Lessons Learned on Assumptions and Scalability with Time-Aware Instrumentation

Guy Martin Tchamgoue
Department of Electrical and Computer Engineering
University of Waterloo
gmtchamg@uwaterloo.ca

Sebastian Fischmeister
Department of Electrical and Computer Engineering
University of Waterloo
sfischme@uwaterloo.ca

ABSTRACT

Software instrumentation is a key technique in many stages of the development process. Instrumentation is particularly important for profiling, debugging, performance evaluation, and security analysis of real-time and embedded systems. Unfortunately, typical software-based instrumentation methods, while useful to extract high-level information from programs, concentrate on preserving only logical correctness and are thus inadequate for time-sensitive applications for which timing must also be preserved.

Time-aware instrumentation is a new view on code instrumentation, one that considers extra-functional properties and specifically timing constraints of instrumented programs. Time-aware instrumentation enables instrumenting software systems while still guaranteeing their timing requirements. This paper summarizes the work on time-aware instrumentation and highlights the lessons learned on assumptions and scalability. Specifically, it shows how strict assumptions enable a strong formal model at the expense of applicability. Subsequent relaxing of assumptions then permits scaling the concept and applying it to large real-word applications with millions lines of code. We believe that these lessons may help steer other research projects in the systems area.

CCS Concepts

•Software and its engineering → Software testing and debugging; Traceability; •General and reference → *Metrics; Performance;*

Keywords

Instrumentation, Tracing, Debugging

1. INTRODUCTION

Modern software systems including embedded systems are getting more and more complex, growing far beyond millions of lines of code. Consequently, debugging programs without observing the system behaviour at run time has become virtually impossible. Program instrumentation is the key technique for observing programs at run time. Instrumentation is used in many stages of the development process as it generates execution traces used by analysis tools such as profilers [5, 18], debuggers [14, 16, 3], and malware detectors [7]. Instrumentation statically or dynamically injects user-defined analysis code into programs to generate execution traces at run time. Program analysis tools are critical for understanding the run-time behavior of software. Unfortunately, typical program instrumentation methods, while useful to extract high-level information from programs, concentrate on preserving only logical correctness and are thus inadequate for time-sensitive applications such as real-time embedded systems. Such systems require instrumentation that preserves both logical correctness and temporal correctness. Standard program instrumentation often incurs extra delays that may change the temporal behavior of the monitored system, leading to misleading execution traces.

Time-aware instrumentation is a new view on code instrumentation, one that considers extra-functional properties and specifically timing constraints. Thus, time-aware instrumentation allows to instrument software systems while still guaranteeing their timing requirements. In recent years, several tools and techniques [1, 6, 8, 10, 11, 12] have emerged to support the instrumentation of time-sensitive systems. Static time-aware instrumentation methods [6, 8, 10, 11, 12] inject analysis code into a program prior to execution; potentially extending the execution on all paths, but also ensuring that no path exceeds a specified time budget. Complementary, a dynamic time-aware instrumentation [1] method dynamically inserts analysis functions into the program's binary at run time to generate execution time traces.

This paper summarizes the propositions on time-aware instrumentation, that range from formal models to a scalable dynamic implementation applicable to deployed systems, to highlight the lessons learned on assumptions and scalability. Basically, we show that setting strict assumptions on both the monitored program and the environment allowed us to build a formal model of instrumentation that comes with predictive metrics, but also with reduced applicability and scalability. On the other hand, subsequent relaxation of specific assumptions allowed us to design a flexible and scalable dynamic time-aware instrumentation framework, which is directly applicable to deployed real-world applications with millions lines of code.

The remainder of the paper is organized as follows. Sec-

EMSOFT'16, October 01-07 2016, Pittsburgh, PA, USA

ISBN 978-1-4503-4485-2/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2968478.2975584

tion 2 describes the challenges faced by instrumentation tools. Section 3 presents a general overview of time-aware instrumentation of software systems. Section 4 summarizes the state-of-the-art on static time-aware instrumentation and shows how strict assumptions lead to predictable but less scalable formal model. Section 5 summarizes the work on dynamic time-aware instrumentation and shows how relaxing the assumptions on the formal model allowed to build a scalable tool. Section 6 presents the lessons learned in building time-aware instrumentation frameworks. Section 7 concludes the paper.

2. INSTRUMENTATION CHALLENGES

Program analysis tools are critical for understanding the run-time behavior of software. Instrumentation of a program can be achieved either statically (i.e., by analyzing the source code and inserting instrumentation) or dynamically (i.e., by manipulating the program during execution to insert instrumentation). Instrumentation and tracing in general incur overhead at execution time. The overhead depends on the concrete application and the goal of the instrumentation. Consequently, the overhead may range from negligible to devastating [8]. For instance, depending on the tool, VALGRIND, a dynamic binary instrumentation tool, can impose a slowdown of up to a factor of 22.2 or worse [17]. However, while some systems may tolerate such a slowdown, time-sensitive applications with physical interaction (e.g., controlling a plant) usually do not tolerate overhead of more than a few percent. Moreover, the added overhead may change the behavior of the monitored program, leading to misleading execution traces and even Heisenbugs [16]. Consequently, an instrumentation tool for time-sensitive applications must maintain the logical and temporal correctness of the monitored application.

As real-time and embedded systems grow in size and complexity, debugging them becomes challenging for developers. The traditional *breakpoint* and *step-through* method as well as the manually inserted *print()* instructions become not only time consuming but also ineffective for real analysis. It is therefore essential to provide robust and powerful techniques to facilitate the tracing and analysis of such systems. Time-aware instrumentation aims to preserve the timing requirements of the system while generating useful execution traces. Time-aware instrumentation can be achieved both through static and dynamic instrumentation.

3. OVERVIEW OF TIME-AWARE INSTRUMENTATION

Time-aware instrumentation extends the basic concept of instrumentation in the temporal domain. It thus uses new concepts, a new model, and a workflow different from that of traditional static and dynamic instrumentation.

3.1 Concept

Fischmeister and Lam [8] introduced the concept of time-aware instrumentation to support the instrumentation of time-sensitive applications. The key idea behind the time-aware instrumentation of a system is to transform the execution time distribution, maximizing the *coverage* of the execution trace while always staying within the time budget of the system. By coverage, we mean the amount of data extracted during instrumentation versus the amount of

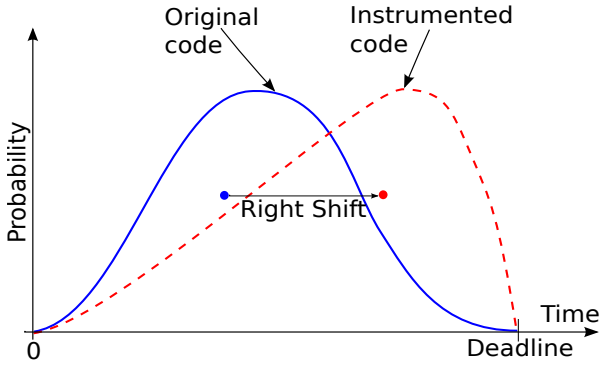

Figure 1: Execution time distribution for a code block before and after time-aware instrumentation showing the shift in the expected execution time [9].

data that should have been extracted. Time-aware instrumentation injects analysis code, potentially extending the execution time on all paths, and ensures that no path takes longer than the specified time budget.

Sometimes, instrumenting on all execution paths while respecting the timing requirements of the program is impossible. There are two possible solutions to this problem: increasing the time budget for the instrumentation or resorting to instrumenting only a subset of all paths (i.e., partial instrumentation). Increasing the time budget on some execution paths may cause deadline misses. This is acceptable for soft real-time systems for instance, where a few deadline misses may still be tolerated. A *partial instrumentation* of a program inserts instrumentation code only at selective instrumentation points and thus, produces only a partial execution trace. Often, this means instrumenting only on non-worst-case execution paths to preserve the timing requirements of the program. However, the instrumentation points are always selected to maximize the coverage of the instrumentation process. Partial instrumentation and subsequently partial traces are useful for building an inductive analysis mechanism for deployed resource and space constrained systems. Many existing analysis tools [3, 14, 18] support partial traces. In some cases, tracing a program for longer time periods or repeatedly may enhance the quality of the final coverage of a partial instrumentation process.

Time-aware instrumentation attempts to honor the timing constraint and shifts the execution time profile closer to the program's deadline as seen in Figure 1. The x-axis of Figure 1 specifies the execution time, while the y-axis indicates the frequency of observing a particular execution time. The right shift of the execution time observed in Figure 1 occurs, because time-aware instrumentation adds tracing code to execution paths, increasing their running times, but ensuring that execution times never exceed the deadline.

Figure 1 also suggests that the workflow for time-aware instrumentation differs from that of standard instrumentation as it needs an extra step to take care of the timing properties of the system. Figure 2 shows the workflow of a time-aware instrumentation tool operating at the source-code level. Initially, a function of interest is selected for instrumentation. At the next stage, the source code analyzer is invoked to extract the control flow graph and break the function into execution paths. In a first try, naive instrumentation is per-

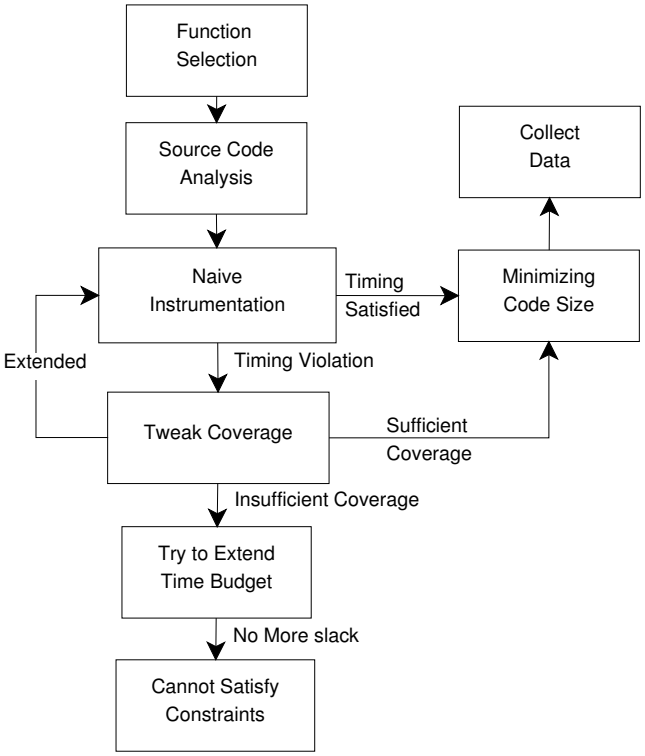

Figure 2: Workflow for time-aware instrumentation of hard real-time systems [8, 9].

formed on all variables of interest. Next, the instrumentation process uses the difference of time between execution paths and different basic blocks to ensure that the instrumented program stays within the time limits of the original program on the worst-case paths. If the timing constraint is violated, the process can employ integer linear programming to lower the coverage of the instrumentation so that it meets the timing requirements.

If the instrumentation coverage is insufficient, then the process will either have to give up (e.g., if it cannot extend the time budget available for the function and the instrumentation), or try to extend the time budget and thereby increases the coverage. If the optimized instrumentation meets the required coverage or if the initial naive instrumentation does not extend the worst-case path, then the whole process will continue and use the execution paths to minimize the required code size. Afterwards the developer can recompile the program to collect traces from the instrumentation.

To perform instrumentation based on the above workflow, it is important to build a *model* that, for instance, allows estimating the impact and effectiveness of different instrumentations on the worst-case execution path.

3.2 Model Definition

For time-aware instrumentation of hard real-time embedded systems, Fischmeister and Lam [8] propose an *abstract model* that captures relevant properties from the source code. This model comprises temporal behavior and control flow of the program, and also specifies the data to be traced. The model enables, for instance, calculating how the execution time changes on each control-flow path using worst-case execution time (WCET) analysis. However, the main use of the model is to determine the *optimal* instrumentation for execution time traces. Optimal means that, given a time budget for the instrumentation overhead, the model provides the best instrumentation possible in terms of trace coverage while guaranteeing the timing requirements of the instrumented application.

The model abstracts a program source code as a directed graph $G = \langle V, E \rangle$, representing the interprocedural control flow of the program, and uses the functions $c : V \to \mathbb{R}$ and $p : E \to [0, 1]$ to capture the program's behavior. Thus, each vertex $v \in V$ in the control-flow graph (CFG) represents a basic block in the program, i.e., a unit of execution in the program that has a single entry and exit points. $E \subseteq V \times V$ is the set of edges that represent the flow of control in the CFG of the program.

Each vertex $v \in V$ is further abstracted by $\langle A, L \rangle$, where A is the set of assignments and L, the logged variables. The function $c : V \to \mathbb{R}$ specifies the required computation time c, or cost, for a vertex v. For instance, $c(v_0) = 12.2t$ means that the basic block at vertex v_0 requires 12.2 time units for its execution. Edges $e := \langle v_s, v_d \rangle$ specify transitions from a source vertex v_s to a destination vertex v_d. The function $p : E \to [0, 1]$ computes the probability $p(e)$ that the execution will use edge e to leave vertex v_s. So, $p(\langle v_0, v_1 \rangle) = 0.5$ means that on average every other execution will continue at vertex v_1 after executing v_0.

3.3 Characteristics

The abstract model is useful as it allows for complete formalization of time-aware instrumentation of real-time embedded systems. For instance, to accommodate interrupts, the model can use an adjusted computation time c of any vertex v using response time analysis (see [9] for details). Handling interrupts is critical for the model to capture the behavior of a real system. The adjusted model assumes that interrupts occur as sporadic events with a known minimal inter-arrival time, and that interrupt service routines are bounded and always eventually terminate.

Furthermore, the model enables formulating an optimization problem to determine the instrumentation points that maximize the information gain without violating the time budget of safety-critical systems. The instrumentation framework, INSTEP [10], supports up to four extra-functional properties including instrumentation intent values, code size, execution time, and detection latency, that developers can use as objectives and constraints to the optimization problem. The complete formalization of the instrumentation problem permits full control over the instrumentation process. It enables, for example, to precisely compute ahead of time the minimal trace buffer size, i.e., the maximal buffer size required at execution time to store traces. This is important for embedded systems that use memory-constrained devices.

4. STATIC TIME-AWARE INSTRUMENTATION

This section summarizes the state-of-the-art on static time-aware instrumentation to show how strict assumptions supports creating a formal model and formulating predictions, but at the expense of applicability.

4.1 Overview

The abstract model presented in Section 3.2 has been used in the implementation of static time-aware instrumentation frameworks for hard real-time and embedded systems [8, 10, 11, 12], but also for instrumenting model-based applications [6]. The instrumentation process, as described in Figure 2, follows four main steps: (1) extract the control flow paths with variable assignments for a function; then, (2) check whether the instrumentation stays within the time budget. If it does not, (3) compute the maximum instrumentation coverage which respects the time budget and (4) optimize this instrumentation for code size. Basically, the instrumentation frameworks differ on how they handle steps 2 and 3.

Thus, while Fischmeister and Lam [8, 9] only instrument on non-worst-case execution paths, Kashif *et al.* [12] use runtime slack-based conditional instrumentation to execute the instrumentation code only when the execution is guaranteed not to increase the WCET beyond the program's deadline. Kashif and Fischmeister [11] relied on code transformations including branch block creation and CFG cloning to increase the instrumentability of a program on and thus, maximize the instrumentation coverage. The instrumentation framework, INSTEP [10], accepts multiple competing extra-functional properties, and uses trees to represent instrumentation intents and automata for cost models. INSTEP further uses local searching to prune to search space of its optimization problem that identifies the optimal instrumentation points. Denil *et al.* [6] propose to instrument model-based applications directly at model level and use a set of rule-based model transformation techniques to optimize the placement of instrumentation blocks, while satisfying the extra-functional constraints of the applications.

Using a model allows to gain insights about the instrumented program, e.g., the *right shift* as depicted in Figure 1 highlights the pressure made by the instrumentation process on the worst-case execution paths of the instrumented program. Furthermore, it becomes possible to define metrics that help predict for example the coverage and the behavior of the instrumented program. One such metrics, the *execution time profile shift effectiveness* [11], measures the effectiveness of a time-aware instrumentation approach, so that different approaches may be compared against each other.

Finally, the static instrumentation frameworks allowed to formulate experiments to test and evaluate the abstract model. Testing the model provides insight about the benchmark applications and also validate the model itself. A validated model provides assurance and quality guarantees about its results on real use-case scenarios. For instance, the model was successfully used for the time-aware instrumentation of a micro-controller code [8], an automotive controller module [10], and the SNU real-time benchmark suite [11].

4.2 Assumptions

The construction of the abstract model of instrumentation relies on a set of strict assumptions. First of all, the instrumentation process is offline, assuming a system with hard real-time requirements. The offline process implies that the complete source code of the program is available. This includes not only the source code of the program, but also the source code of all libraries referenced by the program. Since some of these libraries may be large in size, statically analyzing them may be cumbersome and impractical. Moreover, the source code of libraries may be not always accessible or available.

Further, the model assumes the input source program to be analyzable. For instance, the program should be MISRA-C compliant [15]. MISRA-C provides a standard for implementing safety-critical real-time systems. For example, MISRA-C requires bounded loops, limits recursion, and restricts the usage of pointers, unions, and dynamic memory allocation. This also facilitates the computation of the WCET even at the level of basic blocks, because the model needs to easily determine the critical paths of the program to decide whether to inject its instrumentation code.

Benchmark	Description
OpenEC [8, 9]	Instrumentation of the `handle_power` function of the OpenEC firmware, 20 variables, 42 basic blocks with 20 different control-flow paths.
Flash Filesystem [9]	Instrumentation of a set of 30 functions from a FAT-like filesystem for flash devices containing a total 3000 lines of C code.
SNU Benchmark [10, 11, 12]	The SNU real-time benchmark suite contains 17 C programs with 120 lines of code and 34 basic blocks on average.
EasyWEB [10]	A dynamic web server for NXP LPC17xx ARM-based micro-controllers with a total of 1,846 lines of C code.
Automotive Controller [10]	An automotive control module with 177,298 lines of C code and 6,297 basic blocks.
Adaptive Controller [6]	An adaptive controller that consists of three Simulink models.

Table 1: Case studies of time-aware instrumentation

4.3 Consequences

Setting strict assumptions facilitates the construction of an interesting formal model with good predictive metrics, however, it does so at the expense of applicability and scalability. Thus, using the model and systems building on the model for case studies is time consuming as experiments are specifically made for each application. Solving optimization problems, selecting the right parameters, and performing WCET analysis, make each experiment unique and expensive to realize. Table 1 presents the different case studies where static time-aware instrumentation has been applied. For example, the OpenEC experimental setup cannot be directly used for the Flash Filesystem case. Parameters such as the number of basic blocks, the code size, the number of control-flow paths, the number of functions and variables influence the configuration and the complexity of the optimization problem solved by the instrumentation model, making it difficult to generalize experiments. Already for small systems like the SNU real-time benchmark, Kashif *et al.* [10] showed that the number of equations and expressions in the optimization problem could exceed 1,500 and 33,000, respectively.

Finally, having a model with limited applicability favors *shoehorning*. For instance, experiments revealed that the instrumentation coverage remained low, because large por-

Assumptions	Consequences
Offline	Source code availability, which is not always the case especially for legacy software and proprietary libraries.
Attainable WCET	Restricted applicability which favors shoehorning, no support for concurrent application.
MISRA-C Compliance	Restricted code: No pointers, no dynamic allocation.
Hard real-time systems	Specifically designed and time consuming experiments and case studies.

Table 2: Summary of assumptions and consequences for a static time-aware instrumentation model

tions of the programs were still unavailable for instrumentation. Instrumenting these code areas could affect the WCET and thus violate existing timing constraints. Code transformations [11] and optimizations are introduced to increase the coverage, and thus the usability of the instrumentation technique. However, this also increases the complexity of the model. Another example is the use of hardware support to increase the coverage [12]. While the hardware solution may provide up to 80% coverage, it also requires to extend the underlying processor with specialized instructions, which however further limits the applicability and portability of the framework.

Furthermore, applying WCET analysis forces the model to consider only foreground/background and multiprogramming systems with run-to-completion semantics. While the static model of instrumentation is sound and effective, the need for running WCET analysis before and after the instrumentation reduces the applicability to only hard real-time systems where WCET analysis is common. Concurrency, for example, is not supported as it complicates the evaluation of the cost function, and therefore that of the WCET analysis. Table 2 summarizes the design assumptions and their consequences on the final model and deriving framework.

5. DYNAMIC TIME-AWARE INSTRUMENTATION

This section describes the time-aware dynamic binary instrumentation of soft real-time systems and shows how relaxing assumptions may help build a widely applicable and scalable framework.

5.1 Overview

Using an abstract model produces strong academic results, which may however be difficult to scale to real-world applications. Relaxing assumptions may help simplify the model and build a more scalable system. Thus, loosing the assumptions of Table 2 facilitated the design of a time-aware dynamic binary instrumentation [1].

Dynamic time-aware instrumentation adds an adjustable bound on the timing overhead of the program under analysis, contrarily to general-purpose dynamic binary instrumentation (DBI) frameworks such as DYNINST [2], DYNAMORIO [4], PIN [13], VALGRIND [17], and PEMU [19]. DBI frameworks allow to inspect a running program at different levels of granularity (e.g. *image*, *trace*, *routine*, and *instruction* levels)

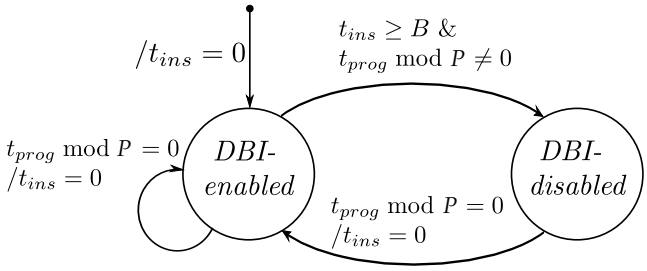

Figure 3: State machine for dynamic time-aware instrumentation with DIME [1].

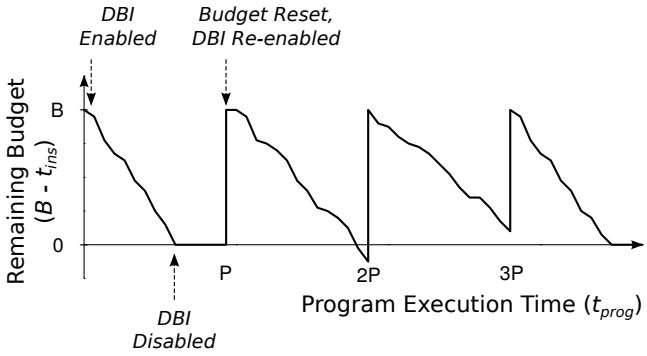

Figure 4: Budget supply for time-aware instrumentation with DIME [1].

by dynamically injecting user-defined analysis code into the program's binary. DIME [1] is a time-aware dynamic binary instrumentation (TDBI) framework that conserves all the properties of a DBI.

To reduce the instrumentation overhead while maximizing its coverage, DIME maintains and periodically switches between two versions of the monitored program: (1) an *instrumented version* that contains the user-defined analysis functions and (2) an *uninstrumented version* that contains no user-defined analysis code. Figure 3 shows the state diagram of DIME, where t_{ins} represents the instrumentation time per period, and t_{prog} the execution time of the program. DIME allows the user to define an instrumentation period P and an instrumentation budget $0 \leq B \leq P$. Thus, during each period P, DIME instruments the program only for B time units, i.e., as long as $t_{ins} < B$, the instrumentation is enabled as in Figure 3. Once, the budget is consumed, i.e., $t_{ins} \geq B$, DIME disables instrumentation and switches to the uninstrumented version of the program. This state switching reduces the overhead and limits deadline misses as the uninstrumented version surely runs faster than the instrumented version. At the end of each instrumentation period as shown in Figure 4, the budget is reset and DIME switches back to the instrumented version of the program. Figure 4 depicts how the budget is replenished and consumed for instrumentation. Sometimes, the instrumentation process may overshoot, violating the defined instrumentation budget as seen Figure 4 around $2P$ time units.

5.2 Assumptions

The dynamic time-aware instrumentation of programs assumes only the availability of executable binaries. This in-

creases the applicability of such a technique to legacy software where the source code may be unavailable. Contrary to the approach using static instrumentation, focusing on binaries removes any restrictions on the structure of the program.

The instrumentation happens online and often relies on a just-in-time compiler to dynamically disassemble, inject the analysis code, and recompile the binary at execution time. This process is however well-known to incur a high runtime overhead and therefore, cannot be applied to programs with hard real-time requirements. Thus, TDBI considers only soft real-time systems such as media players where a few deadline misses may be tolerated.

5.3 Consequences

The limited assumptions of time-aware dynamic instrumentation increases not only the technique's applicability, but also its scalability compared to the static approach. For example, DIME was used to instrument the whole SPEC2006 C benchmark suite and the complete VLC Media Player application. The VLC Media Player counts approximately 600 000 lines of code and uses libraries with more than three million lines of code. Instrumenting the binary removes the restriction on the structure of the application, reduces the cost of experiments, and increases the possibility of case studies. DBI can handle dynamically generated code as it maintains a global view of the program at run time. Finally, DBI offers the possibility to write rich and complex analysis tools, making the technology easy to adopt widely for software analysis and security applications.

While being more scalable and more applicable, TDBI also loosens the formal model of the static instrumentation. The formal model for the static analysis approach uses WCET analysis to ensures that the timing requirements of the instrumented program are not violated. TDBI modifies the executable at run time and thus, it becomes impossible to predict the overhead and the runtime requirements (e.g., memory space and execution time) of the instrumentation process. Consequently, TDBI cannot guarantee to always meet the deadlines of the application. Therefore, the user-defined analysis function may sometimes overshoots the predefined instrumentation budget as shown in Figure 4. Arafa *et al.* [1] show that overshoots may impact the overhead imposed by the instrumentation tool. Therefore, TDBI is unsuitable for instrumenting systems with hard real-time requirements, however, it is suitable for instrumenting soft real-time systems.

On the other hand, TDBI also allows to build probabilistic models of instrumentation as statistical data can be collected and analyzed to provide an insight on the program at run time. This statistical data may include, for instance, the quality of service of the monitored program. In the case of the VLC Media Player, this may, for example, be the number of frames decoded per second. The analysis of this data opens new interesting problems such as feedback-based dynamic instrumentation where the statistical data collected from the monitored program may help steer the instrumentation process itself. Another direction is to perform runtime verification on partial execution traces produced by time-aware instrumentation. Finally, mining partial traces may, for instance, allow to extract bug patterns and models that may help developers analyze programs with reduced overhead.

6. DISCUSSION

Working on time-aware instrumentation allowed us not only to gain insights into timing and real-time systems, but also into the relationship between assumptions, applicability, and scalability for systems research. Clearly, setting strict assumptions such as requiring the availability of source, imposing a certain structure, mandating the type of program, and requiring a complete WCET analysis, allowed us to construct a formal model with strong predictive metrics. The model facilitated the implementation of a static time-aware instrumentation framework that provides useful insights such as the right shift about the instrumented time-sensitive program. However, the static instrumentation framework is also impacted by the many assumptions of the underlying abstract model resulting in a brittle approach with limited applicability and scalability. Our experiments also showed that a restrictive formal model makes case studies expensive and difficult to realize. This prevents the generalization of results and limits the applicability of the model to real-world programs.

Relaxing assumptions brings big rewards. Relaxing most of the assumptions on the formal model facilitated the design and implementation of a flexible and scalable dynamic time-aware instrumentation framework. Unfortunately, the strong predictions enabled by the formal model were lost in the process, however, the concepts (e.g., right shift and information gain through subsequent runs) survived. In addition, the new framework with the relaxed assumptions opened up a number of new challenges and perspectives.

7. CONCLUSIONS

This paper summarizes the current research work on time-aware instrumentation and highlights the lessons learned with respect to assumptions and scalability. Setting initial strict assumptions allowed us to build a formal model, obtain clear insights, and predict the behavior of the instrumented program. The model helped formulate the problem and explore tradeoffs for the implementation. However, the initial model has only reduced applicability and scalability in real-word systems as the set of assumptions and the complexity of the model are cost prohibitive to realize in large applications. Subsequent relaxation of assumptions allowed us to build a flexible and more scalable dynamic time-aware instrumentation tool that still incorporates the core concepts of the initial model. We believe that this process of starting off with strict assumptions to define models and get strong results, and then subsequent relaxation of assumptions to achieve scalability and applicability is a worthwhile process and a good lesson learned for systems research in real-time embedded systems.

8. ACKNOWLEDGMENTS

We would like to thank Hany Kashif and Pansy Arafa for their work on time-aware instrumentation that greatly contributed to the writing of this paper.

9. REFERENCES

[1] P. Arafa, H. Kashif, and S. Fischmeister. DIME: Time-aware Dynamic Binary Instrumentation Using Rate-based Resource Allocation. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, EMSOFT'13, pages 25:1–25:10. ACM, 2013.

[2] A. R. Bernat and B. P. Miller. Anywhere, Any-time Binary Instrumentation. In *Proceedings of the 10th Workshop on Program Analysis for Software Tools*, PASTE'11, pages 9–16. ACM, 2011.

[3] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional Detection of Data Races. *SIGPLAN Not.*, 45(6):255–268, June 2010.

[4] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, Sept. 2004.

[5] H. K. Cho, T. Moseley, R. Hank, D. Bruening, and S. Mahlke. Instant Profiling: Instrumentation Sampling for Profiling Datacenter Applications. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'13, pages 1–10. IEEE, 2013.

[6] J. Denil, H. Kashif, P. Arafa, H. Vangheluwe, and S. Fischmeister. Instrumentation and Preservation of Extra-functional Properties of Simulink Models. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, DEVS'15, pages 47–54. SCSI, 2015.

[7] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic Spyware Analysis. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 18:1–18:14. USENIX Association, 2007.

[8] S. Fischmeister and P. Lam. On Time-Aware Instrumentation of Programs. In *Proceedings of the 2009 15th IEEE Symposium on Real-Time and Embedded Technology and Applications*, RTAS'09, pages 305–314. IEEE, 2009.

[9] S. Fischmeister and P. Lam. Time-Aware Instrumentation of Embedded Software. *IEEE Transactions on Industrial Informatics*, 6(4):652–663, Nov 2010.

[10] H. Kashif, P. Arafa, and S. Fischmeister. INSTEP: A Static Instrumentation Framework for Preserving Extra-Functional Properties. In *IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA'13, pages 257–266, Aug 2013.

[11] H. Kashif and S. Fischmeister. Program Transformation for Time-aware Instrumentation. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies Factory Automation*, ETFA'12, pages 1–8. IEEE, Sept 2012.

[12] H. Kashif, J. Thomas, H. Patel, and S. Fischmeister. Static Slack-Based Instrumentation of Programs. In *Proceedings of the 20th IEEE Conference on Emerging Technologies Factory & Automation*, ETFA'15, pages 1–8. IEEE, Sept 2015.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[14] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-race Detection. *SIGPLAN Not.*, 44(6):134–143, June 2009.

[15] MIRA Limited. MISRA-C:2004 guidelines for the use of the C language in critical systems, oct 2004.

[16] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280. USENIX Association, 2008.

[17] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[18] M. Serrano and X. Zhuang. Building Approximate Calling Context from Partial Call Traces. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO'09, pages 221–230. IEEE, 2009.

[19] J. Zeng, Y. Fu, and Z. Lin. PEMU: A Pin Highly Compatible Out-of-VM Dynamic Binary Instrumentation Framework. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE'15, pages 147–160. ACM, 2015.