an acceptable number of missed deadlines depends on the application. Hard real-time applications have hard timing constraints. Missing a single deadline can result in an error in the system. Typical applications for this domain are safety-critical systems like a shutdown routine in a nuclear power station, flight surface control while piloting airplanes, and pacing control in a heart pacemaker. Obviously, such hard real-time systems require meticulous control of system resources and execution to guarantee proper system functioning and ultimately system safety. Hence, such control is also the main focus of research on real-time systems.

	Soft RT		**Hard RT**	
	On time	**Too late**	**On Time**	**Too late**
Wrong value	Error	Error	Error	Error
Right value	Ok	Maybe ok	Ok	Error

Table 1. Real-time system classification.

Reduction in complexity through limiting the programming languages is one approach to provide better control of resources, execution, and timing. With this goal in mind, several standards on developing safety-critical and real-time embedded systems have emerged over the years. They find use in different domains. For example MISRA C [12] provides coding guidelines and reduces the complexity of C code by forbidding, for instance, recursion, unbounded loops, and dynamic memory allocation. The automotive and other industries use MISRA C. Ravenscar [5] and SPARK [2] address similar issues for the Ada programming language. The RTCA/DO 178B [16] specifies guidelines for developing safety-critical software systems for the avionics domain. The standard touches major topics of the software development cycle and specifies required documentation for different activities. Other domains, like the nuclear domain, have similar standards. Such standards are relevant as they define classes of systems to which solutions can be tailored to. For example, while static analysis is impractical in the general case, the limited use of pointers in MISRA C compliant code permits subjecting such programs to static analysis.

Another popular approach to handling resources, execution, and timing is to follow a time-triggered approach. In these approaches, time is split into small

slices. The system scheduler assigns resource users mutually exclusive access to the resource based on these slices. For example, the time-triggered approach for task scheduling is round robin scheduling. The scheduler assigns one slice of processor time to one process at a time. In communication, Time Division Multiple Access (TDMA) implements a time-triggered approach to limit concurrent access to the shared communication medium. In safety-critical applications, sometimes the developer creates the time-triggered schedule by laying out the time line and determining when which process gets to compute and communicate, so all operations meet their timing deadlines. One advantage of the time-triggered approach is its determinism and thus predictable operation. Time-triggered approaches make operational decisions solely based on a clock. Controlling the clock means controlling all aspects in the system that get derived from that clock. This single source for controlling operations is attractive, because it reduces operational dependencies and thus reduces complexity. Naturally, event-triggered approaches also offer benefits and picking one over the other is a complicated matter and a lasting debate [10].

3 Time-aware Instrumentation

Instrumentation of software programs while preserving logical correctness is an established field. Developers instrument programs for tasks including profiling, testing, debugging, tracing, and monitoring the software systems (e.g., for runtime verification). Today several approaches to instrument software while preserving logical correctness exist and in the tutorial, we will briefly discuss the most relevant works including manual instrumentation, static instrumentation frameworks, dynamic instrumentation with binary rewriting, and hardware-based approaches. However, current approaches are inadequate for real-time embedded applications.

The key idea behind the *time-aware instrumentation* of a system is to transform the execution-time distribution of the system so as to maximize the coverage of the trace while always staying within the time budget. Our notion of coverage implies that the instrumentation will provide useful data over longer periods of tracing. A time-aware instrumentation injects code, potentially extending the execution time on all paths, while ensuring that no path takes longer than the specified time budget.

The time budget is the worst-case execution time of a function without violating a specification. In hard real-time systems, the time budget can be the longest execution time without missing any deadline, and depending on the longest execution time of the non-instrumented version, more or less time will be available for the instrumentation. In systems without deadlines, the time budget can be the current maximum execution time plus a specified non-zero maximum overhead for tracing to the current maximum execution time.

Figure 1 shows the expected consequences of time-aware instrumentation in a hard real-time application on the execution time profile (i.e., the probability density function on the execution time). The x-axis specifies the execution time of

the function, while the y-axis indicates the frequency of the particular execution time. The original uninstrumented code has some arbitrary density function. We have chosen the Gaussian distribution for this example for illustrative purposes; Li et al. provide details from empirical observations of distribution functions [11]. The distribution for the instrumented version differs from the original one. It is shifted towards the right, but still never passes the deadline. This shift occurs because time-aware instrumentation adds to paths, increasing their running times, but ensures that execution times never exceed the deadline.

Note that our execution-time model concentrates on the overhead involved in acquiring data. A related problem is to transport the collected data from the embedded system to an external analysis unit. While that problem admits many solutions, one common solution is to piggyback the buffer information onto serial or network communication.

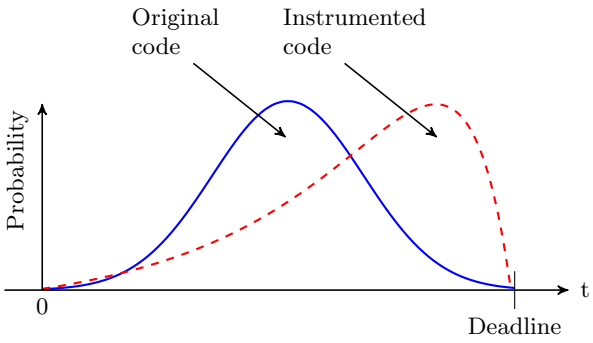


Fig. 1. Execution-time distribution for a code block before and after time-aware instrumentation showing the shift in the expected execution time.

The implementation challenge is to efficiently realize this right shift in the execution time profile without exceeding the deadline. This requires answers to several questions: where to instrument, what to instrument, and how to instrument? We worked on both, a software- and a hardware-based solution.

3.1 Basic Overview [7, 8]

We propose the following instrumentation stages:

– **Source analysis:** The source-code analyzer breaks the functions into basic blocks and generates a call graph. The analyzer also presents a list of variables which are assigned in these basic blocks and the developer can choose a subset of these variables to trace. For hard real-time applications, the analyzer annotates the call graph using execution time information obtained through static analysis or measurements [17].

– **Naive instrumentation:** Using the control-flow graph, the execution times of the basic blocks, and the input variables for the trace, we inject code into the selected function at all instrumentation points.
– **Enforce time budget:** If the naive instrumentation exceeds the time budget, we use an optimization technique to compute an instrumentation which does respect the time budget while maximizing the coverage of the instrumentation.
– **Minimize code size:** If the instrumentation is reliable enough, then we apply semantics-preserving, decreasing transformations to reduce the size of the instrumented code.
– **Collect traces:** The developer finally recompiles and executes the instrumented program.

Figure 2 shows the workflow that results from the steps. To instrument a function, we start by picking the function of interest. We then use the assembly analyzer to extract the control flow graph and break the function into execution paths. In the first phase, we use a tool to instrument all variables of interest and then check whether the execution time on the worst-case path has changed. If it has changed, then we will use integer linear programming to lower the coverage of the instrumentation so that it meets the timing requirements. If the coverage is too low, then we can either give up, if we cannot extend the time budget available for the function and the instrumentation; or extend the time budget, which will allow for higher-coverage instrumentations. If the optimized instrumentation meets the required coverage, or if the initial naive instrumentation does not extend the worst-case path, then we will proceed and use the identified execution paths to minimize the required code size. Afterwards, we can recompile the program and collect the desired traces from the instrumentation.

3.2 Case Study: Flash File System

We investigated an implementation of a wear-levelling FAT-like filesystem for flash devices [4]. The code was originally written by Hein de Kock for 8051 processors. We slightly modified the original implementation so that it would compile with `sdcc`; in particular, we needed to modify the header files to get the code to compile. The implementation consists of about 3000 non-blank, non-comment lines of C code. We ran our tool on 30 functions from the `fs.c` file, dropping some uninteresting functions with mostly straight-line control-flow. Of the 30 functions, 4 functions had more than 100 basic blocks, and `fclose` had 200 basic blocks. For this case study, we also assume that the time budget is the execution time of the longest running path in the function and no interrupts.

Measurements Figure 3 compares density functions for four procedures in the filesystem implementation, both before and after instrumentation. The solid blue line represents the density function of the original procedures, while the dashed red line represents the density function for the instrumented versions. Each of

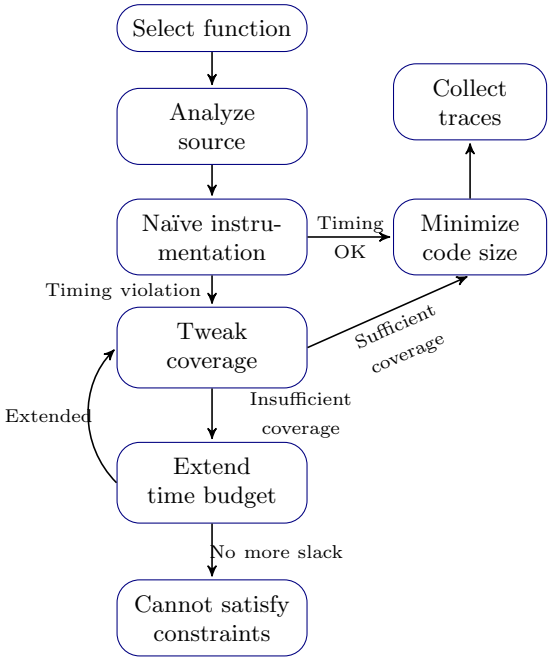


Fig. 2. Workflow of applying time-aware instrumentation.

this figures clearly shows that the original idea underlying our method of time-aware instrumentation, as outlined in Figure 1, works well.

The procedure *fsetpos* shown in Figure 3(b) exhibits the biggest difference between instrumented and non-instrumented versions. The reason is that although this procedure contains many assignments spread across different paths, most assignments do not lie on the worst-case path. The instrumentation engine can therefore capture assignments along these non-critical paths, raising their execution time and putting them closer to the execution time of the worst-case path. Since the engine can capture assignments on many paths, the density function of the execution time for the instrumented version shows a large increase on the right part of the figure, along with a steep decrease on the left part of the figure.

The procedure *rename* shown in Figure 3(d) demonstrates that sometimes the developer might want to add time to the budget for instrumenting to enable the instrumentation of the worst-case path. Figure 4 shows that even with a small increase in the time budget, the coverage can increase significantly. Figure 3 shows the function *fputs* without any additional increase in the time budget, Figure 4(a) shows the function with an extra budget of three assignments, and Figure 4(c) shows the function with an extra budget of 15 assignments. Figure 4(d) summarizes how instrumenting *fputs* improves as we add more time to the time budget for the instrumentation.

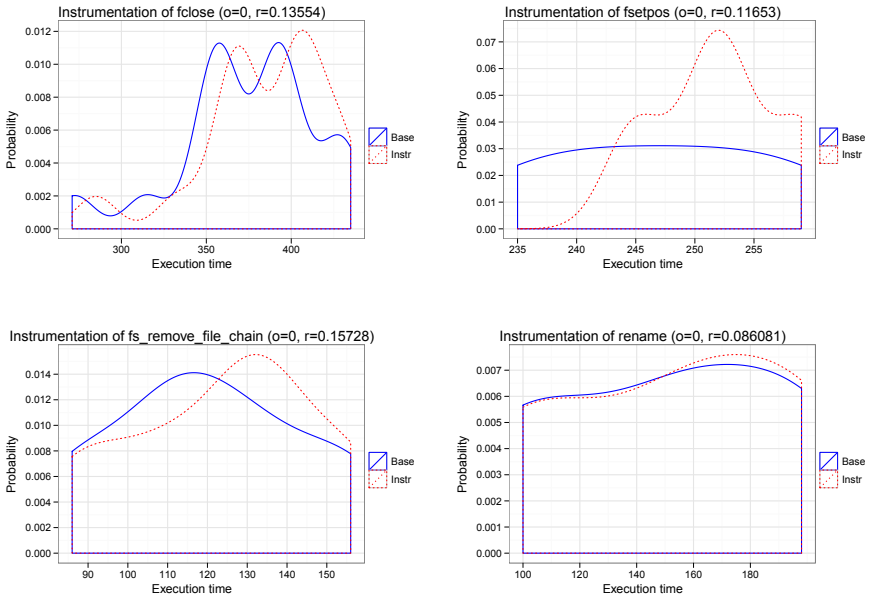


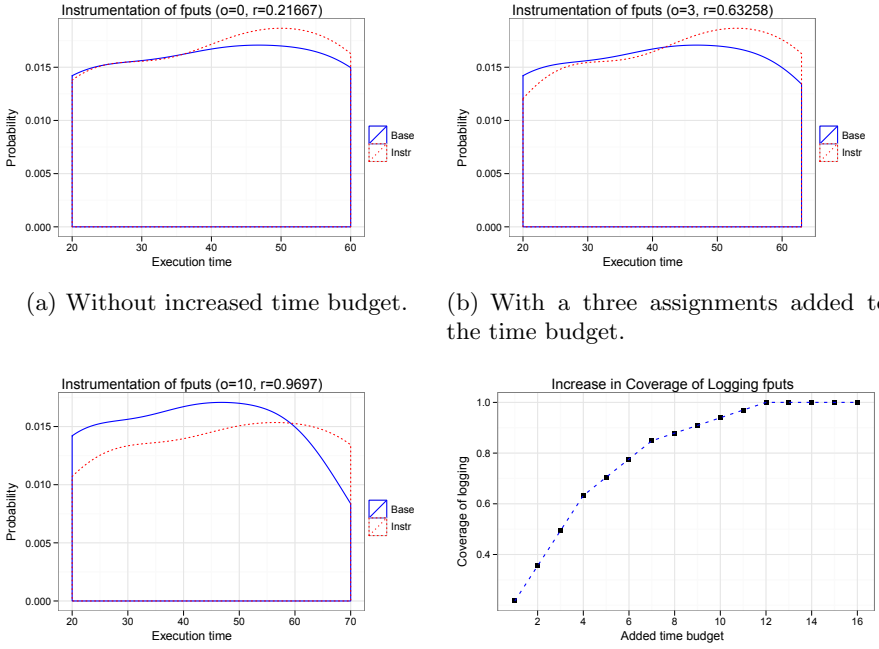
Fig. 3. Examples in instrumenting functions in the filesystem implementation.

4 Time-triggered Runtime Monitoring

Most monitoring approaches in runtime verification are *event-triggered*, where the occurrence of every new event (e.g., change of value of a variable) invokes the monitor. This constant invocation of the monitor leads to *unpredictable overhead* and *bursts* of new events at run time. These defects can cause serious issues at run time especially in embedded safety/mission-critical systems. *Time-triggered* monitoring aims at tackling these drawbacks. Specifically, a time-triggered monitor runs in parallel with the program and samples the program state periodically to evaluate a set of properties.

The second part of the tutorial will focus on two methods: time-triggered path monitoring [6] and time-triggered runtime verification [1]. In both methods, the monitor has to execute at the speed of shortest best-case execution time of branching statements. This ensures that the monitor does not overlook any property violations and can reconstruct the execution path at each sampling point. However, executing the monitor at the speed of best-case execution time results in high involvement of the monitor in execution of the system under inspection.

In this section, we review two techniques for sampling-based execution monitoring [6] and runtime verification [1] in Subsections 4.1 and 4.2, respectively. Both methods employ the notion of *control-flow graphs* (CFG) in order to reason about program execution and its timing characteristics.



(a) Without increased time budget.

(b) With a three assignments added to the time budget.

(c) With a 15 assignments added to the time budget.

(d) Increase in coverage.

Fig. 4. Examples of increasing the coverage by increasing the time budget.

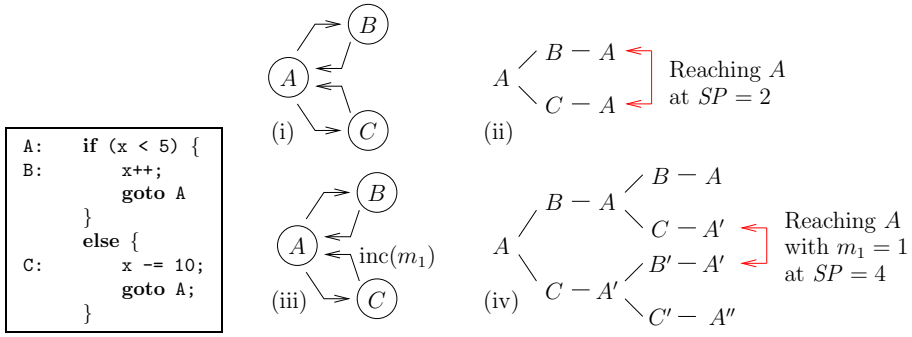
Definition 1. *The* control-flow graph *of a program P is a weighted directed simple graph $CFG_P = \langle V, v^0, A, w \rangle$, where:*

- *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, Figure 5(a) shows a simple C program with three basic blocks labeled A, B, and C and Figure 5(b)(i) shows the resulting control-flow graph.

4.1 Sampling-based Execution Monitoring [6]

In execution monitoring, the objective is to take periodic samples such that the monitor can re-construct execution paths. To this end, the monitor has to execute



(a) A simple C program.

(b) Example of a single instrumentation to extend SP.

Fig. 5. Sampling-based monitoring.

at the speed of shortest best-case execution time of branching statements. For example, in Figure 5(b)(i) the monitor needs to execute at the speed of shortest best-case execution time of $A+B$ or $A+C$; otherwise, the re-construction of the execution path will not be possible. Figure 5(b)(ii) shows the timing diagram for the example. It demonstrates that, assuming all basic blocks take an execution time of 1 time unit, after two time units, it will be impossible to decide whether the program took the path $A \to B \to A$ or $A \to C \to A$. Thereby the sampling period for the program needs to be $SP = 2$.

To increase the sampling period and, hence, decrease the involvement of the monitor, we introduce *markers* to the program. A marker is a simple variable that can be manipulated in basic blocks to distinguish different paths and, hence, resulting in a larger sampling period. In our example, we introduce marker m_1 and instrument vertex C (see Figure 5(b)(iii)). Vertex C manipulates the value of marker m_1 by incrementing it. Thus, the monitor can re-store the basic block *id* (vertex A, B, or C), the current value of m_1, and a time stamp. The timing diagram in Figure 5(b)(iv) shows that introducing the marker increases the sampling period to $SP = 4$, because only after five time units will the program have two or more paths with the same number of increments of m_1 and the same basic block *id*s.

4.2 Sampling-based Runtime Verification [1]

Let P be a program and Π be a logical property (e.g., in LTL), where P is expected to satisfy Π. Let $\mathcal{V}_\Pi$ denote the set of variables that participate in Π. In our idea of sampling-based runtime verification, the monitor reads the value of variables in $\mathcal{V}_\Pi$ and evaluates Π. The main challenge in this mechanism is accurate re-construction of the state of P between two samples; i.e., if the value of a variable in $\mathcal{V}_\Pi$ changes more than once between two samples, the monitor may fail to detect violations of Π. For instance, in the program of

Figure 5(a), if we are to verify the property $\Pi \equiv -5 \leq x \leq 5$, then the monitor requires a fresh value of variable `x` without overlooking any changes. Thus the sampling period for the program needs to be $SP = 2$. Notice that although there are similarities, execution monitoring and runtime verification focus on different issues: the former concentrates on execution paths and the latter on state variable changes.

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

The above example shows how one can take advantage of memory to increase the sampling period of a time-triggered monitor and, hence, impose less overheard on the system. However, there is a tradeoff between the amount of auxiliary memory the system uses at run time and the sampling period. Ideally, we want to maximize the sampling period and minimize the number of history variables. In [1], we showed that this optimization problem is NP-complete.

There are two general approaches to tackle the exponential complexity: (1) mapping our problem to an existing NP-complete problem for which powerful solvers exist (e.g., the Boolean satisfiability problem and integer linear programming), and (2) devising efficient heuristics. The first approach (explored in [1]) involves transforming our optimization problem to integer linear programming (ILP). We now discuss the results of our experiments using this approach. Consider the Blowfish benchmark from the MiBench [9] benchmark suite. This program has 745 lines of code, which results in a CFG of 169 vertices and 213 arcs. We take 20 variables for monitoring. We consider the following different settings for our experiments:

- **Event-based:** gdb extracts the new value of variables of interest whenever they get changed throughout the program execution.
- **Time-triggered with no history:** gdb is invoked every *MSP* time units to extract the value of all the variables of interest.
- **Sampling-based with history:** This setting incorporates our ILP optimization. Thus, whenever gdb is invoked, it extracts the value of variables of interest as well as the history.

In the event-based setting (see Figure 6), since the monitor interrupts the program execution irregularly, unequal bursts in the overhead can be seen. Moreover, the overhead caused by each data extraction is proportional to the data type. Hence, the data extraction overhead varies considerably from one interruption to another. Thus, the monitor introduces probe-effects, which in turn may create unpredictable and even incorrect behaviour. This anomaly is, in particular, unacceptable for real-time embedded and mission-critical systems.

On the contrary, since the time-triggered monitor interrupts the program execution on a regular basis, the overhead introduced by data extraction is not subject to any bursts and, hence, remains consistent and bounded (see Figure 6). Consequently, the monitored program exhibits a predictable behaviour. Obviously, the time-triggered monitor may potentially increase the overhead, which extends the overall execution time. Nonetheless, in many commonly considered applications, designers prefer predictability at the cost of larger overhead.

Figure 6 show the results of our experiments for sampling period of $50 * MSP$. As can be seen, increasing the sampling period results in larger overhead. This is because the monitor needs to read a larger amount of data formed by the history. However, the increase in overhead is considerably small (less than twice the original overhead). Having said that, the other side of the coin is that by increasing the sampling period, the program is subject to less monitoring interrupts. This results in significant decrease in the overall execution time of the programs. This is indeed advantageous for monitoring hard real-time programs. Although adding history causes variability in data extraction overhead, the system behavior is still highly predictable as compared to the event-based setting. The above observations are valid for the case, where we increase the sampling period by $100 * MSP$ as well (see Figures 7).

The tradeoff between execution time and the added memory consumption when the sampling period is increased is shown in Figure 8. As can be seen, as we increase the sampling period, the system requires negligible extra memory. Also, one can clearly observe the proportion of increase in memory usage versus the reduction in the execution time. In other words, by employing small amount of auxiliary memory, one can achieve considerable speedups.

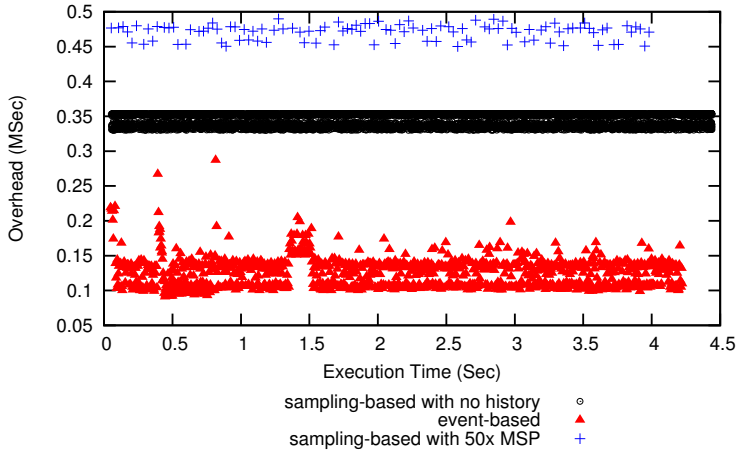


Fig. 6. Experimental results for Blowfish ($50 * MSP$ sampling period).

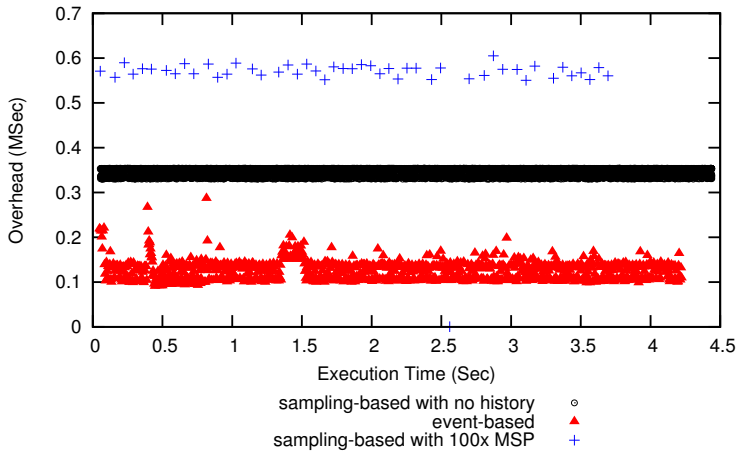


Fig. 7. Experimental results for Blowfish ($100 * MSP$ sampling period).

Although the ILP-based approach always finds the optimal solution to our problem and one can use state-of-the-art ILP-solvers, it cannot deal with huge programs due to the worst-case exponential complexity. For such cases alternative approaches that find near-optimal solutions are proposed in [15].

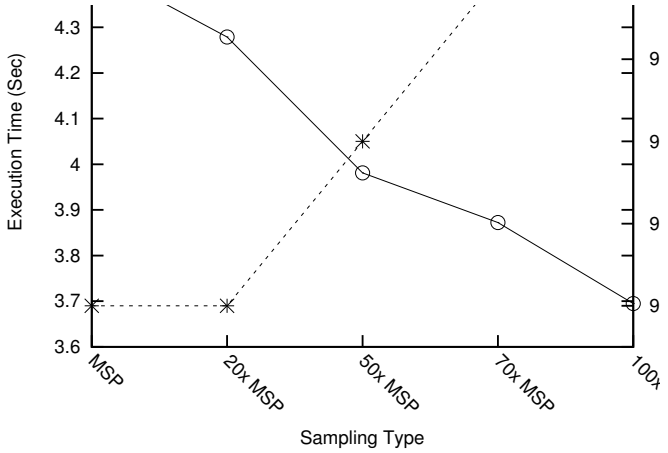


Fig. 8. Memory usage vs. execution time Blowfish.

5 Open Problems

We believe our work on instrumentation and runtime verification of real-time systems has paved the way for numerous future research directions. Interesting open problems include the following:

- **Multicore monitors.** Since in time-triggered runtime verification, the monitor reads a sequence of events in a batch, it can dispatch the events to parallel monitors working on different cores. In particular, our system setting is such that the CPU runs the program under inspection and the GPU runs monitoring parallel tasks. This setting encounters challenging problems, as buffered events may be causally related, making evaluations of temporal properties a difficult task.
- **Monitoring distributed real-time systems.** Implementing distributed real-time systems has always been a challenge for obvious reasons such as clock drifts. Deploying time-triggered monitors involves several research challenges such as developing techniques for precise state reconstruction in a distributed fashion.
- **Overhead minimization.** As discussed in Section 4, although our approach results in a obtaining a predictable, its overall overhead is higher than event-triggered approaches. We need breakthroughs to reduce the overhead of time-triggered monitors. One approach is to develop efficient heuristics that find nearly optimal solutions to the optimization problem proposed in [1].
- **Applicability to broader classes of systems.** The current work makes some assumptions that hold only in specific classes of systems (e.g., MISRA C compliant programs). A challenging problem is to find ways how to eliminate some of the assumptions that tie the approach to particular classes and thus make the approach applicable for new domains. This also applies when staying within the domain of real-time systems, as for example mixed-criticality systems offer interesting applications that need runtime monitoring.

6 Acknowledgement

The research leading to this tutorial 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. B. Bonakdarpour, S. Navabpour, and S. Fischmeister. Sampling-based runtime verification. In *Formal Methods (FM)*, pages 88–102, 2011.
2. B. Carré and J. Garnsworthy. SPARK—an annotated Ada subset for safety-critical programming. In *Proceedings of the Conference on TRI-ADA*, pages 392–402, New York, NY, USA, 1990. ACM.

3. R. N. Charette. This Car Runs on Code. *IEEE Spectrum*, 2009.
4. H. de Kock. small-ffs. `http://http://code.google.com/p/small-ffs`, September 2009.
5. B. Dobbing and A. Burns. The Ravenscar Tasking Profile for High Integrity Real-time Programs. In *Proceedings of the 1998 annual ACM SIGAda international conference on Ada (SIGAda)*, pages 1–6, New York, NY, USA, 1998. ACM.
6. S. Fischmeister and Y. Ba. Sampling-based Program Execution Monitoring. In *ACM International conference on Languages, compilers, and tools for embedded systems (LCTES)*, pages 133–142, 2010.
7. S. Fischmeister and P. Lam. On Time-Aware Instrumentation of Programs. In *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 305–314, San Fransisco, United States, Apr. 2009.
8. S. Fischmeister and P. Lam. Time-aware Instrumentation of Embedded Software. *IEEE Transactions on Industrial Informatics*, 2010.
9. 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.
10. H. Kopetz. Event-Triggered Versus Time-Triggered Real-Time Systems. In *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*, pages 87–101, London, UK, 1991. Springer-Verlag.
11. M. Li, T. V. Achteren, E. Brockmeyer, and F. Catthoor. Statistical Performance Analysis and Estimation of Coarse Grain Parallel Multimedia Processing System. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 277–288, Washington, DC, USA, 2006. IEEE Computer Society.
12. G. McCall. *Misra-C: 2004*. MIRA Limited, Warwickshire, United Kingdom, 2004.
13. A. Mok. Firm Real-time Systems. *ACM Comput. Surv.*, 28, December 1996.
14. S. Natarajan. *Imprecise and Approximate Computation*. Kluwer Academic Publishers, Norwell, MA, USA, 1995.
15. S. Navabpour, C. W. W. Wu, B. Bonakdarpour, and S. Fischmeister. Efficient techniques for near-optimal instrumentation in time-triggered runtime verification. In *Runtime Verification (RV)*, 2011. To appear.
16. Radio Technical Commission for Aeronautics (RTCA). *Software Considerations in Airborne Systems and Equipment Certification*, Dec. 1992.
17. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.