Time-Aware Dynamic Binary Instrumentation

by

Pansy Arafa

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2017

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner	Sathish Gopalakrishnan Associate Professor, University of British Columbia, Electrical and Computer Engineering Department
Supervisor	Sebastian Fischmeister Associate Professor, University of Waterloo, Electrical and Computer Engineering Department
Internal Member	Hiren Patel Associate Professor, University of Waterloo, Electrical and Computer Engineering Department
Internal Member	Werner Dietl Assistant Professor, University of Waterloo, Electrical and Computer Engineering Department
Internal-external Member	William Cowan Associate Professor, University of Waterloo, David R. Cheriton School of Computer Science

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contribution

In what follows is a list of publications which I have co-authored and used their content in this dissertation. For each publication, I present a list of my contributions.

The use of the content, from the listed publications, in this dissertation has been approved by all co-authors.

Dynamic Instrumentation Work:

1. **Pansy Arafa**, Hany Kashif, and Sebastian Fischmeister. *Dime: Time-aware Dynamic Binary Instrumentation Using Rate-based Resource Allocation.* In Proceedings of the 13^{th} International Conference on Embedded Software (EMSOFT), Montreal, Canada, September 2013 [17].
   - Co-designed DIME;
   - Developed the three implementations of DIME;
   - Analyzed the experimental results;
   - Co-conducted the case studies;
   - Wrote portions of the paper.
2. **Pansy Arafa**, Hany Kashif, and Sebastian Fischmeister. *Time-aware Dynamic Binary Instrumentation* (Journal Version). Under Submission.
   - Co-designed and implemented the redundancy-suppression feature;
   - Analyzed the experimental results;
   - Conducted the case studies;
   - Designed and conducted the parameter tuning experiments;
   - Analyzed the parameter tuning results;
   - Wrote the majority of the paper.
3. **Pansy Arafa**, Guy Martin Tchamgoue, Hany Kashif, and Sebastian Fischmeister. *QDIME: QoS-aware Dynamic Binary Instrumentation.* In Proceedings of the 25^{th} International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Banff, Canada, September 2017 [19].
   - Co-designed and implemented QDIME;

- Co-designed and conducted the case studies;
- Analyzed the results;
- Wrote portions of the paper.

Preliminary Work:

4. **Pansy Arafa**, Daniel Solomon, Samaneh Navabpour, Sebastian Fischmeister. *Debugging Behaviour of Embedded-Software Developers: An Exploratory Study.* In Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Raleigh, USA, October 2017 [18].
   - Coded the study videos with the co-authors;
   - Analyzed the experimental results;
   - Wrote the paper.
5. Hany Kashif, **Pansy Arafa**, and Sebastian Fischmeister. *INSTEP: A Static Instrumentation Framework for Preserving Extra-functional Properties.* In Proceedings of the 19^{th} International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Taipei, Taiwan, August 2013 [55].
   - Contributed to the design of INSTEP;
   - Executed static analysis and WCET analysis of the benchmarks;
   - Wrote portions of the paper.
6. Joachim Denil, Hany Kashif, **Pansy Arafa**, Hans Vangheluwe, and Sebastian Fischmeister. *Instrumentation and Preservation of Extra-functional Properties of Simulink Models.* In Proceedings of the Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium, Alexandria, USA, April 2015 [32].
   - Co-designed the instrumentation framework;
   - Wrote a portion of the paper.

Abstract

The complexity of modern software systems has been rapidly increasing. Program debugging and testing are essential to ensure the correctness of such systems. Program analysis is critical for understanding system's behavior and analyzing performance. Many program analysis tools use instrumentation to extract required information at run time. Instrumentation naturally alters a program's timing properties and causes perturbation to the program under analysis. Soft real-time systems must fulfill timing constraints. Missing deadlines in a soft real-time system causes performance degradation. Thus, time-sensitive systems require specialized program analysis tools. Time-aware instrumentation preserves the logical correctness of a program and respects its timing constraints. Current approaches for time-aware instrumentation rely on static source-code instrumentation techniques. While these approaches are sound and effective, the need for running worst-case execution time (WCET) analysis pre- and post-instrumentation reduces the applicability to only hard real-time systems where WCET analysis is common. They become impractical beyond microcontroller code for instrumenting large programs along with all their library dependencies.

In this thesis, we introduce theory, method, and tools for time-aware dynamic instrumentation realized in DIME tool. DIME is a time-aware dynamic binary instrumentation framework that adds an adjustable bound on the timing overhead to the program under analysis. DIME also attempts to increase instrumentation coverage by ignoring redundant tracing information. We study parameter tuning of DIME to minimize runtime overhead and maximize instrumentation coverage. Finally, we propose a method and a tool to instrument software systems with quality of service (QoS) requirements. In this case, DIME collects QoS feedback from the system under analysis to respect user-defined performance constraints. As a tool for instrumenting soft real-time applications, DIME is practical, scalable, and supports multi-threaded applications. We present several case studies of DIME instrumenting large and complex applications such as web servers, media players, control applications, and database management systems. DIME limits the instrumentation overhead of dynamic instrumentation while achieving a high instrumentation coverage.

Acknowledgements

First and foremost, I am grateful to God for bestowing upon me the strength and the knowledge to complete the research work for this thesis.

I am deeply thankful to my supervisor Professor Sebastian Fischmeister, for the opportunities, the guidance, and the continuous support. His advice and consideration assisted me to acquire invaluable skills and find my passion. I would also like to thank my committee members: Professor William Cowan, Professor Hiren Patel, Professor Werner Dietl, and Professor Satish Gopalakrishnan for taking the time and the effort to participate in my examination committee and provide me with valuable feedback.

No words can describe my gratitude to my parents, Sahar and Mohammad. I would not have been here without their sacrifices, kindness, and unlimited support. I thank my mother for always being my closest friend and my role model. Her words and prayers lighten up my days. Thanks to my father for being my backbone in this life. I would like to thank my brother Ahmed and his family whom I miss every day. Also, thanks to my brother Mostafa who always makes me proud. Huge thanks to my lifetime friends for their words of support and encouragement.

Thanks to my little son, Adham; his innocent smiles cheer up my days, and his hugs give me all the strength of the world. Last, but not least, I would like to express my endless gratitude to my husband, Hany. Thanks for pushing me to chase my dreams. Thanks for being the most loving and understanding husband. Thanks for everything beautiful we have together.

Dedication

To my beloved husband, Hany.
To my dear son, Adham.
To my parents, Sahar & Mohammad.
To my siblings, Ahmed & Mostafa.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis, we address the problem of dynamic instrumentation of real-time systems. Program analysis and profiling are essential for understanding program behavior. Program profiling tools can use instrumentation to extract required information at runtime. Instrumentation naturally alter the program behavior especially the timing properties. We introduce the theory, method, and tools for time-aware dynamic binary instrumentation which respects the timing constraints of the program under analysis.

1.1 Motivation

Debugging of complex software systems is difficult and expensive. A study of major U.S. software engineering industries in 2002 revealed that software engineers spend, on average, 70-80% of their time testing and debugging [61, 41]. According to the study's estimation, testing and debugging costs the U.S. economy over $50 billion annually. Thus, it is essential to investigate more efficient and less expensive debugging techniques.

As a preliminary work, we conducted an exploratory study that investigates the developers' behavior while debugging a real-time embedded software system [18]. The study further confirms the complexity and the difficulty of the debugging activity, especially for real-time systems. The study involves 14 programmers debugging semantic errors of a familiar real-time embedded software that consists of 3085 lines of code (LOC). Debugging of real-time embedded software is challenging due to (1) hardware interaction, e.g., loading code to the target board, (2) use of low-level language semantics, e.g., memory management in C, and (3) the need to respect the system's timing requirements. Also,

the study includes seven distinct bugs categorized into incorrect-hardware-configuration bugs and memory leaks. The primary data-collection method is the video-recording of the computer screens during the study sessions. Afterward, multiple observers code the videos to extract information revealing the debugging behavior of the participants. Moreover, when a participant compiles the system, it gets copied to a separate folder that is used later to view the edits made in-between the compilation tries. Finally, the participants fill out multiple forms to describe their experience through the debugging sessions.

Although the participants are familiar with the system under investigation, they faced a noticeable difficulty in locating and fixing the bugs. Only in 64% of the debugging sessions, the participant was able to find the code location of the bug. They also spent a long time examining the bugs regardless the bug's type. The total time spent examining a bug varies between 9 minutes and 1.9 hours with an average of 33 minutes. Only in 38% of the debugging sessions, the participant successfully fixed the bug. Moreover, we noticed two forms of indecisive behavior that can contribute to failing to fix the bugs. According to the *activity visitation pattern* [18], many participants show a high frequency of transition between debugging activities while examining a bug. The debugging activities of interest are code browsing, code editing, document reading, compiling, and testing. The second observed indecisive behavior is called the *ping-pong behavior*. It means, according to the editing location, moving far from the bug after approaching it at least twice. In many debugging sessions, the participant adopted this behavior decreasing his chance to fix the bug.

Program profiling and analysis are useful for debugging and understanding the runtime behavior of programs. Many analysis frameworks use instrumentation to extract runtime information during program execution. Instrumentation implies the insertion of extra instructions that collect the traces into the program. Therefore, instrumentation adds delay to the program execution. Both, hard and soft, real-time systems must fulfill timing constraints.

There exists research work on time-aware methods to preserve the timing properties of real-time systems during the instrumentation process [37, 56, 55, 57]. Current approaches for time-aware instrumentation solely rely on static and source-code instrumentation techniques. Previous works require WCET analysis of the input program to guide the placement of instrumentation code. Their instrumentation frameworks modify the source code prior to execution. They also need WCET analysis after program instrumentation to guarantee that timing constraints are met. While these approaches are sound and effective, the need for running WCET analysis pre- and post-instrumentation reduces the applicability to only hard real-time applications where WCET analysis is common. Furthermore, the previous frameworks also operate on the source code of input programs. Hence, the developer has to

include the source code of all library dependencies of the program that he wants to instrument. Moreover, statically analyzing these library dependencies is impractical. Consider, for instance, the VLC media player [13] v2.0.5 which has approximately 600 000 lines of code and uses libraries with more than three million lines of code. Statically analyzing the source code of a multi-threaded program like VLC along with its library dependencies becomes simply impractical.

In this thesis, we present a time-aware dynamic binary instrumentation methodology [17]. The idea is to enable dynamic instrumentation of program binaries while bounding the overhead of the instrumentation process. The proposed methodology assumes only the availability of the program executable. It requires no preprocessing or WCET analysis of the input program. Time-aware dynamic binary instrumentation is practical and scalable for instrumenting soft real-time applications. It also supports multi-threaded programs.

1.2 Real-time Systems

Real-time systems are rich in extra-functional (non-functional) properties [67, 90]. Extra-functional properties include timing, code size, memory consumption, response time, and communication bandwidth. Various real-time systems have different extra-functional requirements, but they are all time-sensitive. A real-time system must maintain, not only functional correctness, but also timing constraints [29]. The correctness of such systems depends on the results they produce in addition to the time at which these results are produced. In hard real-time systems, missing a deadline can result in system-failure which can be life-threatening in some cases. On the other hand, it is possible to occasionally miss deadlines in soft real-time systems. Missing deadlines in soft real-time systems can lead to performance degradation. Examples of real-time systems are the new generation of airplanes and spacecraft avionics, the braking controller in automobiles, and the vital-sign monitor in hospital intensive-care units. More examples include web servers, media players, high-performance networks, and robotic controllers.

1.3 Program Analysis

Program analysis is an important debugging technique. It is critical for understanding program behavior and optimizing system performance. Examples of program analysis

objectives are function call tracing, time and space profiling, and collecting runtime statistics on instruction and function usage. Many program analysis and profiling tools use instrumentation to extract the required information at runtime. Instrumentation is the insertion of analysis code into the program code to trace the program execution. It is mandatory to maintain the program original behavior after instrumentation so, for example, references and pointers to displaced instructions must be updated appropriately. In general, there exist two instrumentation approaches; hardware based, and software based. Hardware-based tracing methods [74, 82] are known for causing significant perturbation to the program being traced [77]. Also, hardware-based tracing methods collect low-level data and, hence, require higher-level support to provide traces at a higher-level of abstraction [75, 76]. Software-based instrumentation can occur either automatically or manually. Manual instrumentation requires that the developer specify the instrumentation locations [89]. The developer, in this case, to find a bug in some program code, inserts print statements and follows the flow of control or prints the value of variables [89, 59]. This process of adding and removing print statements keeps on going until the developer locates and eventually fixes the bug. Manual instrumentation is highly flexible, but the induced effect of instrumentation on the timing behavior is hard to estimate by the developer. Automatic instrumentation can happen either statically before the program runs or dynamically during program execution. Static instrumentation tools include EEL [64], ATOM [92], Etch [84], and Morph [101]. Dynamic instrumentation frameworks insert analysis code during program execution to extract required information. Examples of such frameworks are DynamoRIO [22], Pin [69], Valgrind [79], and Dyninst [24].

Static instrumentation methods are based on static analysis and cannot react to changes in application behavior at run time. Static instrumentation, generally, incur lower runtime overhead compared to dynamic binary instrumentation (DBI). On the other side, DBI, as opposed to static instrumentation, does not require any pre-processing of the program under analysis. This makes DBI more practical and usable by developers for profiling and tracing purposes. More importantly, DBI can instrument any program while static methods are limited to the code they can analyze, and, for example, cannot instrument dynamically generated code and dynamically loaded libraries.

Instrumentation naturally causes perturbation to the program under analysis. Instrumentation methods [71, 60] insert code in the original program to enable tracing, which results in modifying the programs timing behavior. Since real-time software is time-sensitive and needs to obey timing constraints, real-time systems require specialized program analysis tools.

1.4 Time-aware Instrumentation

Time-aware instrumentation preserves a program logical correctness and respects its timing constraints. Fischmeister et al. [37] introduced time-aware instrumentation by statically instrumenting a program's source code only at code locations that preserve the program's worst-case execution time (WCET). Instrumentation of a program using time-aware instrumentation techniques shifts the program's execution time profile towards its deadline. The authors report in [37] that applying their techniques to a case study resulted in a low instrumentation coverage. The reason was that large portions of the code were shared with the worst-case path and, hence, could not be instrumented. Kashif et al. [56] applied code transformation techniques to the program under analysis to increase instrumentation coverage. The idea involves creating and duplicating basic blocks in a program to increase the locations at which instrumentation code can be inserted while preserving timing constraints. The authors in [55] develop an instrumentation framework, INSTEP, for preserving multiple competing extra-functional properties. Such properties include timing, code size and detection latency. INSTEP uses cost models and constraints of the extra-functional properties together with the user's instrumentation intent to transform the input program into an instrumented program that honors the specified constraints.

1.5 Pin Framework

DIME is implemented as an extension to Pin, which is a DBI framework developed by Intel [69]. Pin provides a cross-platform API for building program-analysis tools. It targets IA-32 and Intel64 architectures, and supports multiple operating systems such as Windows, Linux, OSX, and Android. Intel architectures running any of these operations systems is a common platform for soft real-time applications such as the case studies presented in Sections 3.4 and 4.6. Moreover, Pin is popular and well-supported with over 300 000 downloads and 700 citations [33]. Multiple Intel commercial development tools are built on top of Pin [4]. Intel Parallel Inspector is a memory and threading debugger that can identify memory leaks, allocation errors, data races, and deadlocks. Another commercial tool is Intel Parallel Amplifier, which aid in performance optimization. Additionally, Intel Parallel Advisor is a threading prototyping tool used to analyze and tune program's threading design before implementation. Pin offers the following features:

- Ease-of-use: Pin's user model allows inserting calls to instrumentation code at arbitrary locations in the executable using a simple but rich C/C++ API. Pin has

more than 450 well-documented, easy to use instrumentation APIs [100]. Using Pin, a developer can analyze a program at the instruction level with minimal knowledge about the underlying instruction set.

- Portability: Although Pin allows extraction of architecture-specific information, its API is architecture-independent.

- Transparency: A program instrumented by Pin maintains the same instruction and data addresses, and the same register and memory values compared to uninstrumented execution. Thus, Pin extracts information that correctly describes the program's original behavior.

- Efficiency: Pin uses a just-in-time (JIT) compiler to insert and optimize instrumentation code. It utilizes a set of dynamic instrumentation and optimization techniques; such as code caching, trace linking, register reallocation, inlining, liveness analysis, and instruction scheduling.

- Robustness: Since Pin discovers the code in runtime, it can handle statically unknown indirect-jump targets, dynamically generated code, dynamically loaded libraries. Also, it can analyze mixed code and data, and variable-length instructions.

To build an analysis tool using Pin, the developer should create a *pintool* which basically consists of two types of routines. The analysis routine contains the code to be inserted in the program during execution, whereas the instrumentation routine decides where to insert the analysis-routine calls. Pin injects to the program executable and uses a JIT compiler to translate the executable, instrument it, and retain control of it. The unit of compilation is the *trace*: a straight-line code sequence that ends in an unconditional control transfer, a predefined number of conditional control transfers, or a predefined number of instructions. When the program starts execution, Pin compiles the first trace and generates a modified one. The generated trace is almost identical to the original, but it enables Pin to regain control. Pin transfers control to the generated trace, then Pin regains control when a branch exits the trace. Afterwards, Pin compiles the new trace and continues execution. Whenever the JIT compiler fetches some code to compile it, the pintool is allowed to instrument the code before compilation. Pin saves the compiled code and its instrumentation in a code cache in case it gets re-executed [95, 86, 69].

Pin supports different granularities for the instrumentation routine; image, trace, routine, and instruction granularity. The instrumentation-routine granularity defines when Pin should execute the instrumentation routine. For example, in instruction granularity, Pin instruments the program a single instruction at a time. Similarly, Pin offers

multiple analysis-routine granularities i.e., where to insert the analysis-routine call. The instrumentation-routine and the analysis-routine granularities can be different. For instance, a pintool can support trace granularity for the instrumentation routine using the TRACE_AddInstrumentFunction() API. That means the pintool can access the basic blocks and the instructions inside the trace using BBL_InsertCall() and INS_InsertCall(), respectively. Note that a pintool can have multiple instrumentation and analysis routines.

Most of Pin's overhead originates from the execution of the instrumentation code (in the analysis routines). Such overhead varies according to the invocation frequency of the analysis routines and their complexity. On the other hand, dynamic compilation and insertion of instrumentation code (by the instrumentation routine) represent a minor source of the overhead [69].

The following numbers show the slow-down factors of the SPEC2006 benchmark running on top of Pin on a Windows 32-bit platform as reported by Devor in [33]:

- *Instruction-counting tool* (inscount): 2.45 and 1.56 for SPECint and SPECfp, respectively.
- *Memory-tracing tool* (memtrace): 4.74 for SPECint and 3.26 for SPECfp.
- *Memory-tracing tool using Pin-buffering API* (membuffer): 4.64 and 3.2 for SPECint and SPECfp, respectively.

The instruction-counting tool counts the executed instructions of every executed basic-block. The memory-tracing tool collects the address trace of instructions that access memory. Additionally, Luk et al. report in [69] that the application slow down due to dynamic instrumentation by Pin [69], DynamoRIO [22], and Valgrind [79] are 2.5, 5.1, and 8.3 times, respectively. These numbers are reported for a light-weight tool counting basic-block using an IA32 Linux platform running the SPECint benchmark.

Multiple factors can affect the overhead of DBI frameworks such as the complexity of the analysis tool, the application's complexity, and the length of the application's execution time. For example, Intel internally uses a heavy-weight Pin analysis tool that performs sophisticated memory analysis on Intel's production applications to analyze memory reference behavior. As stated by the authors in [95], this tool incurs average slow down of 38 and maximum slow down of approximately 110 for SPECint. Also, Valgrind's memcheck tool introduces average overhead of 22.2 and maximum overhead of 57.9 for SPEC2000 benchmark [79]. Memcheck is a complicated analysis tool that detects uses of undefined values.

We implement DIME extensions for supporting time-aware instrumentation in Pin. As mentioned before, Pin is easily extensible for the creation of program analysis tool. Moreover, Pin has lower overhead compared to other similar tools which is an essential feature to achieve time-aware instrumentation.

1.6 Rate-based Resource Allocation

DIME employs rate-based resource allocation technique to bound the runtime overhead of the instrumentation process. Rate-based resource allocation is a widely known methodology for priority scheduling of tasks [53, 91, 14]. Different schemes of rate-based resource allocation have been proposed for various computer-system domains. Examples of such domains are operating systems [53], real-time systems [52] including mixed-criticality ones [15], and multimedia applications [51]. A rate-based system allows a specific task to run according to a previously defined rate, for example, *"x milliseconds per second"*. The system has a capacity (i.e., a budget) for the task to execute in each time period (T). Once the budget is consumed, the task is suspended until the next time period (T) starts.

Rate-based resource allocation models are flexible in managing tasks that have unknown or varying execution times. They can also deal with tasks whose execution times or execution rates can significantly deviate at run-time from the expected behavior. They enable direct mapping of timing and importance constraints into priority values. Rate-based resource allocation models can protect a real-time system from performance degradation. These models guarantee full resource utilization and overload-avoidance in a resource-constrained system.

1.7 Goals and Contributions

In this thesis, we investigate the time-aware dynamic binary instrumentation technique for the analysis of soft real-time systems. We develop the proposed technique into a fully-implemented analysis tool and study its ability to respect a system's extra-functional constraints, especially timing. Additionally, the thesis evaluates the runtime overhead and the instrumentation coverage of time-aware dynamic instrumentation. The following is a summary of our contributions:

- Introducing the concept of time-aware dynamic binary instrumentation by employing rate-based resource allocation method [17]. Dynamic instrumentation relaxes the

assumptions of time-aware instrumentation to increase its scalability and applicability to soft real-time systems.

- Developing DIME: a tool for the proposed instrumentation technique [17]. DIME is a fully-implemented time-aware dynamic binary instrumentation framework that limits the instrumentation time to a pre-specified budget. DIME bounds the runtime overhead of the instrumentation process to respect the timing properties of programs.

- Proposing a redundancy-suppression technique to increase the instrumentation coverage of DIME. To promote the ability of DIME to collect quality information, it utilizes its budget, when applicable, to only extract non-redundant traces during instrumentation.

- Studying the optimization possibilities of DIME's parameters. The operation of the time-aware dynamic instrumentation is highly dependent on two parameters. Defining the relation between the performance of DIME and its parameters offers practical guidance for the parameter tuning of the instrumentation framework.

- Introducing the concept and the tool for quality-of-service-aware dynamic binary instrumentation (QDIME) [19]. Being a customizable and feedback-based analysis tool, QDIME respects the user-defined performance thresholds and constraints to guarantee an acceptable performance during instrumentation.

1.8 Organization

This thesis is organized as follows. Chapter 2 presents an overview of the related work on program instrumentation and analysis. Chapter 3 proposes DIME: a time-aware dynamic binary instrumentation framework using rate-based resource allocation. Chapter 4 presents the redundancy suppression in time-aware dynamic binary instrumentation with the goal of increasing the instrumentation coverage. Moreover, Chapter 5 discusses the parameter tuning experiments, along with their implications on DIME. This chapter, also, lists the limitations, the guidelines, and the work flow of DIME. Chapter 6 presents a QoS-aware dynamic binary instrumentation technique that respects the performance constraints of QoS systems. Finally, Chapter 7 concludes the thesis and discusses the potential future work.

Chapter 2

Related Work

In this chapter, we review the related work on different instrumentation methodologies. The related research topics include both static and dynamic binary instrumentation; DBI tools utilizing Pin, and time-aware instrumentation techniques.

2.1 Static Instrumentation Frameworks

Static binary instrumentation frameworks provide libraries and APIs to modify program binaries offline before execution. Examples of such frameworks are EEL [64] and ATOM [92]. Another example is Etch [84], a binary rewriting framework for Win32 applications running on Intel x86 processors. QPT is a program profiler, based on static binary instrumentation, that measures the execution frequency of basic blocks and control flow [65]. Morph [101] also is a static binary instrumentation system that optimizes program executable based on the collected profile information. Static binary instrumentation is unable to handle dynamic code features such as dynamically generated code, dynamically loaded libraries, and unrestricted indirect jumps and calls [42].

2.2 Dynamic Instrumentation Frameworks

Dynamic binary instrumentation (DBI) tools use code transformation during program execution to extract program profile. Dyninst [24] is an API that provides a library for dynamic binary instrumentation. Dyner [98], based on Dynisnt, is a platform-independent

interactive tool that attaches to a running program, and allows the developer to insert and remove instrumentation. MDL is a machine-independent language for dynamic binary instrumentation [48]. It allows specification of the instrumentation and the collected performance information. SystemTAP [50], Kerninst [93] and DTrace [27] provide interfaces for kernel instrumentation. More examples of DBI systems include GDB, Vulcan [35], and Detours [49]. Most of these dynamic binary instrumentation systems are probe-based, and consequently, suffer from transparency issues [23, 69]. In probe-based instrumentation, original instructions in memory are overwritten by the instrumentation code. This leads to modifying the native behavior of the program under analysis.

Other DBI tools preserve transparency using software code caches such as Valgrind [79], DynamoRIO [22], and Pin [69]. DynamoRIO [22] is a powerful runtime code manipulation system that dynamically instruments an executable. An unmodified application can be monitored and controlled in DynamoRIO's virtual execution environment. The application's code and the inserted code are interleaved together. DynamoRIO provides an interface for creating customized analysis tools by abstracting away the details of the underlying system. Valgrind [79] is a DBI framework for building customized heavyweight analysis tools. It uses just-in-time (JIT) binary re-compilation to instrument code blocks. Popular Valgrind analysis tools are Memcheck; a memory error detector, callgrind; a call-graph generator , and CacheGrind; a cache and branch-prediction profiler. Valgrind alters the execution of multi-threaded applications. It allows only one thread to run at a time, and consequently, changes the application's thread scheduling. Pin [69] provides a high-level API for dynamic binary instrumentation. It uses a JIT compiler to insert and optimize instrumentation code. Pin automatically embraces register reallocation, inlining, liveness analysis, and instruction scheduling to optimize jitted code. On the other hand, Valgrind and DynamoRIO rely on the tool writer to invoke special operations to boost performance (e.g., inlining). The program instrumented by Pin maintains the same instruction and data addresses, and the same register and memory values compared to native execution.

2.3 Pin-based DBI Frameworks

Due to Pin's efficiency and ease-of-use, many research studies have utilized Pin for specific analysis requirements, or to reduce DBI overhead in general [85, 28, 72, 68]. Wallace and Hazelwood [97] introduce SuperPin; a parallelized version of Pin to reduce the overhead of DBI. SuperPin executes an uninstrumented version of the application, and then forks off multiple instrumented slices of code regions. Each slice runs in parallel to the application in a separate processor core. The experimentation shows significant performance improve-

ment for instrumentation tasks that are amenable to parallelization and merging. The performance of SuperPin highly depends on the number of processor cores, and available memory. Moreover, there exists multiple sources of delay, such as fork overhead, pipeline delay and compilation slowdown. Moseley et al [76] use a probe-based application monitor to fork a shadow process that is to be instrumented and profiled. The forked process runs in parallel to the original with certain restrictions to prevent interference with the execution of the original process. Other works as well parallelize the program profiling and analysis process by utilizing multicore systems [102, 99]. As will be illustrated in Chapter 3, DIME is a generic approach to time-aware dynamic binary instrumentation. Hence, DIME can be extended to utilize these parallelization techniques.

Upton et al. [96] reduce the data-collection overhead of system profiling. They implement a buffering system for Pin to efficiently collect chunks of data and process the full chunk at once. The buffering system optimizes the generated code for buffer writing and reduces the cost of full-buffer detection. Kumar et al. [63] optimize the instrumentation code to decrease the DBI overhead. They reduce the number of executed instrumentation points and the cost of each point. These instrumentation approaches, opposed to DIME, ignore the program's timing and performance constraints.

Arnold and Ryder [20] introduced a framework for reducing the cost of instrumented code. They use code-duplications combined with counter-based sampling to switch between instrumented and non-instrumented code. Checking code is inserted at method entries and backedges. Thus, their profiling approach is limited to intra-procedural acyclic program paths. Also, this approach does not take into account the execution time of the instrumentation code. Although event-based sampling is an effective way of instrumenting events according to their frequency of occurrence, overhead bursts can have a negative effect on the performance of time-sensitive applications [21]. Other sampling-based techniques have been proposed for performance optimizations [40]. These techniques either apply optimizations specific to the instrumentation objective or use compiler-specific information to perform optimizations.

PinOS [25] is a Pin extension for instrumenting not only the user-level code but also the kernel OS. It is built on top of the Xen virtual machine monitor, and it inherits the powerful instrumentation Pin API. PinOS fails to provide strong isolation; although it steals the memory from the guest OS, the instrumented process is still able to access the memory used by the analysis routines. PEMU [100] is another Pin-compatible kernel and user-space DBI framework. It supports out-of-VM high-level instrumentation of Kernel operations. PEMU provides an additional software layer to maintain the isolation requirement. Again, the techniques mentioned in this section, generally, can be used to complement DIME approach for time-aware instrumentation.

2.4 Static Time-aware Instrumentation

All the previously mentioned instrumentation approaches are poor at maintaining the extra-functional properties of the program. They affect the program's behavior especially the temporal behavior [56]. Partial instrumentation can be used to respect timing constraints [88]. Fischmeister et al. [39, 37] present time-aware static source-code instrumentation to honor the program's timing, especially the worst-case execution time (WCET). This means adding instrumentation code only to non-worst-case paths of the program. Time-aware instrumentation shifts these paths to have higher execution times closer to the program's deadline. The case studies, investigated in [37], show the promise of the general concept of time-aware instrumentation, but also, reveals new challenges. A relatively high percentage of the program paths shares basic blocks with the worst-case path. This prevents the instrumentation of large portions of the code to avoid the violation of the worst-case execution time constraint.

Kashif and Fischmeister [56] apply program transformation techniques to tackle this challenge and increase the effectiveness of time-aware instrumentation. The first program transformation technique is branch block creation which creates locations in the program for instrumentation. This increases the number of instrumentable basic blocks which leads to a higher instrumentation coverage. The second technique is control-flow-graph (CFG) Cloning that duplicates CFG sub-graphs to permit instrumenting them. CFG Cloning adds no overhead to the worst-case path at the expense of code size. In [57], Kashif et al. introduce a slack-based mechanism which allows conditional instrumentation of the worst-case path of the program. The timing constraints of hard real-time systems are usually conservative. Thus, there exists a time slack between the WCET and the actual execution time of the program [57]. Slack-based conditional instrumentation will execute the instrumentation code only if the program has sufficient slack at run time.

INSTEP [55] is a fully implemented static instrumentation framework that considers multiple competing extra-functional properties such as timing and code size. The inputs of INSTEP are the instrumentation intents, the extra-functional constraints, the cost models, and the program source code. INSTEP derives instrumentation alternatives and applies local search to optimize the instrumentation solution. Denil et al. [32] present an instrumentation framework for Simulink models that preserves extra-functional properties. Similar to INSTEP, the proposed framework considers the instrumentation intents and the cost models to respect the system's constraints. The authors use rule-based model transformation techniques to instrument the model and optimize the placement of instrumentation blocks.

Static time-aware instrumentation methods are sound and effective for hard real-time

systems. On the other side, the mentioned methods adopt strict assumptions [94]: source-code availability, static-analysis preprocessing, WCET analysis, and MISRA-C compliance. The availability of the source code is needed to conduct static-analysis preprocessing and WCET analysis. Not only the source code of the program should be available, but also the source code of all the referenced libraries. Practically, the source code of the libraries may not be available or accessible. Also, the source code of many libraries are large in size; statically analyzing them is complex and may be impractical. WCET analysis is a common, and sometimes a mandatory, practice for the development of hard real-time systems. However, the requirement of running WCET analysis before and after the instrumentation reduces the applicability of these methods to only hard real-time systems. Finally, static time-aware instrumentation assumes the program source code to be analyzable. For example, the program should be MISRA-C compliant. MISRA-C provides a standard for the development of safety-critical real-time systems and facilitates the computation of WCET. MISRA-C limits the use of pointers, recursion and dynamic memory allocation.

The dynamic technique proposed in this thesis loosens the assumptions of time-aware instrumentation to increase its applicability and scalability. Dynamic time-aware instrumentation only assumes the availability of the executable binary. It requires no preprocessing or WCET analysis before or after instrumentation. Also, the dynamic technique removes any restrictions on the program structure. However, dynamic instrumentation is known to incur higher runtime overhead than static instrumentation. Therefore, Dynamic time-aware instrumentation considers only soft real-time systems which can tolerate a few deadline misses.

2.5 Program Sampling

Sampling is a widely used approach to reduce the runtime overhead of program tracing by trading off the amount of extracted information [26, 16, 20, 47, 45, 70, 58, 66, 38]. Sampling extracts runtime information at a predefined rate. Mostly, sampling is triggered by hardware interrupts, periodic software-event counters, or both.

The authors in [26] employ the timer-based interrupts on Alpha processors to collect register contents. The performance counters interrupt the processor to record values from the current context. They use a sampling period of fixed number of instructions in addition to small randomization factor. Similarly, the Digital Continuous Profiling Infrastructure (DCPI) [16] is a sampling profiler that utilizes hardware performance counters. The sampling period of DCPI is based on frequent randomized periodic interrupts.

As mentioned previously, Arnold and Ryder [20] uses instrumentation along with counter-based sampling to collect program profile with low overhead. This method only captures the temporal profile of intra-procedural acyclic program paths. Hirzel and Chilimbi [47] further extend Arnold-Ryder work to sample longer program bursts allowing for low-overhead temporal profiling. Their bursty tracing framework can span procedure boundaries, and accordingly, can collect inter-procedural profiling information.

Adaptive bursty tracing (ABT) is a statistical profiling mechanism that aims to maximize the coverage of infrequently executed code, i.e., cold code [45]. This tracing method samples code segments at a rate inversely proportional to their execution frequency. ABT reduces the sampling frequency and overhead of hot code and guarantees the tracing of cold code. Marino et al. [70] present a sampling algorithm, for multi-threaded applications, which progressively adapts the sampling period at runtime based on the code execution frequency. Kasikci et al. [58] propose the bias-free sampling (BFS) method which uses breakpoints to sample instructions independently of their execution rate. BFS is an event-based sampling approach which samples infrequently executed instructions. The proposed approach avoids profiling hot instructions and, thus, limits the runtime overhead.

With the objective of bug isolation, Liblit et al. [66] propose a sampling framework to gather runtime information from remote user-executions of deployed software. The authors extend Arnold and Ryder's work mentioned earlier to collect sample data. They randomize the sampling period, instead of using a fixed rate, based on a geometric distribution to guarantee a fair random sample. Contrary to Arnold-Ryder approach, this one can collect inter-procedural profiling information. Finally, Fischmeister and Ba [38] introduce a sampling-based monitoring technique for time-sensitive applications along with techniques to statically determine sampling periods.

Both DIME and the sampling method trade off the overhead and the coverage of program tracing. Similar to sampling, DIME defines a time period to collect runtime information. Additionally, DIME specifies an instrumentation budget to respect the program's extra-functional constraints (especially timing). The budget determines the time allowed for program instrumentation. Although the mentioned sampling techniques aim to reduce the overhead of program profiling, none takes the program constraints into consideration. Also, the definition of the *period* in DIME is different from that of the sampling approach. The *sampling period* denotes the time between collecting two consecutive samples. On the other hand, the *time period* of DIME defines the time between consecutive budget-replenishment events (will be discussed later in Chapter 3). Finally, DIME is a generic instrumentation approach that is independent of the analysis objective, e.g., call tracing, opcode profiling, or branch profiling. In other words, the instrumentation budget of DIME is defined with respect to the system specifications and regardless of the analysis objective.

Chapter 3

DIME: Time-Aware Dynamic Binary Instrumentation

In this chapter, we illustrate DIME, a time-aware dynamic binary instrumentation (DBI) framework [17]. DIME is built as an extension to Pin [69]. The idea is to enable dynamic instrumentation of program binaries while still bounding the overhead of the instrumentation process. DIME supports multi-threading and is a practical and scalable tool for instrumenting soft real-time applications.

3.1 Overview of DIME

DIME is a dynamic time-aware binary instrumentation tool [17]. It ensures that the instrumentation process respects, as much as possible, the timing properties of the program. DIME achieves this using rate-based resource allocation [53] by limiting the instrumentation time to a predefined budget B per time period T. The instrumentation budget B is specified during the system design process. Instrumentation code executes for a total of t_{ins} time units in every time period T. Optimally, the total instrumentation time t_{ins} per period T should not exceed the instrumentation budget B. If the instrumentation consumes the given budget before the end of the time period T, the framework will disable instrumentation. At the beginning of the next period T, the budget resets to B time units and the instrumentation is re-enabled. This process repeats until the program terminates.

Figure 3.1 describes the operation of DIME. There are two states of operation: *DBI-enabled* and *DBI-disabled*. In the first state, DIME can insert instrumentation code and

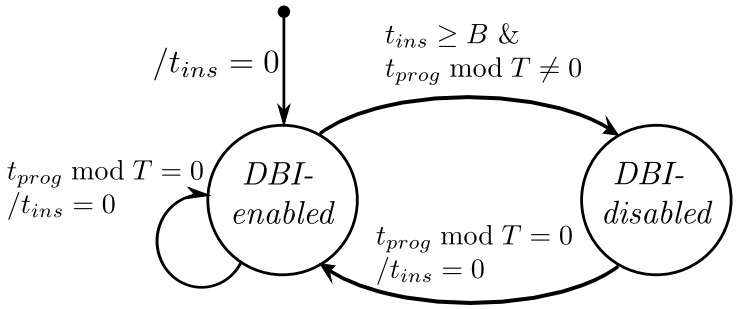


Figure 3.1: State-machine for DIME's operation.

extract information from the program during its execution. Furthermore, DIME has to measure the time consumed by the executed instrumentation code t_{ins}. In the second state, the framework prohibits code insertion and, thus, program instrumentation. DIME switches between the two states: *DBI-enabled* and *DBI-disabled* according to the consumption of the instrumentation budget. Let t_{prog} be the running time of the program since the start of execution. DIME will switch from *DBI-enabled* to *DBI-disabled* when the instrumentation consumes all its budget ($t_{ins} \geq B$) and a new period T has not yet started ($t_{prog} \bmod T \neq 0$). At the beginning of every time period T, i.e., ($t_{prog} \bmod T = 0$), DIME will reset the instrumentation time, $t_{ins} = 0$. It would also switch states from *DBI-disabled* to *DBI-enabled* if the instrumentation was disabled.

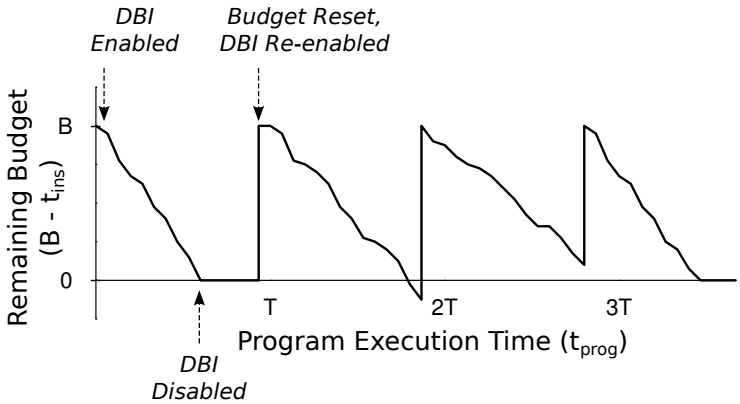


Figure 3.2: Rate-based DBI.

Figure 3.2 further illustrates the rate-based DBI approach. The X-axis represents the program's execution time t_{prog} and the Y-axis shows the remaining instrumentation budget ($B - t_{ins}$). The program launches in the *DBI-enabled* state where the framework

has full instrumentation budget. In the first time period $[0, T)$ of the program's execution, instrumentation code executes and reduces the available budget. Once the instrumentation has fully consumed the budget, the framework will switch to the *DBI-disabled* state and will prohibit instrumentation. At time T, the budget is reset and the framework returns back to the *DBI-enabled* state. The negative value in the second time period $[T, 2T)$ is an overshoot. An overshoot will occur, if, for instance, the remaining budget equals two time units, but the last instrumentation (before switching to *DBI-disabled* state) takes three time units. In the third time period $[2T, 3T)$, the total budget consumption is less than the budget B, so the framework remains in the *DBI-enabled* state.

3.2 Implementation Using Pin

This section describes the implementation of DIME. We mentioned, in Section 1.5, that instrumentation routines and dynamic compilation have a small overhead compared to the execution of the instrumentation code in the analysis routines [69]. This means that the main source of DBI overhead using Pin is the analysis routines. Hence, our objective is bounding the overhead of the analysis-routine execution to the pre-specified budget. The total instrumentation time t_{ins}, in this case, is approximately the total execution time of the instrumentation code inside the analysis routine. The time-aware extensions can be applied to any pintool, i.e., a pintool created by a developer for any instrumentation objective can be modified to support time-aware DBI.

```
void analysis(...){
    time_start = get_time();
    //Execute instrumentation code
    ...
    time_end = get_time();
    budget_var -= time_end - time_start;
}
...
void instrumentation(...){
    //Do budget checking
    ...
    //Switch state accordingly
    ...
    //Insert analysis calls based on state
    ...
}
...
void sig_handler(...){
    budget_var = B;
}
```

Listing 3.1: Handling of instrumentation budget in DIME.

Listing 3.1 shows a pseudocode that abstractly describes the handling of the instrumentation budget in DIME. A variable budget_var initially holds the maximum allowable instrumentation budget (per period T), B. A signal *sig* is scheduled to fire every T time units. In the *sig* signal-handler routine, the instrumentation budget is reset i.e., budget_var is reset to B. The function get_time() is responsible for reading the current timestamp using precise timers such as the time stamp counter (TSC) or the high precision event timer (HPET). Section 3.3 discusses the exact implementation of get_time(). DIME measures the instrumentation time by subtracting the timestamps at the beginning and end of the analysis routine. It then subtracts the instrumentation time from budget_var. The instrumentation routine performs a budget check to determine the state (*DBI-enabled* or *DBI-disabled*). If instrumentation budget is available, then the instrumentation routine will insert a call to the analysis routine, otherwise it does not. The instrumentation routine usually inserts calls to the analysis routine(s) based on the instrumentation objective. Consider, for instance, a pintool that prints memory addresses of program reads or writes. The instrumentation routine then checks for the type of instructions. If the instruction reads or writes to memory, then the instrumentation routine will insert a call to the analysis routine and will instruct Pin to pass the memory reference's effective address to the

analysis-routine call.

The instrumentation routine in DIME performs the following operations:

1. Checking for instrumentation budget
2. Inserting calls to the analysis routine
3. Processing before and/or after inserting analysis routine calls

Note that budget checking is the only difference between the instrumentation routine in DIME and that in native Pin (unmodified pintools). Optimally, the instrumentation routine should be able to:

1. Incur minimal overhead in the *DBI-disabled* state
2. Honor the instrumentation budget, i.e., disable instrumentation once budget_var reaches zero
3. Guarantee full utilization of the budget, i.e., re-enable instrumentation once the signal fires

Checking for budget at each instrumentation point in the *DBI-enabled* state enables strictly honoring the instrumentation budget. Budget checking at each instrumentation point in the *DBI-disabled* state allows a quick transition to the *DBI-enabled* state, i.e., allows instrumentation to start from the beginning of period T and, hence, full utilization of the budget. On the other hand, continuous budget checking adds runtime overhead. Reducing the frequency of budget Checking by performing it at a higher granularity (not at each instrumentation point) in the *DBI-disabled* state, will reduce the budget checking overhead in that state at the expense of budget utilization. Checking for budget at a higher level in both states, will reduce the overall budget checking overhead, but will cause overshoots beyond the instrumentation budget to often occur as a result of loosely honoring the budget. These tradeoffs will be highlighted in the discussion of the different implementation alternatives in Section 3.2.4.

There are three different implementations for the instrumentation routine in DIME: *Trace Version*, *Strict Trace Version*, and *Trace Conditional*. From an implementation point of view, the way DIME performs budget checking is the main difference amongst the three implementations.

3.2.1 *Trace Version*

Trace Version checks for budget at each instrumentation point and makes use of Pin's *trace versioning* APIs to enable and disable instrumentation. Pin's trace versioning APIs allow dynamic switching between multiple types (versions) of instrumentation at runtime. We mentioned earlier that a trace in Pin is defined as a sequence of program instructions that has a single entry point and may have multiple exit points. If Pin detects a jump to an instruction in the middle of a trace, it will create a new trace beginning at the target instruction. When Pin switches versions, it creates a new trace starting from the current instruction.

Trace Version inserts analysis calls to check for budget and switch instrumentation versions (if necessary) at each instrumentation point. In *Trace Version*, every trace is assigned a version ID, either V_INSTRUMENT, which represents the *DBI-enabled* state, or V_BASE for the *DBI-disabled* state. Listing 3.2 shows a pseudocode outline of the *Trace Version* implementation that favors readability over optimality. Pin calls the instrumentation routine at every new trace. First, budget checking is performed by inlining an extra analysis routine (budget_check() routine). The budget_check() routine consists of one code statement: return (budget_var > 0). Note that Pin is capable of inlining short routines that have no control flow. If the current trace version is V_BASE, the instrumentation routine will check if it needs to switch the version to V_INSTRUMENT and vice versa. The instrumentation routine performs this check by inserting a dynamic check using Pin's API InsertVersionCase(). This API will set the trace version to V_INSTRUMENT if the inserted call to budget_check() returns true, and will set it to V_BASE otherwise. Finally, the switch case will insert a call to the analysis routine only if the current version is V_INSTRUMENT.

```
1  void instrumentation(...){
2       For each instrumentation point{
3           if(version == V_BASE) {
4               InsertCall(budget_check);
5               //check switching to V_INSTRUMENT
6               InsertVersionCase(1,V_INSTRUMENT);
7           }
8           else if(version == V_INSTRUMENT){
9               InsertCall(budget_check);
10              //check switching to V_BASE
11              InsertVersionCase(0,V_BASE);
12          }
13          switch(version) {
14              case V_BASE:
15                  break; //Do Nothing
16            case V_INSTRUMENT:
17                  ...
18                  InsertCall(analysis_routine);
19                  ...
20                  break;
21          }
22      }
23 }
```

Listing 3.2: Instrumentation routine of *Trace Version*.

To clarify, let Trace_1 be the sequence of instructions in Listing 3.3. Assume that Trace_1 has version = V_INSTRUMENT, and "For each instrumentation point" means "For each instruction" in this example. Pin calls the instrumentation routine at every trace. The instrumentation routine inserts an inlined call to budget_check() before every instruction in the trace. According to the switch case in Listing 3.2, the instrumentation routine inserts a call to the analysis routine before every instruction in the trace. The API InsertVersionCase() guarantees that the execution of the inserted analysis routine will occur, only if the output of budget_check() matches the ID of the current version (which is 1 in this case). Directly after the execution of the instrumentation routine, budget_check(), that is inserted before the first instruction, is executed. Assume that its output is 1 which means that the budget is currently larger than zero. In this case, Pin will execute the analysis routine that is inserted before the first instruction. Afterwards, Pin executes budget_check() that is inserted before the second instruction. Assume that the budget is now fully consumed, so budget_check() returns 0. Since the output mismatches the ID of the current trace, Pin will switch ver-

sion to V_BASE (i.e. disable instrumentation). Accordingly, Pin will create a new trace Trace_2, with version = V_BASE, starting from the second instruction. Pin will then execute the instrumentation routine to instrument Trace_2 according to its version. Also, the analysis routine inserted before the second and the third instruction will be ignored i.e., not executed.

```
1 mov eax, dword ptr [rsp+0x30]
2 and eax, 0x10000000
3 mov dword ptr [rsp+0x3c], eax
```

Listing 3.3: Example (1) of a trace.

DIME achieves low overhead by using Pin's InsertVersionCase() API. When the instrumentation routine calls the API InsertVersionCase() and switches versions, Pin creates a new trace starting from the currently-executing instruction. In the *Trace Version* implementation, DIME continuously calls InsertVersionCase(), forcing Pin to continuously check for the trace version. These checks can be inserted at every instruction, at the start of every basic block, etc. according to the instrumentation objective. Thus, the low overhead trace version API enables DIME to always check for available budget at each instrumentation point. This enables immediate transitions to both *DBI-enabled* and *DBI-disabled* states. This way it guarantees full utilization of the budget by switching to V_INSTRUMENT from V_BASE once budget is available for instrumentation. This also causes a high budget checking frequency.

3.2.2 *Strict Trace Version*

Strict Trace Version has a similar implementation to *Trace Version* but with a reduced frequency of budget checking in the *DBI-disabled* state. This means a lower budget checking overhead and at the same time a delayed transition to the *DBI-enabled* state. A delayed transition to the *DBI-enabled* state means a delay in enabling instrumentation which reduces the budget utilization. The *Strict Trace Version* implementation of DIME also makes use of Pin's trace versioning APIs.

In the *DBI-disabled* state, *Strict Trace Version* checks for a budget reset in the instrumentation routine to switch to the *DBI-enabled* state instead of performing analysis-routine calls to check for budget at each instrumentation point. Listing 3.4 outlines the implementation of *Strict Trace Version*. A boolean variable budget_reset is introduced that is initially set to false. This variable will be set to true when the signal *sig* fires every period T. The instrumentation version is initially V_INSTRUMENT and will switch to V_BASE when

the instrumentation budget runs out during the period T. In V_INSTRUMENT, budget checking happens at each instrumentation point as in *Trace Version* implementation. In the version V_BASE, the instrumentation routine does not insert a check for switching versions until the budget is reset and variable budget_reset is set to true. This modified condition prevents the instrumentation routine from calling InsertVersionCase() in the *DBI-disabled* state. To illustrate, DIME checks the budget in the *DBI-enabled* state at each instrumentation point (e.g. at the instruction level). However, when the state changes to *DBI-disabled*, the budget checking will only happen when the instrumentation routine gets called, i.e., at the trace level. This reduces the budget checking overhead compared to *Trace Version* which performs budget checks more often.

Strict Trace Version reduces the budget checking overhead at the expense of budget utilization. Remember that inserting calls to InsertVersionCase() makes Pin check for the trace version and create a new trace. Consider the scenario when one call to InsertVersionCase() switches the version from V_INSTRUMENT to V_BASE. This causes the creation of a new trace. When DIME calls the instrumentation routine for the new trace and budget_reset is false, DIME will not insert version checks or analysis-routine calls. Hence, the trace will run to completion without any version switches even if the budget gets reset. When Pin creates a new trace and budget_reset is true, the instrumentation routine will insert a version check that will trigger switching versions to V_INSTRUMENT. In other words, *Strict Trace Version* has lower budget utilization compared to *Trace Version* because it postpones using the instrumentation budget till the start of a new trace.

```
1  void instrumentation(...){
2     For each instrumentation point{
3         if(version == V_BASE  &&  budget_reset == true) {
4             budget_reset = false;
5             InsertCall(budget_check);
6             //check switching to V_INSTRUMENT
7             InsertVersionCase(1,V_INSTRUMENT);
8         }
9         else if(version == V_INSTRUMENT){
10            InsertCall(budget_check);
11            //check switching to V_BASE
12            InsertVersionCase(0,V_BASE);
13        }
14        switch(version) {
15            case V_BASE:
16                break; //Do Nothing
17            case V_INSTRUMENT:
18                ...
19                InsertCall(analysis_routine);
20                ...
21                break;
22        }
23    }
24 }
25
26 void sig_handler(...){
27    budget_var = B;
28    budget_reset = true;
29 }
```

Listing 3.4: Implementation of *Strict Trace Version*.

3.2.3 *Trace Conditional*

The *Trace Conditional* implementation aims to reduce the budget checking overhead associated with trace versioning. It avoids calling analysis routines for either budget checking or version switching. It, however, suffers from a delayed switching between the *DBI-enabled* and *DBI-disabled* states. This causes DIME to overshoot frequently beyond the instrumentation budget.

```
void instrumentation(...){
    if(budget_var > 0){
        ...
        InsertCall(analysis);
        ...
    }
}
```

Listing 3.5: Instrumentation routine of *Trace Conditional*.

Trace Conditional performs all its budget checking in the instrumentation routine without any analysis-routine calls. Listing 3.5 presents the implementation for *Trace Conditional*. The instrumentation routine checks the available budget using a simple if statement at the beginning of every trace. So, in both states, *DBI-enabled* and *DBI-disabled*, budget checking occurs at the trace level. This decreases the overhead of the instrumentation routine of *Strict Trace Version*. It also reduces the overhead of analysis calls for the purposes of checking budget and switching instrumentation versions. *Trace Conditional* achieves this, however, at the expense of budget utilization because switching to the *DBI-enabled* state only occurs at the beginning of a new trace (similar to *Strict Trace Version*). *Trace Conditional* also loosely honors the instrumentation budget compared to *Trace Version* and *Strict Trace Version*. This is because switching to the *DBI-disabled* state only occurs at the beginning of a new trace.

3.2.4 Qualitative Comparison

Table 3.1 provides a qualitative comparison of the different implementations of DIME. The *Trace Version* and *Strict Trace Version* implementations are based on Pin's trace versioning APIs. This enables them to switch between versions based on analysis-routine calls for budget checking. *Trace Conditional* works differently as it performs budget checks in the instrumentation routine. *Trace Version* checks for budget before each analysis-routine call. Hence, it immediately switches to the *DBI-disabled* state after a budget check returns false (no budget available) before an instrumentation point. It also immediately switches to the *DBI-enabled* state and executes an instrumentation point when a budget check returns true. This enables *Trace Version* to fully utilize the budget and strictly honor the instrumentation budget but with a high budget checking overhead (relative to *Strict Trace Version* and *Trace Conditional*). *Strict Trace Version* delays the switching to the *DBI-enabled* state until the beginning of a new trace. This results in a lower utilization of the budget, strictly honoring the budget, and less budget checking overhead. *Trace*

Table 3.1: Qualitative comparison of the three implementations of DIME.

	Trace Version	***Strict Trace Version***	***Trace Conditional***
Analysis-routine call for budget checking	Yes in both states	Only in *DBI-enabled* state	No
Switch to *DBI-disabled* state	Immediate	Immediate	Delayed till start of new trace
Switch to *DBI-enabled* state	Immediate	Delayed till start of new trace	Delayed till start of new trace
Budget utilization	Full	Waits till start of new trace	Waits till start of new trace
Honoring the budget	Strict	Strict	Loose
Budget checking frequency	High	Medium	Low

Conditional delays switching to both states but does not perform any analysis calls for budget checking. Hence, it has a low budget utilization, does not strictly respect the budget, and has the least budget checking overhead.

3.3 Performance Evaluation

The qualitative evaluation of Section 3.2.4 is insufficient to decide when to use each of the three implementations. For example, a delayed switch to the *DBI-disabled* state can cause overshoots beyond the instrumentation budget until a new trace starts. The severity of the overshoots compared to the budget checking overhead depends on factors like the complexity of the analysis routine and the program behavior. Therefore, it is important to consider these factors to be able to decide on a suitable implementation of DIME to use for a specific instrumentation objective. We now empirically investigate the different implementations in terms of execution overhead, and later in Section 3.4 discuss the instrumentation coverage and applicability domains.

3.3.1 Experimental Setup

We experiment with the SPEC2006 C benchmark suite [46] which consists of integer and floating point benchmarks. SPEC2006 is a common benchmark for evaluating performance of dynamic instrumentation tools [69, 22, 79]. We run the benchmarks on an Ubuntu 12.04 operating system patched with a real-time kernel v3.2.0-23 which converts Linux into a fully preemptible kernel. We compile the benchmarks using gcc v4.6.3 (with -O3 optimization level) and use pintools from the Pin kit v2.12-56759. The experimentation includes four platforms:

- An embedded target hosting a dual-Core Intel 1.66 GHz processors with 2 MB of cache, 2 GB of RAM, and digital IOs.
- An embedded target hosting a single-core VIA NAS7040 board with a C7-D 1.8 GHz processor and 128 KB of cache, 2 GB of RAM, and digital IOs .
- Two standard workstation hosting a quad-core i7-2600 3.4 GHz Intel processors with 8 MB of cache, and 16 GB of RAM.

We implement the function get_time() as an inlined assembly instruction that queries the processor cycles from the Intel processor's Time Stamp Counter (TSC). We inhibit task migration between cores and lock core speed's to operate at their maximum frequency to obtain accurate results from the TSC. The experiments run with a real-time scheduling policy and priority. Note that these modifications are for the purpose of obtaining accurate results for performance evaluation and are not required for the correct operation of DIME as Section 3.4 demonstrates.

To evaluate the performance of DIME, we use four pintools from the Pin 2.12 kit; dcache, inscount, regmix, and topopcode. dcache is a data-cache simulator that outputs the number of data-cache load hits and misses, and store hits and misses. We consider the instrumentation routine a light-weight one, since it just checks for the instruction type and accordingly inserts a call to one of the analysis routines. The tool contains seven analysis routines which are heavy-weight, since the routines contain nested function calls and may also contain a loop to retrieve data-cache information. At the end of the program execution, the tool writes the output to a file. inscount is a simple instruction counting tool that has a light-weight instrumentation routine and a light-weight analysis routine. The instrumentation routine only inserts a call to the analysis routine which increments an instruction counter. regmix is a register profiler that prints the used registers along with the number of read-accesses and write-accesses of each. regmix has a light-weight analysis routine that

only increments a counter, whereas, its instrumentation routine is heavy weight. The instrumentation routine extracts register information, at instrumentation time, through two calls to a function containing nested loops. The tool writes the output to a file at the end of the program's execution. topopcode is a profiler that prints the opcode of the executing instructions at runtime. The instrumentation routine is a heavy-weight one that calls two functions before analysis-routine insertion. Also, the analysis routine is heavy-weight since it is responsible for extracting information and printing the output at runtime. Originally, the instrumentation-routine granularity of all these tools is at the trace level, except for dcache which operates at the level of instructions. To implement a DIME version of dcache, we changed the granularity of dcache to operate on traces by looping over the instructions of the trace basic blocks. For the four tools, we used a budget B of 0.1 seconds per time period T of one second.

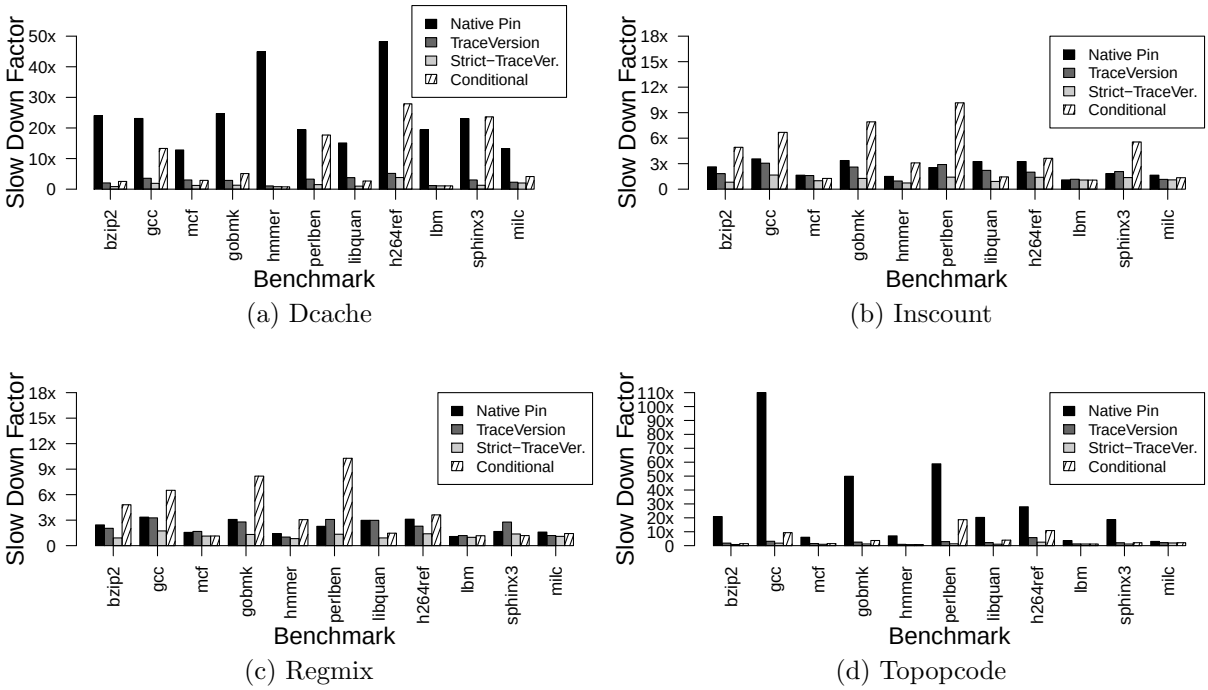
We empirically evaluate the performance of DIME using the following metrics:

- **Slow down factor of the dynamically instrumented program:** The slow down factor is the ratio of the execution time of the instrumented benchmark running on top of Pin to the execution time of the natively running benchmark. This metric highlights the overhead reduction of DIME compared to native Pin execution. It also compares the overhead of the three DIME implementations according to the nature of the instrumentation objective. Moreover, it guides the choice of which DIME implementation to use for a specific instrumentation objective.

- **Overshoots:** Recall that an overshoot will occur when instrumentation time exceeds the budget; i.e., $(B - t_{ins} < 0)$. The frequency of the overshoots as well as their severity measure how strictly each of the DIME implementations honors the budget. This metric varies according to the instrumentation objective and affects the overhead. It again helps in making an informed decision of the DIME implementation to use for a certain instrumentation objective.

3.3.2 Experimental Results

DIME, on average, outperforms native Pin in terms of overhead in the heavy-weight analysis-routine tools: dcache and topopcode. Figures 3.3a and 3.3d show the slow-down factors of Pin and DIME implementations with dcache and topopcode tools, respectively. The average slow down of native Pin with dcache is 24.3x, and with topopcode is 29.6x. *Trace Version* and *Strict Trace Version* achieve an overage slow down of 2.8x and 1.5x, respectively, with dcache. They also have an average slow down of 2.3x and 1.3x with



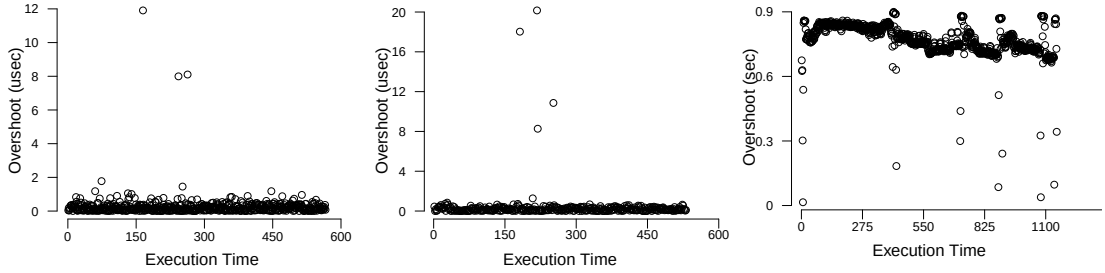
(a) Dcache (b) Inscount

(c) Regmix (d) Topopcode

Figure 3.3: Slow-down factors of native Pin and the three implementations of DIME.

topopcode, respectively. This reflects the higher budget checking overhead of *Trace Version* over *Strict Trace Version*. *Trace Conditional* has a slow down of 9.2x with dcache and 5x with topopcode. This is due to the frequent overshoots of *Trace Conditional* beyond the instrumentation budget.

Strict Trace Version has a low average slowdown of 1.1x for light-weight analysis-routine tools, while *Trace Conditional* is unsuitable for usage with such tools. With inscount, native Pin incurs a slow down of 2.3x, while *Trace Version* and *Strict Trace Version* maintain an average slow down of 1.9x and 1.1x, respectively, as shown in Figure 3.3b. Figure 3.3c presents the slow down with regmix, which are 2.2x, 2.2x, and 1.1x for native Pin, *Trace Version*, and *Strict Trace Version*, respectively. A light-weight analysis routine implies that the tool incurs minimal overhead with native Pin because analysis routines are the main source of overhead [69]. In such case, the overhead of DIME for checking budget and switching states is noticeable even if the instrumentation routine is heavy-weight (e.g. regmix). The reason is that DIME only bounds the execution time of the instrumentation code in the analysis routines. Also, frequent execution of a light-weight analysis routine, which might not consume the instrumentation budget, increases the budget checking over-



(a) *Trace Version* (b) *Strict Trace Version* (c) *Trace Conditional*

Figure 3.4: Overshoots of the three implementations of DIME with the mcf benchmark and dcache tool.

head. The results for *Trace Conditional*, in Figures 3.3b and 3.3c, reveal the performance degradation caused by the overhead of the overshoots in the light-weight analysis-routine tools. *Trace Conditional* has an average slow down of 4.2x with inscount and 3.8x with regmix. On average, the slow down of *Strict Trace Version* is lower than that of *Trace Version*, since *Strict Trace Version* has a lower budget-checking overhead (as discussed in Section 3.2.4). Additionally, both *Trace Version* and *Strict Trace Version* incur lower slow down in average compared to *Trace Conditional*.

Trace Version and *Strict Trace Version* have a very low overshoot value (order of microseconds) compared to *Trace Conditional*. Figure 3.4 shows the overshoot magnitude for the three implementations of DIME over the execution time of the mcf benchmark while instrumenting it using the DIME version of the dcache pintool. Although *Trace Conditional* has the lowest budget-checking overhead, its loose budget-respect results in high values of overshoots as shown in Figure 3.4c. Occurrence of high-valued overshoots depends on the structure of the program since *Trace Conditional* will not switch to *DBI-disabled* till the start of the next program trace. Figures 3.4a and 3.4b present the values of overshoots for *Trace Version* and *Strict Trace Version*, respectively. The values for the most frequent overshoots lie below 2 usec for both *Trace Version* and *Strict Trace Version*. This confirms that the two implementations strictly respect the budget since both switch to the *DBI-disabled* state once the budget is fully consumed.

Summary: DIME can achieve an average slow down as low as 1.25x using *Strict Trace Version* which always maintains a lower slow down compared to Pin. Both *Trace Version* and *Strict Trace Version* maintain low magnitude overshoots but with a higher budget checking overhead for *Trace Version*. *Trace Conditional* has a high average slow down

compared to the other implementations due to high magnitude overshoots. This makes *Trace Conditional* suitable for instrumentations with heavy-weight analysis routine that require achieving a high raw instrumentation coverage. Raw instrumentation coverage is the amount of information that DIME extracts as a ratio of the coverage of native Pin. Instrumentation coverage will be discussed in more details in Section 3.4 and Chapter 4. Both *Trace Version* and *Strict Trace Version* strictly respect the instrumentation budget. *Strict Trace Version* is well-suited for applications that require very low instrumentation overhead. *Trace Version* has a higher raw instrumentation coverage compared to *Strict Trace Version* but with a relatively higher overhead.

3.4 Case Studies

This section presents two case studies that demonstrate the applicability and scalability of DIME. The underlying idea relies on observations made in other work, that partial traces are useful in many applications as full traces contain many redundancies [78].

3.4.1 VLC Media Player

This case study demonstrates that instrumenting a soft real-time application such as a media player while playing a video requires a time-aware instrumentation approach. VLC is a free portable open-source media player developed by the VideoLan organization [13]. As mentioned in Section 1.4, VLC v2.0.5 has approximately 600 000 lines of code and uses libraries with more than three million lines of code. Our goal is extracting VLC's call context tree while VLC plays a high definition, 29.97 fps, 720x480, 1 Mbps bitrate video. The calling context helps in understanding programs, analyzing performance, and applying runtime optimizations [88]. We use the *DebugTrace* pintool, that is available as part of Pin's v2.12 kit, to extract VLC's call trace. We build a tool that extracts the call context trees from the call trace generated by native Pin and from the partial traces generated by DIME [88]. The time period T is set to one second throughout this case study. Table 3.2 shows the results of the case study.

Only DIME implementations permit extracting the call context tree while maintaining a continuous video playback. The video playback while instrumenting VLC using Pin and DIME were recorded and are available for viewing[1]. The original video has 599 blocks that

[1] "https://uwaterloo.ca/embedded-software-group/projects/time-aware-instrumentation"

	Trace Version	*Strict Trace Version*	*Trace Conditional*
Max budget without pauses	14%	38%	22%
Coverage in one run (nodes)	93.2%	92.8%	83.4%
(edges)	90.2%	90.9%	75.0%
Runs for 98% of CC tree	4	4	5
Runs for 99% of CC tree	5	5	6
Raw coverage at min. budget	23.6%	17.6%	62.8%

Table 3.2: Results for the VLC case study.

VLC decodes for viewing frames. VLC, using native Pin, decodes only 75 blocks which translates into an unwatchable video and errors messages for dropping frames.

All DIME implementations can recover 99% of the full call context tree with only few re-runs. *Trace Version* and *Strict Trace Version* extract 90% of the call context tree in one run. The first row in Table 3.2 shows the maximum instrumentation budget for each implementation that allows a continuous video playback without dropping frames. It also shows the coverage obtained by each implementation as a percentage of the nodes and edges of the full call context tree. Although *Trace Version* runs with the least budget, it obtains the highest coverage compared to the other implementations. Fully utilizing the budget, in this case, translates into more coverage of the call context tree. In general, DIME achieves a very good coverage while maintaining a continuous video playback.

When comparing the DIME implementations against each other with the minimal necessary budget, *Trace Conditional* extracts the most information. After getting the maximum instrumentation budget for each implementation that allows continuous video playback. The minimal necessary budget is the minimum of all these maxima. The last row in Table 3.2 shows the raw instrumentation coverage of each implementation at the minimal budget (14%). At the minimal budget, *Trace Conditional* extracts the most raw information, followed by *Trace Version*, then *Strict Trace Version*. This is because *Trace Conditional* loosely obeys the budget and *Strict Trace Version* has a delayed switch to the *DBI-enabled* state.

3.4.2 Laser Beam Stabilization

DIME is also useful for instrumenting control applications. The second case study is a laser beam stabilization (LBS) experiment developed by Quanser [10]. Laser beam stabilization is an important technology currently used in manufacturing equipment, surveillance, aircraft targeting, etc. The experiment consists of a stationary laser beam source pointing

	Average Displacement (mm)	Displacement Variance (mm)	Stability Budget (%)	Memory Pattern (%)
Original LBS	0.022	0.0002	N/A	N/A
Native Pin	2.988	0.487	N/A	100
Trace Version	0.509	0.881	0.3%	<1%
Strict Trace Version	0.588	0.531	0.6%	<1%
Trace Conditional	0.129	0.183	9.0%	78.1%

Table 3.3: Results for the LBS case study.

at a moving mirror. The reflected beam is detected by a high-resolution position sensing detector which measures the relative displacement of the beam from the nominal position. The mirror is free to oscillate along one axis. These oscillations power a motor driving an eccentric load. The turning of the motor plus an introduced disturbance voltage induce the undesired vibrations in the laser beam position. A feedback control system with a 1 KHz sampling rate stabilizes the laser beam position. Instrumenting such a time-sensitive application requires a time-aware instrumentation technique.

In this case study, we attempt to extract a memory access pattern from the LBS experiment. A memory access pattern contains the following for each accessed memory location: the effective address of the memory location, address of the instruction accessing memory, whether the memory access is a read or write, the data read/written, and the thread id. This information is useful in detecting memory-management problems and is used by numerous memory analysis tools such as QNX memory analyzer [9] and Valgrind [79]. We use the *DebugTrace* pintool, that is available as part of Pin's v2.12 kit, to extract the memory trace.

Only our DIME implementations can stabilize the laser beam while instrumenting. We repeated the LBS experiment 10 times for each instrumentation type. Each experiment runs for approximately 10 seconds. We collect the displacement (of the beam from the nominal position) data as measured by the position sensing detector. Native Pin fails to stabilize the laser beam which has a jittery response all over the detector. The instability of the native Pin version of the pintool is visible from the average displacement shown in Table 3.3 in the first column. The average displacement from the actual target is about 3mm, while the original unmodified software is about 0.02mm.

Trace Conditional is the only implementation of DIME that extracts most of the memory access pattern while allowing stabilization of the laser beam. Table 3.3 shows the maximum budget for each implementation of DIME that allowed stabilization of the laser

beam. *Trace Conditional* extracts information with the highest budget compared to the other implementations while maintaining the lowest displacement. This demonstrates that for this type of application, the overhead of budget checking causes more performance degradation compared to the overhead of overshoots. *Trace Conditional* achieves the best coverage for the memory access pattern while the other implementations are unsuited for instrumenting the LBS experiment. We think that the reason is the LBS program structure which has a very low number of branches and has successive bursts of memory accesses. This creates long instruction traces in Pin. This allows *Trace Conditional* to instrument all memory access instructions in a long trace in one call of the instrumentation routine (if enough budget is available) without checking for budget between memory accesses. The analysis routine calls for budget checking and version switching between memory accesses in *Trace Version* and *Strict Trace Version* causes a degradation in the response time of the controller. This shows that program structure can be a factor in choosing a suitable implementation of DIME for instrumentation.

3.5 Summary

Most of the existing instrumentation tools do not consider timing properties of applications. Current time-aware instrumentation tools are both static and source-code tools that require WCET analysis before and after instrumentation. This makes them impractical for instrumenting library dependencies and makes them more suited for hard real-time applications where WCET analysis is commonly employed. We propose DIME; a time-aware dynamic binary instrumentation tool. DIME has three implementations as extensions to Pin. These implementations differ in their budget checking overhead, strictness of respecting budget, overshoots beyond the budget, and instrumentation coverage. The performance evaluation of DIME shows an average reduction in overhead by 12, 7, and 3 folds compared to native Pin. Two case studies demonstrate the applicability and scalability of DIME to media-playing software and control applications. They also show that the coverage obtained by the different implementations vary according to the instrumentation objective and program structure.

Chapter 4

Redundancy Suppression in DIME

DIME extracts partial tracing information since DIME disables instrumentation when the instrumentation budget is consumed. This chapter discusses tracing-redundancy suppression to increase the instrumentation coverage of DIME through multiple runs.

4.1 Overview

Instrumentation frameworks, including native Pin [69] and DIME [17], may generate an amount of runtime information that contains many redundant entries. The extracted information can form several gigabytes of data. Many researchers have reported the complexity of understanding software systems using the collected runtime information due to its sheer size and complexity [43, 44]. One reason is the redundancy of extracted traces which occurs due to, for example, the repetition of sequence of events [30]. The high redundancy consumes memory and disk space. It also provides a minor contribution to the program understanding process [30]. In [43], removing contiguous repetitions in call traces dropped the size of the extracted information to between 5% and 46% of the original size. For many analysis tools, instrumenting each instruction once is sufficient since the collected information is the same regardless of the number of times the instruction is executed or instrumented. Examples of these tools include branch profilers used for extracting code coverage and memory profiling tools used for building memory access patterns. Cornelissen and Moonen [31] analyzed the call traces of six different systems. The number of unique calls reported in this work ranges from 0.01% to 6.1% of the total number of extracted calls (geometric mean = 0.3%). The authors measured the repetitiveness which corresponds to

the degree of information redundancy. The level of repetitiveness ranges from 93.8% to 99.9% of the call traces.

Since DIME extracts partial tracing information, it can obtain higher instrumentation coverage through the avoidance of collecting redundant information. To respect the timing properties of a program, DIME disables the instrumentation when the instrumentation budget is consumed. Accordingly, DIME generates partial tracing information compared to native Pin. In other words, there exists a trade-off between the instrumentation budget and the instrumentation coverage. Hence, multiple runs of DIME are required to increase the instrumentation coverage and optimally achieve full coverage. From a performance point of view, it is preferable to minimize the number of required runs. For DIME, this implies obtaining the maximum possible coverage from each single run without violating the timing constraints. Accordingly, DIME should avoid tracing redundant instrumentation. We specifically focus on the type of analysis tools that do not require tracing redundant information. DIME should utilize the available instrumentation budget for extracting unique (non-redundant) information.

To prohibit redundant instrumentation, DIME should be able to identify the instrumented code regions. In general, the minimum piece of information needed to identify a code region is the starting address. Thus, the basic idea is to enable DIME to save the starting addresses of instrumented code regions in a log, and DIME should then check the log before instrumenting a new code region. For the approach to be efficient, DIME should:

- Prevent re-instrumentation of a code region in the current run and all subsequent runs of the program under analysis.
- Avoid increasing runtime overhead.
- Avoid creating large-sized logs which increase DIME's memory consumption. Searching a large log may also result in increased runtime overhead.

Both steps, saving to the log and searching it, take place in the instrumentation routine, so its overhead is expected to be negligible. We avoided adding these steps to the analysis routine which is the main source of overhead in Pin [69].

In what follows, we discuss our approach for suppressing redundancies in DIME.

4.2 Granularity of Logged Code Regions

The first design aspect that we discuss is the granularity of code regions to be recorded in the log. We can log addresses of code regions either at the instruction or the trace level. As mentioned in Section 1.5, a trace is a straight-line code sequence that ends in an unconditional control transfer, a predefined number of conditional control transfers, or a predefined number of instructions. It is inefficient to log the address of each instrumented instruction, since

1. This requires frequent access to the log which adds to the runtime overhead.
2. This results in a large log size which consumes memory.
3. This leads to searching a large-sized log which can delay program execution and add to the runtime overhead.

An alternative to logging instruction address is to log addresses at a coarser granularity, the trace level. The instrumentation routine analyzes traces to insert analysis-routine calls. If Pin detects a jump to an instruction in the middle of a trace, Pin will create a new trace beginning at the target instruction. So, the instructions inside a trace are always in series i.e., uninterrupted by instructions from another trace. Thus, DIME will save the trace starting address in addition to the length of the instrumented portion in the trace ($\langle$Trace Address, Trace Length$\rangle$). Specifically, DIME will save the relative starting address of the trace with respect to the trace's image. This guarantees that saved addresses are deterministic between successive runs (especially for the traces of shared libraries). On the other side, as mentioned earlier in Section 3.2, trace version switching can cause Pin to create a new trace. Thus, some trace addresses might only exist in a subset of the runs.

4.3 Efficient Log Search

The second design aspect is saving the trace addresses and the trace lengths in a manner that allows for efficient searching of the log. In this section, we propose three approaches for saving trace addresses and length. We compare among them qualitatively and quantitatively deriving unexpected results. We choose one of these approaches to provide DIME with the capability of suppressing redundant instrumentations.

4.3.1 Hash-Table Log

The first approach uses a hash-table as the log for saving instrumented traces. In this approach, DIME saves the trace address to identify instrumented traces. Whenever DIME instruments a trace, it adds the trace's address to the hash-table. Also, before instrumenting any trace, DIME searches for the trace address in the hash-table. Let A be the current trace and B be a trace in the log L; Then, DIME will only instrument A, *iff* $(address(A) \neq address(B)) \forall B \in L$. The advantages of using a hash-table for logging traces are:

- Fast logging (average case: constant; worst case: linear in the hash-table size).
- Fast searching (average case: constant; worst case: linear in the hash-table size).
- Low number of false negatives (as will be discussed in Section 4.4). False negatives will occur if DIME prohibits instrumentation of an uninstrumented trace. For instance, let A be the current trace, where $A = \langle 100, 80 \rangle$ (i.e., $address(A) = 100$ and $length(A)$=80). If the log contains trace B, where $B = \langle 100, 20 \rangle$, DIME will prohibit the instrumentation of A. Thus, instructions in address range 120 to 180 will not be instrumented in any run.

The disadvantage of this approach is that:

- Using a hash-table enables DIME to only compare trace addresses while ignoring the trace length. This results in false positives. A false positive will occur if DIME allows instrumentation of a previously instrumented trace. For example, let A be the current trace, where $A = \langle 150, 20 \rangle$. DIME may fail to find A in the log, although the log contains trace $B = \langle 100, 80 \rangle$. This means that trace A is previously instrumented as a part of trace B. Note that one trace being part of another happens due to the creation of new traces through version switching as explained earlier.

Section 4.4 presents experimental results that support the listed advantages and disadvantages for the three approaches.

4.3.2 BST Log

The second approach is using a binary search tree (BST) to log the addresses of the instrumented traces along with their lengths. Being sorted, the BST facilitates jumping

to a specific range of addresses. When DIME instruments a trace, it adds the trace to the log such that the trace address is the key and the trace length is the value. Before instrumenting a trace, DIME searches the BST using the trace address. If not found, DIME will jump to the log-entry that has the first smaller trace address compared to the current trace address. DIME will then decide if the current address lies within the trace of the discovered log-entry. Let A be the current trace and B be a trace in the log L. DIME will not instrument A, if $\exists B \in L$ s.t. $(address(B) \leq address(A) < address(B) + length(B))$. The advantage of using a BST for logging traces is:

- Less false positives compared to the hash-table approach due to considering the lengths of the logged traces.

The disadvantages of this approach, on the other hand, are:

- Slower than the hash-table approach in the average case; the complexity of both saving and searching is $log(N)$.
- Relatively high false negatives. Consider the following example. Assume the current trace is $A = \langle 100, 200 \rangle$, and the log entry $B = \langle 50, 80 \rangle$ in the log L. This approach will prevent instrumenting trace A since its starting address lies within the log entry B. This, however, will consequently prevent DIME from instrumenting the uninstrumented portion of trace A i.e., from address 130 to address 300.

Additionally, after the program execution and before saving the log to a file for use in subsequent runs, DIME merges directly consecutive traces leading to a smaller log size. For example, if the log contains two log entries $\langle 100, 50 \rangle$ and $\langle 150, 50 \rangle$, DIME will merge them into one log entry $\langle 100, 200 \rangle$. Merging log entries decreases the log size and, therefore, reduces the search time leading to less runtime overhead in subsequent runs of DIME.

4.3.3 Merger-BST Log

The third approach utilizes a BST as well, but it addresses the second disadvantage of the previous approach. Using this approach, DIME will prohibit instrumentation, only if the *whole* current trace is part of a log entry. Otherwise, DIME allows instrumentation and merges the current trace with the log entry if needed. Let A be the current trace and B be a trace in the log L. DIME will not instrument A, if $\exists B \in L$ s.t. $(address(B) \leq address(A) < address(B) + length(B) \wedge address(A) + length(A) <$

$address(B) + length(B)$). For example, let the current trace be $A = \langle 100, 200 \rangle$, and the log entry contains $B = \langle 50, 80 \rangle$. In this approach, DIME instruments trace A. Afterwards, DIME merges trace A and B into one log entry $\langle 50, 250 \rangle$ to avoid redundancies in the log. The advantages of this approach are:

- Less false negatives, compared to the BST approach.

The disadvantages are:

- Slower than the hash-table approach in the average case; the complexity of both saving and searching is $log(N)$.
- Higher false positives than the BST approach since it allows re-instrumentation of some portions of a trace.

Note that a trace address and length will be saved in the log, only if the trace is actually instrumented i.e., the trace's version is V_INSTRUMENT.

4.4 Evaluation of the Log Search Approaches

In this section, we describe our experiments to evaluate the three log-search approaches. For brevity and conciseness, we evaluate the approaches using only the *Trace Version* implementation of DIME (Section 3.2.1). According to the results, we choose which approach extends DIME with the feature of suppressing redundant instrumentation output.

We experiment with two SPEC2006 C benchmark [46] programs (lbm and mcf) for three runs. We later, in Section 4.5, use more SPEC benchmarks to evaluate the performance of DIME over up to eight runs. The experiments run on top of a workstation hosting a quad-core i7 3.4 GHz Intel processors with 8 MB of cache, and 16 GB of RAM. The operating system is an Ubuntu 12.04 patched with a real-time kernel v3.2.0-23 to convert Linux into a fully preemptible kernel. We use the same implementation of the get_time() function as in Section 3.3.1. To obtain accurate results, we inhibit task migration between cores and lock core speed to the maximum frequency. The experimentation environment also maintains a real-time scheduling policy and priority. These modifications guarantee accurate results for performance evaluation and are not mandatory for DIME correctness. The experiment uses a branch-profiling analysis tool which has a heavy-weight analysis routine. It prints out the jump, call, and return instructions in addition to the source address and the destination

address. The tool is based on the *branch_target_addr* pintool that is available as a part of the Pin's kit v2.12-56759. The pintool is modified to extract the branch profile of the whole program instead of only a part of it. A branch profiler is useful for investigating the code coverage of a program. The time-period parameter in DIME is set to one second and the instrumentation budget is set to 0.1 seconds. Each experiment, in this section and Section 4.5, is conducted once due to the very long execution time when instrumenting the benchmarks on top of native Pin. For example, povray benchmark originally executes in 2.5 minutes, but on top of native Pin, it consumes four days of CPU time. The execution time of the other benchmark programs, on top of Native Pin and DIME, will be discussed in Section 4.5. However, the runs of each DIME experiment can be considered as repetitions since all the runs operate identically. There exists one minor difference between the first run and the following ones. The first run starts with an empty log while the following runs read the log from a file before launching and instrumenting the program.

The evaluation of the proposed approaches is based on the following metrics:

1. **Instrumentation Coverage**: The ratio of the instrumentation output of DIME to that of native Pin. Note that we consider only unique (non-redundant) traces. Increasing the instrumentation coverage is the main objective of the proposed approaches.

2. **False positives:** A false positive will occur if DIME permits the instrumentation of a previously instrumented trace or trace-portion. This metric measures the ratio of false positives to the total number of instrumented traces in the current run. The ratio of false positives indicates the efficiency of the log searching approach in identifying previously instrumented traces. As the ratio of false positives decreases, the budget utilization increases and the number of required runs to maintain high coverage decreases.

3. **False negatives:** A false negative will take place when DIME refuses to instrument a trace which was not instrumented before. The metric measures the ratio of false negatives to the total number of traces that got rejected by DIME in the current run. This value includes the trace portions as well i.e., a part of the trace is instrumented but the other part is not. As the ratio of false negatives increases, the ability of the approach to maintain high coverage decreases.

4. **Slow-down factor of the instrumented program**: The ratio of the execution time of the dynamically instrumented benchmark to the execution time of the natively running benchmark. This metric examines the ability of DIME to reduce

runtime overhead, compared to Pin, while saving to and searching the log to suppress instrumentation redundancy. It also checks if the three approaches introduce different runtime overhead. Low runtime overhead is essential for the instrumentation of time-sensitive systems as discussed before.

5. **Overshoots:** An overshoot will occur when actual instrumentation time exceeds the budget. The magnitude of the overshoots shows how strictly DIME respects the instrumentation budget. This metric also checks the effect of the different approaches on the magnitude of the overshoots.

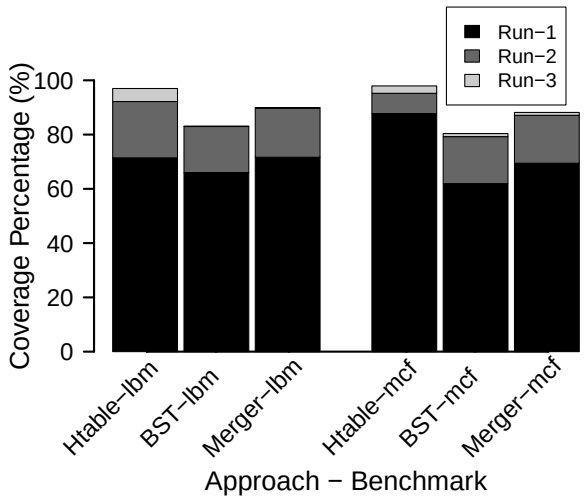


Figure 4.1: Instrumentation coverage of the redundancy suppression approaches

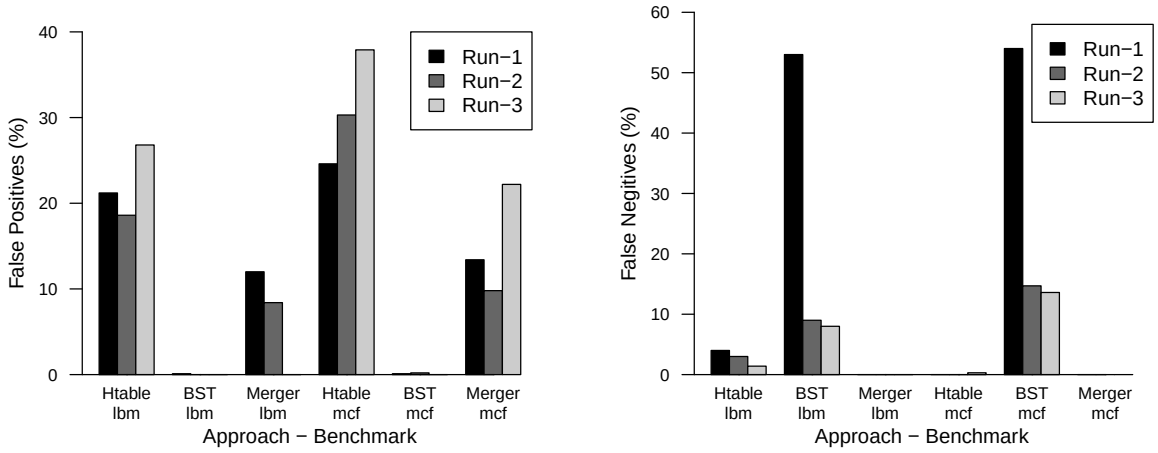
```
test r12, r12
setnz byte ptr [rsp+0x3b]
jnz 0x7ffff7de26a8
mov eax, dword ptr [rsp+0x30]
and eax, 0x10000000
...
call 0x7ffff7df2850
```

Listing 4.1: Example (2) of a trace.

Figure 4.1 shows the instrumentation coverage of the three approaches with lbm and mcf. The hash-table approach guarantees the highest instrumentation coverage. After three runs, it achieves 97% of the instrumentation coverage of native Pin for lbm benchmark,

and 98% for mcf. The coverage of BST is 83% and 80% for lbm and mcf, respectively. Finally, BST-Merger generates 90% and 88% of the instrumentation output for lbm and mcf, respectively. The low ratio of false negatives of the hash-table approach is one reason for achieving the highest coverage. The hash-table approach is a conservative one which favors re-instrumenting some trace portions over uninstrumenting them. Also, some scenarios lead to a decreased instrumentation coverage for the BST and Merger BST approaches compared to the hash-table approach. As mentioned previously, a trace can have multiple exits, e.g., can include multiple jump instructions. Listing 4.1 is an example of a trace with multiple exits (contains jnz and call instructions). Assume the starting address of the trace is 34192 and the trace length is 62. Assume DIME encounters this trace for the first time, and cannot find the address 34192 in the log as a key or as a part of another log-entry. Hence, DIME allows instrumentation of this trace. Assume that enough instrumentation budget is available to instrument all the instructions in the trace. Thus, the trace address along with its length are saved in the log as $\langle 34192, 62 \rangle$. The instrumentation-routine inserts analysis-routine calls for all the instructions in the trace. Assume that in the first run of DIME, the first three instructions only execute and a jump (through jnz) occurs. In the second run, DIME (BST and BST-Merger) prohibits instrumentation for the trace 34206 (starting from the *mov* instruction) since it lies inside the logged trace $\langle 34192, 62 \rangle$. Accordingly, no information is extracted starting from the address 34206 since these instructions do not execute in the first run and DIME prevents their instrumentation in the following runs. Although, the program runs with the same inputs, this can occur due to non-deterministic execution of some shared libraries such as libc and the Linux loader. For such shared libraries, execution can slightly change according to the processor state. In such cases, the BST and the Merger BST approaches fail to extract some information, thus decreasing their instrumentation coverage.

The ratio of false positives is shown in Figure 4.2a. The hash-table approach has the highest ratio with both lbm and mcf benchmarks. The BST-Merger approach has moderate values of false positives, whereas BST has approximately zero false positives. This means that BST accurately identifies the previously instrumented traces and efficiently utilizes the budget to instrument other traces. On the other hand, BST has a high ratio of false negatives, as shown in Figure 4.2b, which is an undesirable feature. BST-Merger sustains approximately zero false negatives, and hash-table has negligible ratios of false negatives. The scenarios discussed in Section 4.3 explain the values in Figures 4.2a and 4.2b. Note that the false-negatives ratio is more critical than false positives. Although false positives cause instrumentation redundancies, it is safer. False negatives prevent code portions from being instrumented in any run which can dramatically decrease the instrumentation coverage. The results show that there is a trade-off between false positives and false negatives. In



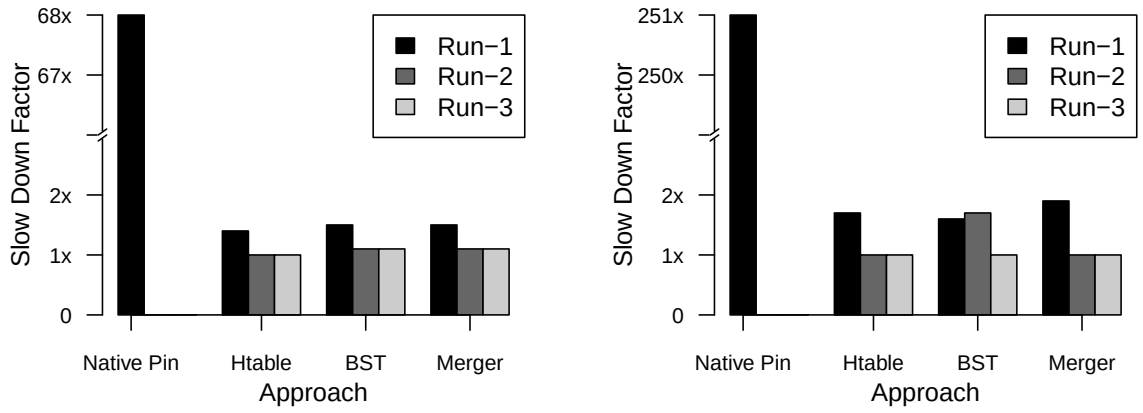
(a) False Positives

(b) False Negatives

Figure 4.2: False-positives ratio and false-negatives ratio of the redundancy suppression approaches

such case, we prefer the approach that maintains low ratio of false negatives even if it has high ratio of false positives. Thus, the hash-table and the BST-Merger approaches outperform the BST one in this regard.

Figure 4.3a presents the slow-down factors of native Pin and the proposed log-keeping approaches of DIME with the lbm benchmark. On top of native Pin, lbm runs 68x slower than the native execution. On the other hand, the hash-table approach of Pin achieves a slow-down of 1.4x, 1x, and 1x for three consecutive runs. The overhead of the BST approach is 1.5x, 1x, and 1x, while that of BST-Merger is 1.5x, 1x, 1x for three runs. In Figure 4.3b, native Pin slows down the execution 251x with the mcf benchmark. Whereas, the slow-down factors of the hash-table approach for the three runs are 1.7x, 1x, and 1x. These of BST are 1.6x, 1.7x, and 1x, and BST-Merger shows a slow-down of 1.9x, 1x, and 1x. To sum up, DIME reduces the runtime overhead by at least 45 folds for lbm and 132 folds for mcf. These numbers reveal that the three modifications of DIME are able to dramatically reduce the runtime overhead of native Pin. Thus, all of the three DIME modifications are suitable for dynamically instrumenting time-sensitive systems. Comparing the three approaches to each other, none of them shows a significant overhead-decrease over the others. Consequently, runtime overhead is not a factor that differentiates among the three approaches.



(a) lbm benchmark

(b) mcf benchmark

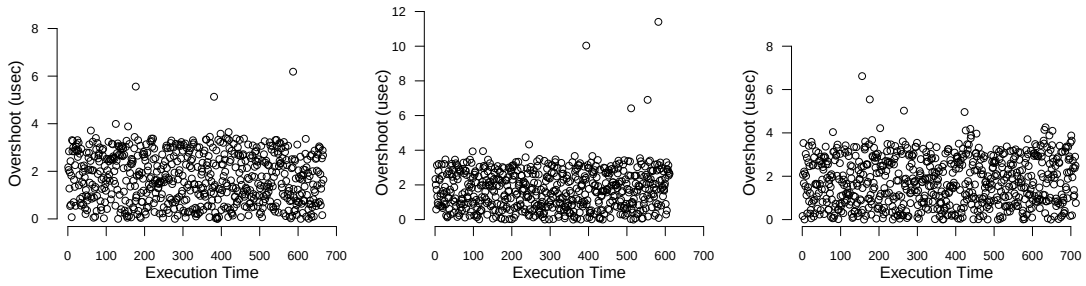
Figure 4.3: Slow-down factors of native Pin and the redundancy suppression approaches of DIME

Figure 4.4 shows the overshoots' magnitude for the three proposed approaches of DIME over the execution time of the mcf benchmark while instrumenting it using the DIME version of the branch-profiling pintool. The three approaches respect the instrumentation budget; the values for the most frequent overshoots lie below 4 microseconds. The differences in the overshoots' magnitudes among the three approaches of DIME are insignificant. Thus, this metric is also not a factor to favor one approach over the others.

Non-intuitively, the evaluation metrics reveal that the simplest approach, which is the hash-table one, results in the best instrumentation coverage results. Moreover, the hash-table approach provides low values of false negatives, maintains low runtime overhead, and respects the instrumentation budget. Accordingly, we choose the hash-table approach to support instrumentation-redundancy suppression in DIME.

4.5 Performance Evaluation

This section presents the experimentation of redundancy suppression in DIME and discusses its performance. In these experiments, DIME uses the *Trace Version* implementation along with the hash-table trace logging.



(a) Hash-table (b) BST (c) Merger BST

Figure 4.4: Overshoots of the redundancy suppression approaches in the first run with the mcf benchmark.

4.5.1 Experimental Setup

The experimental setup is similar to the one described in Section 4.4. We experiment with the SPEC2006 benchmark suite [46] including C and C++ integer and floating point programs. The instrumentation objective is extracting the branch-profile of the programs. We use the Pin kit v2.12-56759 and gcc v4.6.3. The time-period parameter is also set to one second and the instrumentation budget is set to 0.1 seconds for all the benchmark programs. Note that the experimentation included more benchmarks, however, these extra benchmarks are not reported since the execution time on top of native Pin exceeded twenty days.

We evaluate the performance of DIME using the metrics: (1) instrumentation coverage and (2) slow-down factor of the instrumented program. These metrics have already been defined in Section 4.4.

4.5.2 Experimental Results

DIME is capable of maintaining high instrumentation coverage. Figure 4.5 shows the ratio of the amount of the extracted instrumentation output through multiple runs of DIME with respect to that extracted by native Pin. DIME is capable of extracting 97% and 99% of the instrumentation output in four runs for dealII and mcf benchmarks consequently. In five runs, DIME generates a coverage of 99% for both namd and lbm. It extracts 92% and 97% in the sixth run for milc and povray consequently. DIME also extracts 98% when instrumenting xalancbmk for seven runs, and 91% of the instrumentation output of sphinx3 after eight runs.

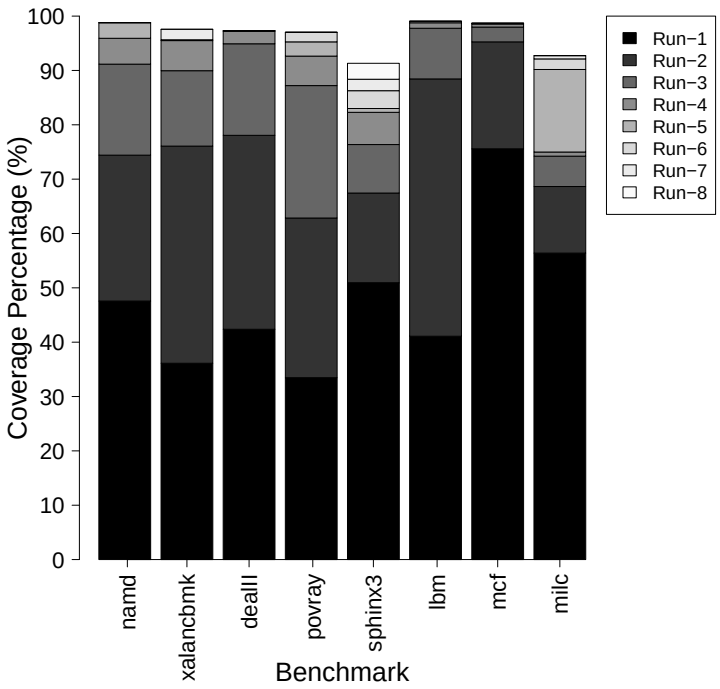


Figure 4.5: Instrumentation coverage of DIME with redundancy suppression

DIME outperforms native Pin in terms of runtime overhead. It reduces the runtime overhead of dynamic instrumentation by one to three orders of magnitude compared to native Pin. Figure 4.6 shows the slow-down factor of native Pin, and the average slow-down factor of the eight runs of DIME for each benchmark program. Note that the average slow-down factor is the geometric mean. Native Pin dramatically slows down the program execution. The average slow down of native Pin is 706x the original benchmark execution, with maximum value of 2971x and minimum value of 67x. The benchmarks' continuous execution time on top of native Pin ranges from four hours to nine days with an average of four days. On the other hand, the benchmarks on top of DIME take from three to 42 minutes with an average of nine minutes. The average slow-down of DIME is 1.5x, with a maximum value of 8x and a minimum value of 1x.

Running a program multiple times on top of DIME is less time-consuming than native Pin. More importantly, DIME allows the program to maintain its timing properties. Respecting such properties is essential for time-sensitive systems. Additionally, multiple runs of DIME provide very high instrumentation coverage compared to native Pin.

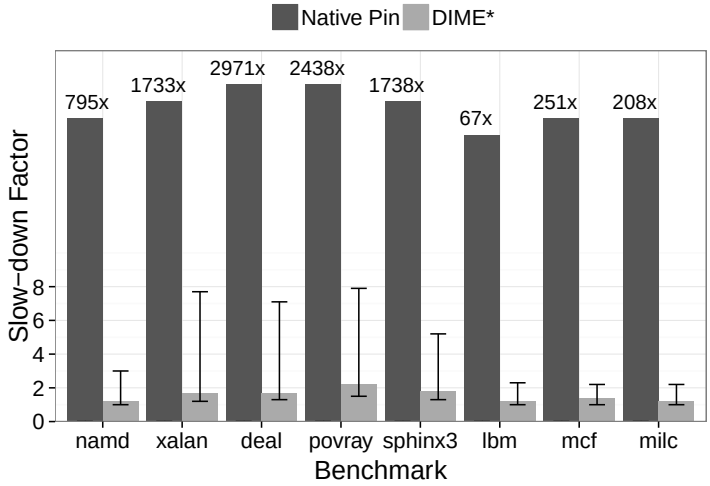


Figure 4.6: Slow-down factors of native Pin, and DIME with redundancy suppression

4.6 Case Studies

This section demonstrates the scalability and the practicality of DIME through two case studies.

4.6.1 VLC Media Player

This case study reveals the usability of DIME for instrumenting multi-threaded time-sensitive software. VLC [13] is a multi-platform media player that consists of approximately 600 000 lines of code and uses dependencies of approximately three million lines of code. Similar to the case study in 3.4.1, we aim to extract the call context tree of VLC v2.0.8 while playing a high definition, 29.97 fps, 720x480, 1 Mbps bitrate video. The tool, used in this case study, is based on the DebugTrace pintool which is available in Pin's v2.12 kit. We use the tool to extract the call traces, then we use the call traces to build a call context tree. The platform is an Ubuntu 12.04 machine hosting a quad-core i7 2.6 GHz Intel processors with 8 GB of RAM. For DIME, we set the time period to one second and the budget to 0.1 second which enables VLC to run the video smoothly.

Table 4.1 shows the number of video blocks that VLC decodes for viewing frames. The number of decoded blocks are shown for VLC without instrumentation, with native Pin,

	Decoded Blocks	Nodes(%)	Edges(%)
Original	594	N/A	N/A
Native Pin	37	100%	100%
DIME (run 1)	574	95.7%	93.6%
(run 2)	594	98%	96.2%
(run 3)	594	98.2%	96.5%

Table 4.1: Results for the VLC case study (redundancy suppression)

and with DIME. The table also lists the extracted percentage of the nodes and the edges of the call-context tree (i.e., the cumulative coverage of DIME compared to Pin).

Our conclusion is that DIME dramatically decreases the run-time overhead of dynamic instrumentation, respects the timing properties, and maintains almost the same instrumentation coverage. On top of native Pin, VLC fails to maintain a continuous video playback. The video is unwatchable; only 37 video blocks (out of 594) are decoded. VLC generates multiple errors of dropping video frames due to very slow processing. On the other hand, DIME enables VLC to play the video continuously. VLC decodes 574, 594, and 594 video blocks (out of 594) for the three runs consecutively. DIME extracts 98% of the call context tree nodes and 96% of the edges.

4.6.2 PostgreSQL

DIME is useful for extracting sufficient analysis information while maintaining high quality of service (QoS). PostgreSQL [8] is a powerful object-relational database management system that supports SQL standards. PostgreSQL 9.1 consists of approximately one million lines of code. When an application sends a database request to PostgreSQL, a connection will be established, and a parser will check the query syntax and create a query tree. Then, an optimizer generates a query plan from the tree and sends the plan to the executor. Finally, the executor steps through the commands in the query plan to retrieve the required information or to make the required updates to the database. Pgbench [7] is a benchmarking tool for testing the performance of PostgreSQL. It runs a sequence of transactions multiple times in multiple database sessions. The objective of this case study is the extraction of the branch-profile of PostgreSQL while processing a total of 800 000 transactions. PostgreSQL runs 16 database sessions where each session consists of 50 000 transactions. The case study runs on an Ubuntu 12.04 platform hosting a quad-core i7 2.6 GHz Intel processors with 8 GB of RAM. We set the budget of DIME to 7% and the time period to one second. These values enable DIME to extract sufficient information while keeping high performance of PostgreSQL.

Table 4.2 shows the performance of PostgreSQL while (1) running natively, (2) running on top of native Pin, and (3) running on top of DIME. The total time consumed to process the 800 000 transactions is an indication of the performance. As the processing time increases, the degradation in the quality of service (QoS) increases. The processing time reported excludes connection-establishing time. The table also shows the coverage of DIME with respect to that of Pin.

In this case study, DIME extracts 97% of the analysis data while reducing the runtime overhead by 68 folds compared to native Pin. Originally, PostgreSQL processes all the transactions in 46 seconds. Native Pin causes a slowdown of 96x; PostgreSQL executes the same number of transactions in 1.2 hours. On the other hand, DIME maintains slowdown of only 1.4x, 1.4x, and 1.3x in three runs, consecutively.

	Total Time (sec)	**Coverage(%)**
Original	46	N/A
Native Pin	4419	100%
DIME (run 1)	65	42%
(run 2)	65	75%
(run 3)	61	97%

Table 4.2: Results for the PostgreSQL case study (redundancy suppression)

4.7 Summary

DIME provides high instrumentation coverage, respects the timing properties, and dramatically reduces the run-time overhead of dynamic binary instrumentation. DIME suppresses logging of redundant tracing information to increase instrumentation coverage. The results show a reduction in the overhead of the instrumentation process by one to three orders of magnitude compared to native Pin while achieving up to 99% of the instrumentation coverage. DIME was able to extract 97% of the call context tree of the VLC video player while playing a high definition video. VLC fails to provide a watchable video while being instrumented using native Pin. DIME was also used for the branch profiling of the PostgreSQL database management system and was able to extract 97% of the instrumentation information in three runs. DIME extracts this information in less than two minutes per run while native Pin takes 1.2 hours to extract the information. These case studies show the scalability of DIME and its ability to limit the instrumentation overhead while achieving a high instrumentation coverage.

Chapter 5

Parameter Tuning of DIME

5.1 Overview

This chapter discusses the idea of tuning the parameters of DIME to achieve higher performance, i.e., lower runtime overhead and higher instrumentation coverage. The operation of rate-based dynamic binary instrumentation depends on two main factors: the time period T and the instrumentation budget B. DIME allows instrumentation to run for a maximum of B time units in every time period T (as illustrated in Chapter 3). The instrumentation budget B is specified during the system design phase. The experiments conducted in Chapters 3 and 4 demonstrate the applicability of DIME for different programs at the default time period of one second. However, it is unknown how the value of the time period affects the performance of DIME. For instance, does changing the value of the time period T increase or decrease the slow-down factor of the instrumented program? Does the value of the time period T influence the coverage extracted by DIME? Is there an interaction between the value of the instrumentation budget B and that of the time period T? Defining the relation between T and DIME's performance provides a deeper understanding of the internals of DIME and, accordingly, offers practical guidance for DIME parameter tuning.

5.2 Methodology and Experimental Design

This section describes the setup of the experiments which aim to understand the effect of DIME's parameters on the runtime overhead and the coverage of the instrumented program. The main factors of interest are the time period T and its interaction with

the instrumentation budget B. There are other factors that can contribute to DIME's performance such as the experimentation platform, the analysis tool, and the program to be instrumented. We conduct the experiments on three different program categories and vary the values of the time period T and the budget B. The experiments report the slow-down factors of the instrumented programs along with the unique coverage and the raw coverage. Finally, the analysis-of-variance (ANOVA) statistical test is used to judge the experiment hypotheses.

The budget value B, in this chapter, denotes the percentage of the time period specified for instrumentation. Assume that the execution time of an instrumented program P equals to 10 seconds. A budget of $B = 10\%$, for example, means that the maximum total instrumentation time is one second regardless of the time period T. The time-period value T determines how the budget usage is distributed over the execution of the instrumented program. A high time-period value forces the budget to be available in larger chunks but less frequency than a low time period.

5.2.1 Hypotheses

The following hypotheses formulate the research questions investigated through the experiments.

Hypothesis 1. *There exists a negative correlation between the value of the time period T and the slow-down factor of the instrumented program atop of DIME.*
As discussed in Chapter 3, when DIME fully consumes the budget B, it turns off instrumentation by switching the trace version from V_INSTRUMENT to V_BASE. As the time-period value decreases, the frequency of version switching in DIME increases. For example, with a time-period value of 1 sec, DIME will switch versions twice per second in the worst case. The first version switch occurs when the budget is fully consumed, and the second switch is due to budget replenishment at the end of the period. Whereas, for a 0.25 second time period, there exist eight version switches per second in the worst case leading to a higher runtime overhead. This hypothesis evaluates the efficiency of DIME with respect to runtime overhead.
We define the null hypothesis as $H_0^s : \mu_{low}^s = \mu_{high}^s$ such that μ_{low}^s and μ_{high}^s are the means of the slow-down factor at the low level and the high level of T respectively. In other words, the null hypothesis H_0^s states that the means of the slow-down factors are equal at the low and the high levels of T. We conduct an ANOVA test to judge the null hypothesis. A rejected null hypothesis proves the presence of a correlation between the time period T and

the slow-down factor of the instrumented program. On the other hand, if the ANOVA test failed to reject the null hypothesis, it suggests the absence of the mentioned correlation. The following sections include more details on the ANOVA tests and results.

Hypothesis 2. *There exists a negative correlation between the value of the time period T and DIME's unique instrumentation coverage.*
The unique instrumentation coverage represents the amount of extracted non-redundant information. As the time period increases, DIME is expected to collect more redundant traces according to the instructions' locality. The principle of temporal locality states that recently executed instructions are likely to be re-executed in the near future [83]. Also, a program can spend about 90% of its execution time running only 10% of its instructions [83]. For example, nested loops have a relatively high number of branch instructions of which many are non-unique. With a high time-period value, DIME may waste the budget extracting all the branch instructions in the nested loops. This results in a higher total number of extracted traces but may lead to a lower number of unique traces. Whereas, a low time-period value forces DIME to split the budget covering more code regions and increasing the probability of collecting unique traces. This hypothesis studies the impact of DIME parameters on collecting quality runtime information.
Given that μ^u_{low} is the mean unique coverage at the low level of T and μ^u_{high} is that at the high level of T, the null hypothesis is defined as $H^u_0 : \mu^u_{low} = \mu^u_{high}$. Using ANOVA, we test the null hypothesis H^u_0. The rejection of the null hypothesis can prove the correlation between the time period T and the unique instrumentation coverage. Contrarily, failing to reject the null hypothesis suggests that the means of the unique coverage at the high and the low levels of T are equal, i.e., no correlation exists.

Hypothesis 3. *The value of the time period T is positively correlated with DIME's raw instrumentation coverage.*
The raw instrumentation coverage indicates the total amount of extracted runtime information. As the time period increases, DIME may collect more redundant traces resulting in a higher number of traces in total (as explained previously). This hypothesis evaluates the impact of the time-period T on DIME's ability to extract runtime information. This is useful when, regardless of the uniqueness, the full output of the analysis tool is of interest. An example of such an analysis tool is the data-cache simulator which extracts cache hits and misses during program execution.
The null hypothesis, in this case, is $H^r_0 : \mu^r_{low} = \mu^r_{high}$ where μ^r_{low} and μ^r_{high} denote the mean raw coverage at the low and the high level of T respectively. We can confirm the existence of a correlation between the time period T and the raw instrumentation coverage if the

ANOVA test rejects the null hypothesis H_0^r. Otherwise, failing to reject the null hypothesis implies the absence of the mentioned correlation due to the equality of the means of the raw-coverage values at the high and the low levels of T.

5.2.2 Experimental Factors

This section describes a number of factors that contribute to the performance of DIME and explains how the experimental setup handles these factors.

Input Factors and Levels An experiment aims to study the dependency between the factors (or independent variables) and the response (or dependent) variables. The factors are the input variables set by the experimenters, whereas the response variables are the output. There exist two factors to manipulate through the experiments to understand their effect on the performance of DIME.

- **The time period** T: It is the rate at which DIME replenishes the instrumentation budget B. Studying the effect of the time period T on the slow-down factor and the instrumentation coverage is the primary objective of the experiments.
  Levels: In the experiments, the time period T has two levels: (1) low with $T = 0.25$ sec and (2) high with $T = 4$ sec. The relatively high difference between the levels of T can help to emphasize the impact of T on the response variables (if exists). In other words, the effect of T will be noticeable visually and statistically.

- **The instrumentation budget** B: It is the percentage of the time period T in which DIME allows the execution of the analysis routine. The value of the instrumentation budget B is specified during the system design process. However, we are interested in testing if an interaction exists between the instrumentation budget B and the time period T. An interaction is the case in which one factor a produces a different effect on the response variable at different levels of another factor b [73]. In other words, can the budget value B alter the impact of the time period T on the performance of DIME?
  Levels: The experiments operate with two levels of the instrumentation budget B: (1) low; $B = 2\%$ of T and (2) high; $B = 25\%$ of T. The low level of B represents a restrictive instrumentation budget, whereas the high level demonstrates a relatively relaxed one.

Response variables. A response variable is the measured outcome of the experiment. The response variables of interest in our case are as follows:

- **Slow-down factor of the dynamically instrumented program**: the ratio of the execution time of the instrumented program on top of DIME to the execution time of the natively running program. This measurement reflects the runtime overhead of dynamic instrumentation using DIME. To get the best performance of DIME, the experiments aim to find the value of T, with respect to B, that minimizes the slow-down factor.
- **Unique instrumentation coverage**: the ratio of the number of *unique* traces of DIME to that of native Pin. This value highlights the ability of DIME to provide quality information. The objective is to define the value of T, with respect to B, that provides the maximum unique coverage. The user of DIME should focus on this metric along with T if the analysis tool collects non-interesting redundancies, e.g., for code coverage or building call-context tree.
- **Raw instrumentation coverage**: the ratio of the total number of traces collected by DIME to that collected by native Pin. This metric, also, highlights DIME's ability to gather quality information but only when all the extracted traces are useful, i.e., no redundancy. The goal is to find the value of T, given B, that maximizes the raw coverage.

The ratio of the number of unique traces to that of the total traces highly depends on the implementation of the analysis tool and the formatting of the output traces. In this chapter, we are solely interested in the *impact* of DIME parameters on the number of unique traces and the total number of traces.

Analysis tool. To block the effect of the analysis-tool factor on the response variables, the experiment involves only one analysis tool: the branch profiler. As mentioned in Chapter 4, the branch profiler has a heavy-weight analysis routine. At run time, it extracts the jump, call, and return instructions out of the whole program (and not only a part of it). The branch profiler is a modified version of the *branch_target_addr* pintool that is available in Pin kit v2.14-71313.

Platform. The experiments run on three workstations of the same hardware and software specifications. Each workstation hosts a 64-bit dual 6-core Intel-Xeon CPU (E5-2620V3).

Each workstation has 15 MB of processor cache, 256 GB of RAM, and a CPU frequency of 3.2 GHz. The workstations are running Ubuntu 12.04 patched with the real-time kernel v3.12.66-rt89 to convert the kernel into a fully preemptive one. To further reduce the variability in the experimental results, we prohibit core migration such that an experiment process runs on a single core. We also lock core frequencies to the maximum value and increase the scheduling priority of the experiment processes. Note that we use a unified hardware and OS specifications to block the effect of the platform factor on the response variables [81]. Also, all the implementation aspects of DIME complies with the description in Chapter 3. Through the experiments, we use the *Trace Version* implementation of DIME without the redundancy suppression feature.

5.2.3 Benchmark Sets

To cover a wide range of software programs, the experiments include three program categories: CPU, IO, and memory intensive. We investigate whether the program category contribute to the impact of DIME parameters on its performance. The following are the three benchmark sets included in the experimentation to present the mentioned program categories:

1. **CPU intensive**: The SPEC 2006 benchmark suite is widely used in industry and academia to stress test a system's processor [46]. Our experiments use the complete SPEC C benchmarks which consist of twelve integer and floating-point programs. The SPEC C suite covers various domains as listed in Table 5.1. For some benchmark programs, the suite includes multiple input sets (see third column in Table 5.1). In such cases, the experiments execute the benchmark programs with each of the provided inputs.

2. **IO intensive**: IOzone is an I/O file-system benchmarking tool [5]. IOzone runs 13 tests as shown in Table 5.2. A single execution of a test runs 70 iterations, by default, covering a file-size range of 64 KB to 512 MB and a record-size range of 4 KB to 16 MB.

3. **Memory intensive**: Stress-ng is a testing benchmark tool for various physical subsystems such as CPU and cache [12]. The experiments use the virtual-machine (VM) Stress-ng benchmark set which exercises the system's memory. Ten of the VM benchmarks are able to run on our platform. The rest of the VM benchmarks are either not implemented for the experimental platform or failed to operate with dynamic instrumentation (of both native Pin and DIME). Table 5.3 describes Stress-ng benchmark

Table 5.1: CPU intensive: list of SPEC C benchmark programs

Benchmark	**Domain**	**Num. of Inputs**
Perlbench	Programming Language	3
bzip2	In-memory compression	6
gcc	C Language Optimizing Compiler	9
mcf	Optimization/Scheduling	1
gobmk	Artificial Intelligence/Game playing	5
hmmer	Search Gene-Sequence Database	1
sjeng	Artificial Intelligence/Game playing	1
libquantum	Physics/Quantum Computing	1
h264ref	Video Compression	3
milc	Physics	1
lbm	Fluid Dynamics	1
sphinx3	Speech recognition	1

programs and lists the number of operations per single execution. The numbers of operations set in the experiments equal to these performed in a two-minute native execution. The benchmarks execute sequentially with the option of no random seeding (*–no-rand-seed*) to ensure the reproducibility of the tests. Both native Pin and DIME run with the instrumentation option *follow_execv* to ensure the instrumentation of the child processes when exist.

5.2.4 Factorial Design

The experiments follow a 2^2 factorial design with high and low levels of both T and B. Through the experiments, we measure the values of the response variables (slow-down factor, unique coverage, and raw coverage). To reduce the experimental error, we run ten replications for each benchmark set. Each replication involves the four combinations of the levels of T and B.

An analysis-of-variance (ANOVA) statistical test analyzes the experimental results for each benchmark set and each response variable. ANOVA is useful for examining the significance of the input factors by testing whether the means of two or more populations are equal [73]. For each benchmark set, a factorial ANOVA was conducted to compare (1) the effect of the time period T, (2) the effect of the instrumentation budget B, (3) the effect of the benchmark program P, and (4) the interaction effect between T and B on the response variable.

Note that the benchmark program P is included in the ANOVA test as a blocking factor.

Table 5.2: I/O intensive: list of IOzone benchmark programs

Benchmark	**Description**
Write/Re-write	writing a newfile then writing to an existing file
Read/Re-read	reading an existing file then reading a recently-read file
Random read/write	reading and writing to random locations within a file
Random mix	Mixed reading and writing to random locations within a file
Read backwards	reading an existing file backwards
Re-write record	writing and re-writing a particular location in the file
Stride-read	reading a file with strided access behavior
fwrite/re-fwrite	writing and re-writing a file using fwrite()
fread/re-fread	reading an existing file and reading a recently-red file using fread()
pwrite/re-pwrite	writing and re-writing a file using pwrite()
pread/re-pread	reading an existing file and reading a recently-red file using pread()
pwritev/re-pwritev	writing and re-writing a file using pwritev()
preadv/re-preadv	reading an existing file and reading a recently-red file using freadv()

Table 5.3: Memory intensive: list of Stress-ng benchmark programs

Benchmark	**Description**	**Num. of Operations**
bigheap	Heap re-allocation	3,590,698
mremap	calls of mmap(), mremap(), and munmap()	807
brk	iterations of expanding the data segment by one page	64,860,767
msync	Data synchronization between file and memory	1,241,716
mmapfork	Forking 32 child processes, each allocating memory	2
stackmmap	flushing dirty pages of a 2MB stack using msync()	3,540
vm	mmap()/munmap() calls and writing to the allocated memory	1,592,852
vm-rw	Memory transfer between child and parent processes	12,439
vm-splice	data transfer from memory to /dev/null through a pipe	44,247,682
malloc	repeatedly allocate, reallocate, and free memory	100,000,000

The slow-down factor and the coverage of each program are different. Thus, the program adds to the variability of the response variables. We must account for this variability when conducting the ANOVA test although P is not a variable of interest.

5.3 Experimental Results

This section discusses the interpretation of the ANOVA test results and reflects on the hypotheses mentioned in Section 5.2.1.

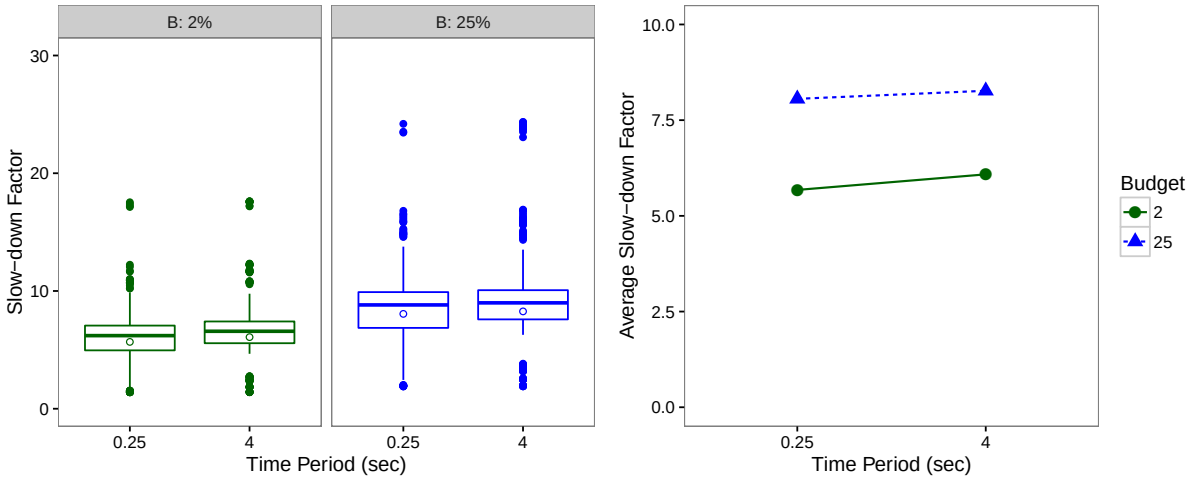
5.3.1 ANOVA Test Results

Recall that we use ANOVA to test the significance of the main effect of the time period T and the interaction factor (between T and B) on the response variables. The significance is decided based on the P-value at the 0.01 significance level. Appendix A lists the detailed ANOVA result tables of all the conducted tests. First, we list the ANOVA test results of the three response variables and for all the benchmark sets. Then, we discuss the interpretation of the results in details.

1. Slow-down Factors

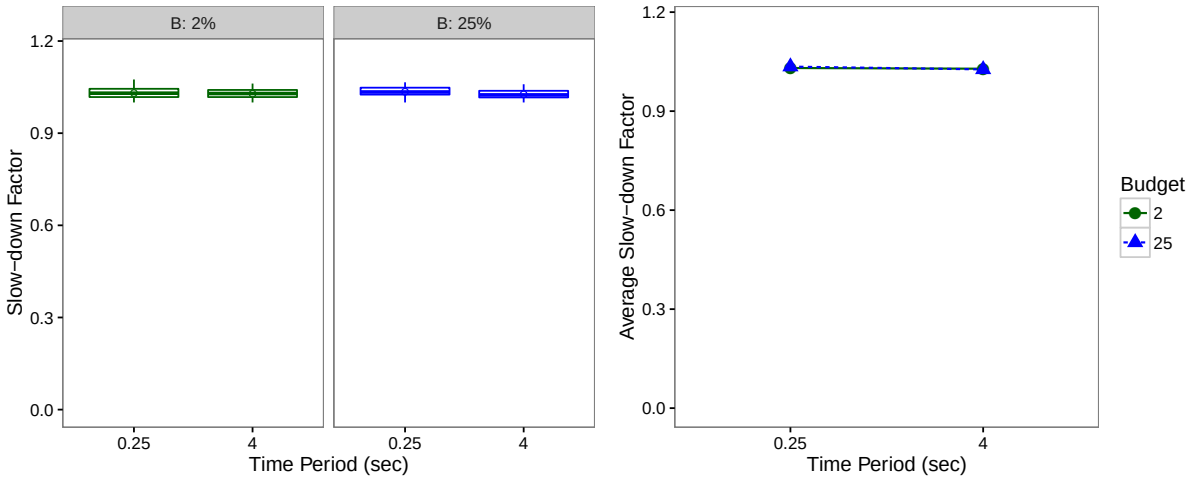
Figures 5.1a, 5.2a, and 5.3a summarize the distributions of the slow-down factors of SPEC, IOzone, and Stress-ng benchmarks respectively. The x-axis shows the time-period value T, and the y-axis represents the slow-down factors of the benchmark programs. The data of the low budget value (2%) is plotted in green while that of the high budget (25%) is shown in blue. Figures 5.1b, 5.2b, and 5.3b show the interaction effect between T and B on the slow-down factors of SPEC, IOzone, and Stress-ng benchmarks in order. The y-axes of the interaction plots show the geometric means of the slow-down factors at the different levels of T and B. Parallel lines in an interaction plot indicate the insignificance of interaction between factors.

The main effect of the time period T on the slow-down factor is statistically significant for the three benchmark sets. Also, the interaction factor is statistically significant for IOzone and Stress-ng. As the parallel lines in Figure 5.1b indicate, the interaction factor is insignificant for SPEC benchmarks.



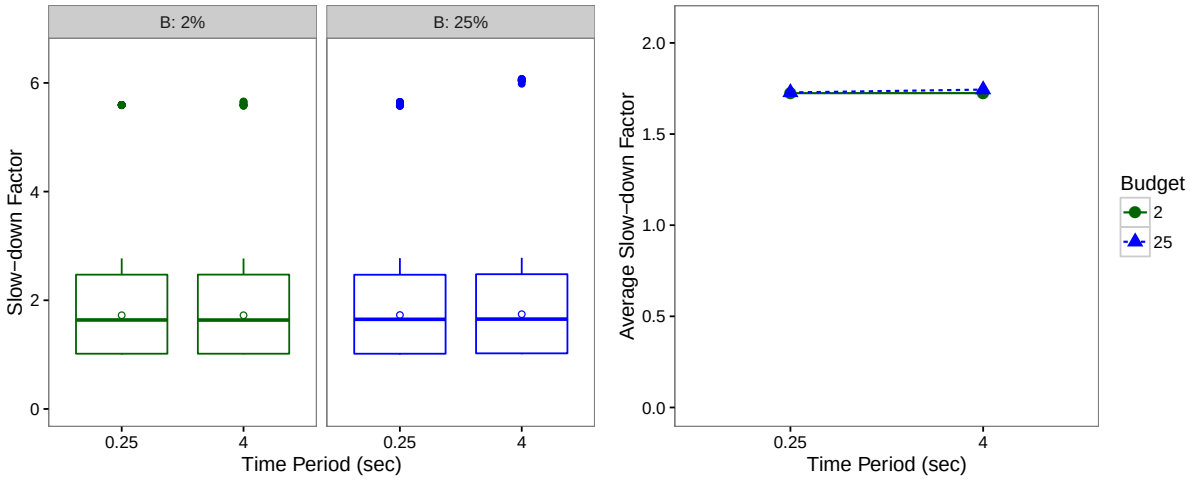
(a) Box Plot (b) Interaction Plot

Figure 5.1: SPEC slow-down factors



(a) Box Plot (b) Interaction Plot

Figure 5.2: IOzone slow-down factors

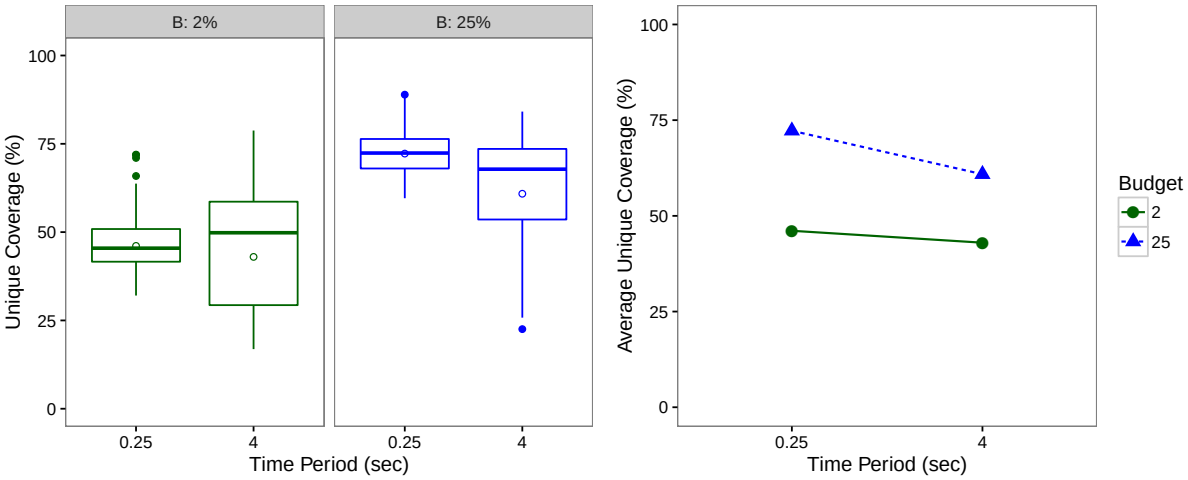


(a) Box Plot (b) Interaction Plot

Figure 5.3: Stress-ng slow-down factors

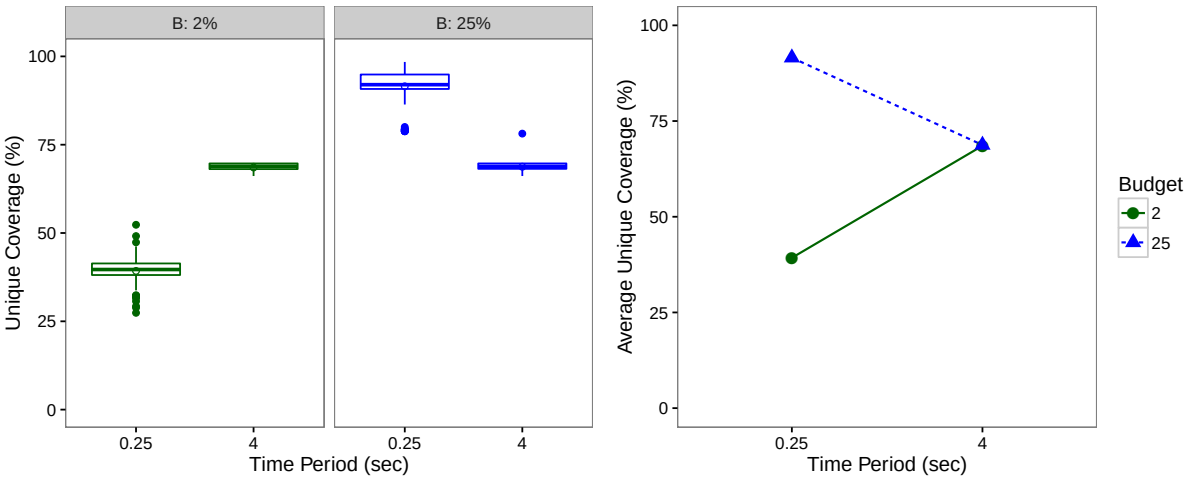
2. Unique Instrumentation Coverage

Both effects of the time period T and the interaction factor on DIME unique instrumentation coverage are statistically significant. Figures 5.4, 5.5, and 5.6 show the box plots and the interaction plots describing the unique coverage of SPEC, IOzone, and Stress-ng consecutively



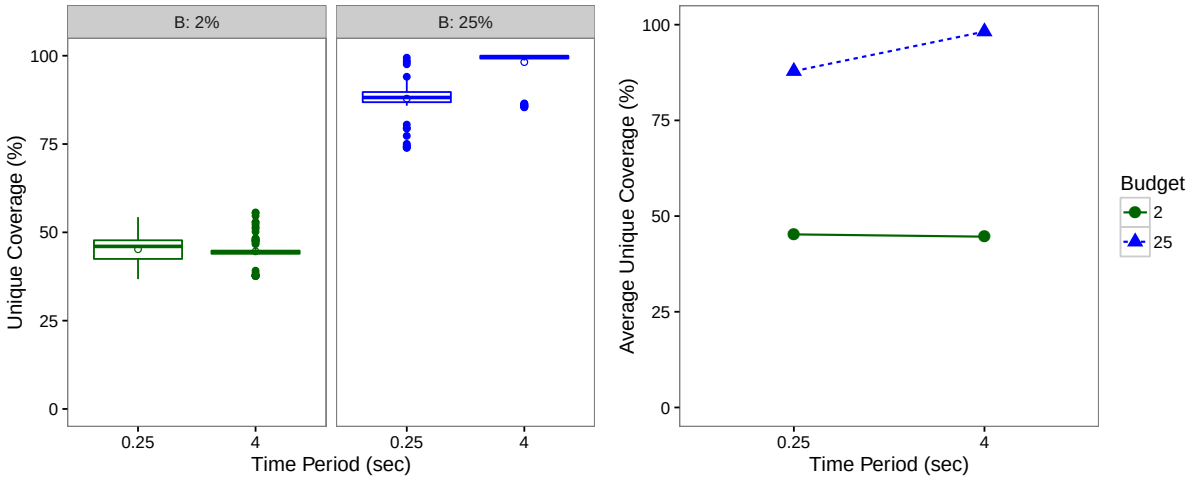
(a) Box Plot (b) Interaction Plot

Figure 5.4: SPEC unique instrumentation coverage



(a) Box Plot (b) Interaction Plot

Figure 5.5: IOzone unique instrumentation coverage

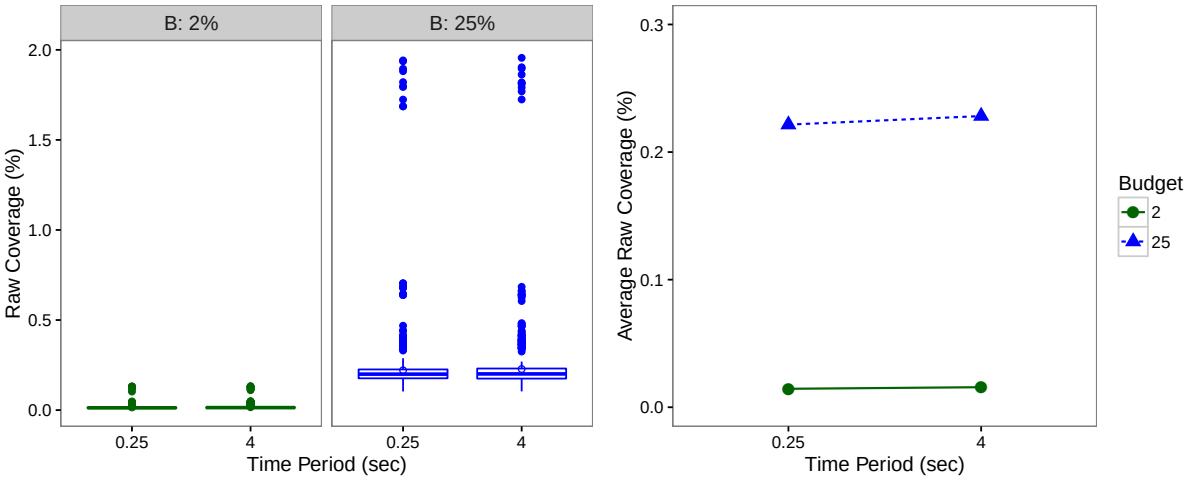


(a) Box Plot (b) Interaction Plot

Figure 5.6: Stress-ng unique instrumentation coverage

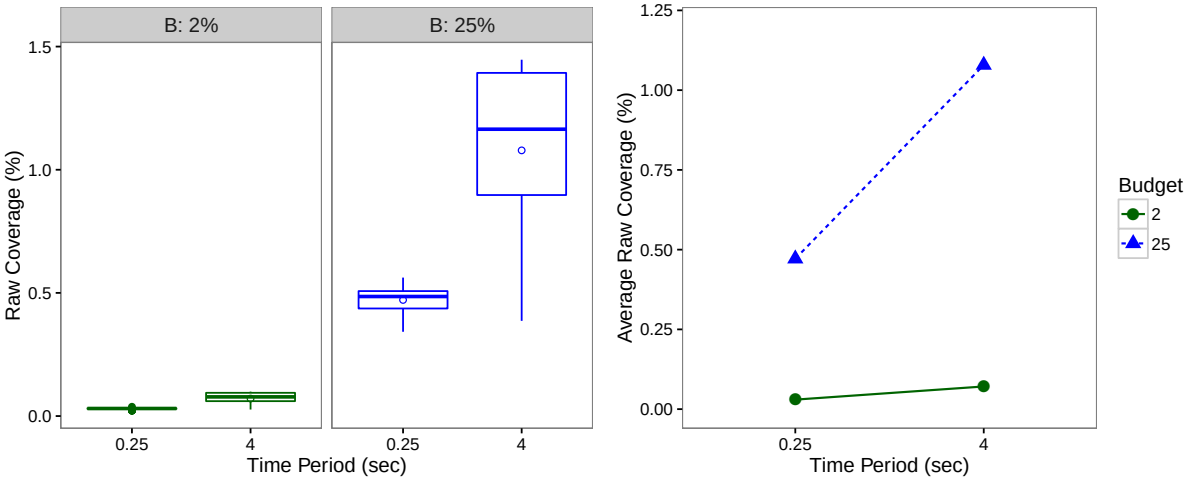
3. Raw Instrumentation Coverage

The ANOVA test proved the statistical significance of the time-period effect and the interaction factor on the raw instrumentation coverage of IOzone and Stress-ng. On the other side, the mentioned effects are insignificant on SPEC's raw coverage. The summaries of the raw-coverage results are shown in Figures 5.7a, 5.8a, and 5.9a and the interactions appear in Figures 5.7b, 5.8b, and 5.9b for SPEC, IOzone, and Stress-ng in order.



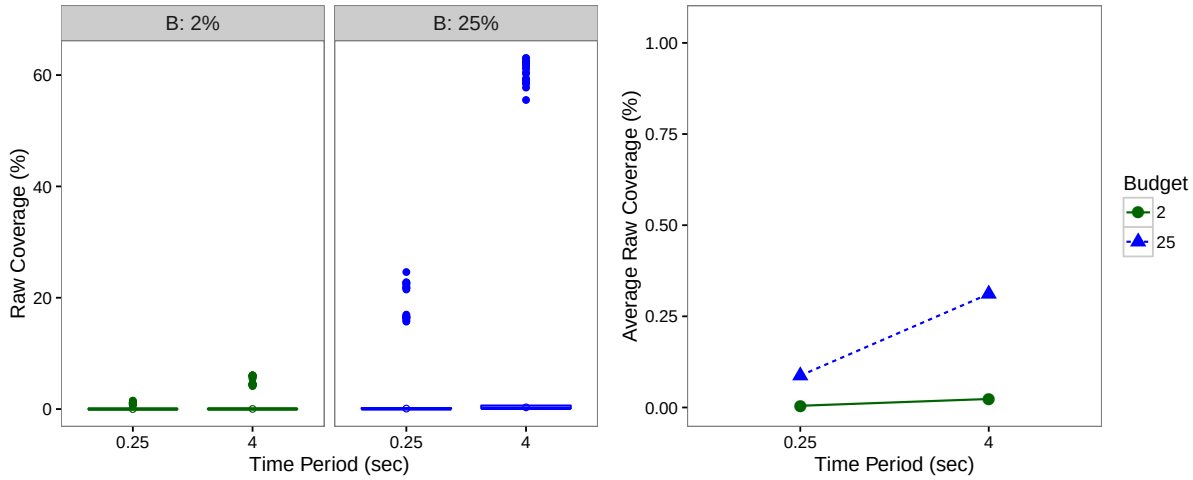
(a) Box Plot (b) Interaction Plot

Figure 5.7: SPEC raw instrumentation coverage



(a) Box Plot (b) Interaction Plot

Figure 5.8: IOzone raw instrumentation coverage



(a) Box Plot (b) Interaction Plot

Figure 5.9: Stress-ng raw instrumentation coverage

5.3.2 Discussion

Hypothesis 1: Fail to reject the null hypothesis.

The runtime overhead of CPU-intensive programs is independent of the time-period value of DIME. The ANOVA test proved the significance of the time-period effect on SPEC's slow-down factor (Figure 5.1). However, a deeper look into SPEC's results reveals that the ANOVA test was biased by the slow-down factors of one benchmark program: *h264ref*. For all SPEC benchmark programs except *h264ref*, the time period shows no effect on the slow down factor. By excluding *h264ref* and re-conducting the ANOVA test, both the main effect of the time period and the interaction factor are insignificant.

Although the impact of the time period T on the slow-down factors of IOzone is statistically significant, it is practically negligible. In all cases, the slow-down factors of IOzone on top of DIME falls below 1.1x with a maximum of 1.07x. This implies that in practice, the user of DIME can ignore the impact of the time period value and the interaction factor on the runtime overhead of IO-intensive programs.

Similarly, it is practically possible to ignore the impact of the time period and the interaction factor on the slow-down factors of memory-intensive programs. As the boxplot in Figure 5.3a shows, the time period and the interaction factor do not impact the slow-down factors of Stress-ng except for one benchmark program *vm* which appears as outliers

in the plot. The re-conduction of the ANOVA test without the slow-down values of the *vm* program results in the insignificance of the time-period effect and the interaction factor.

According to the previous observations, we fail to reject the null hypothesis H_0^s. There exists no correlation between the time period T and the slow-down factor of the instrumented program on top of DIME. The low time period of $T = 0.25$ seconds increases the worst-case version-switching frequency by 16 times compared to the high time period of $T = 4$ seconds. However, the instrumented programs do not suffer from increased runtime overhead in this case. This may indicate the efficiency of the version switching mechanism of DIME.

Hypothesis 2: Fail to reject the null hypothesis.

We expect the unique instrumentation coverage to increase as the time period decreases. A high time period keeps the budget available for longer time but in less frequency. DIME may consume the budget extracting traces from the same code portion. The principle of locality states that programs tend to re-execute recently executed instructions [83]. Contrarily, a low time period will force the budget to be distributed over the program execution. This will increase the probability of DIME extracting unique traces. The unique coverage results of SPEC complies with this scenario. As Figure 5.4 shows, there exists a negative correlation between the unique coverage and the time period value.

The instrumentation of the SPEC benchmark results in much higher trace redundancies than that of IOzone and Stress-ng. This is based on the unique coverage and the raw coverage results of native-Pin instrumentation of the three benchmark sets. This observation may justify the different correlations seen in Figures 5.5 and 5.6 for IOzone and Stress-ng respectively. Infrequently executed instructions (or cold code) may benefit from increasing the time period. An initialization function *Init()*, for example, may run only once during the program execution. With a low time-period, the available budget during the execution of *Init()* may be insufficient to extract all the instructions of interest,i.e., the branches in our experiments. Whereas, a high time-period value increases the probability of DIME extracting all the required instructions due to the availability of the budget for longer. The unique coverage of Stress-ng, with high B, has a positive correlation with the time period. Similarly with low B, the unique coverage of IOzone positively correlates with the time-period value. As a result, we fail to reject the null hypothesis H_0^u.

The interaction factor is showing a significant effect on the unique coverage of IOzone (Figure 5.5). The budget B changes the impact of the time period T such that T has a negative correlation with the unique coverage in the case of high B, but a positive

correlation in the case of low B. It is, also, interesting to notice in Figure 5.5b that, with the same time period, low and high budget values result in equal unique coverage on average. This is an example of the instrumentation budget that is wasted extracting redundant traces.

From a different point of view, we can notice the interesting effect of the budget B on both the runtime overhead and the unique instrumentation coverage. In general, increasing the budget B results in a substantial increase in the unique coverage with a limited overhead sacrifice. In the results of Stress-ng benchmark for example, changing the budget from 2% to 25% causes an average unique-coverage increase of 47.5% with almost no added runtime overhead.

Hypothesis 3: Reject the null hypothesis.

The raw-coverage results recommend rejecting the null hypothesis H_0^r for IO and memory intensive applications. In other words, the time period positively correlates with the raw coverage of Stress-ng benchmark programs. Similarly, a high value of the time period will increase the raw instrumentation coverage of IOzone. This is useful if the user of DIME is interested in the full output of the analysis tool. In the case of a data-cache simulator, for instance, the user should opt for a high time period. On the other side, we observed no effect of the time-period value on SPEC's raw instrumentation coverage.

5.4 Guidelines and Limitations

This chapter along with the previous two introduced the concept and the design aspects of the time-aware dynamic instrumentation tool DIME. This section discusses the usability, the work flow, and the limitations of DIME.

DIME is a dynamic instrumentation tool that respects timing constraints. DIME is useful through multiple stages of the software development process. The developer can use DIME to debug the software system by collecting runtime information that describes the program execution. Examples of such information include memory access patterns, variable traces, and register profiles. Also, DIME is a beneficial performance-engineering tool. In other words, the developer can collect runtime information to evaluate the performance of the system and optimize performance bottlenecks. Moreover, DIME can contribute to runtime monitoring and verification frameworks. The authors in [54] utilized DIME as the system monitor to verify Linear Temporal Logic (LTL) properties of software systems.

Being thread safe, DIME is useful for the debugging and the analysis of multi-threaded applications. In this case, it is possible to divide the budget among the running threads based on their priorities for example. Additionally, multiple independent instances of DIME can run simultaneously on the same platform. Thus, the developer may use DIME to understand the interaction between multiple applications running on the same platform.

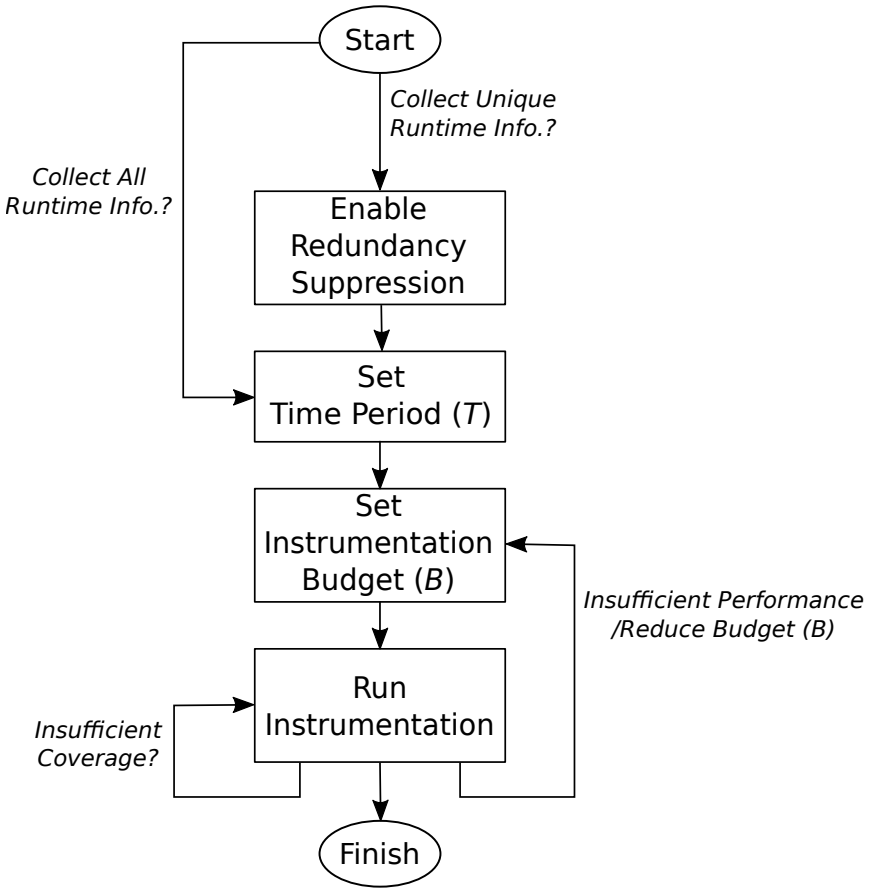


Figure 5.10: Workflow Diagram of DIME

DIME is a fully-implemented analysis tool that requires only the availability of the program's executable and not the source code[1]. Similar to Pin, DIME requires the creation of an analysis tool (pintool) that contains the implementation of the instrumentation and the analysis routines. Additionally, DIME needs the developer to set the values of the instrumentation parameters (or keep their default values). Figure 5.10 shows the work

[1]DIME is available for download at https://github.com/pansy-arafa/DIME

flow of using DIME to instrument a program. First, the developer can enable the redundancy suppression feature to force DIME to collect only unique traces. The second step is setting the value of the time period (T). It is advised to keep the default value of one second since the experiments in this chapter revealed no effect of the time-period value on DIME's runtime overhead. However, the developer may increase the value of T if he is interested in increasing the raw coverage. Third, the developer should set the value of the instrumentation budget B, according to the system design documents, or use the default value of 10% of T. The final step is to execute the instrumentation, i.e., run the program on top of DIME. If the performance of the program suffers during instrumentation, the developer should reduce the instrumentation budget and re-execute DIME. Finally, since DIME collects a partial amount of runtime information, multiple runs of DIME may be required to collect sufficient instrumentation coverage. Turning on the redundancy suppression feature can highly reduce the number of DIME runs. Note that the actions of setting the parameters of DIME and turning on/off the redundancy suppression feature are easily achieved through command-line parameters.

The following is the list of limitations of the time-aware dynamic binary instrumentation:

- Runtime overhead. Although DIME limits the runtime overhead of dynamic instrumentation, it does not guarantee zero overhead. In other words, some applications can be highly sensitive to dynamic instrumentation. If the program's performance suffers on top of Pin without any pintool, then a less intrusive analysis technique can be more suitable in this case, such as sampling or time-aware static instrumentation.

- Memory footprint. Dynamic instrumentation naturally requires the availability of sufficient RAM memory to perform dynamic compilation during execution. Also, there should be enough memory storage space to store the runtime information collected by DIME. The size of such information can reach multiple gigabytes of data.

- Security. DIME can instrument a program only if its security specifications allow the attachment of DIME to the program's process. For example, the child processes of the Google Chrome browser prohibit the attachment to DBI tools since they execute within a restrictive environment.

- Multiple runs. The redundancy suppression feature may require multiple runs of DIME. Although the program inputs should be the same in each run, changes in the processor state may lead to a minor difference in program execution. A different processor state among runs can, for example, alter the execution of shared libraries.

5.5 Summary

This chapter discussed the conducted experiments to evaluate the impact of DIME's parameters on its runtime overhead and instrumentation coverage. The experiments involve three benchmark sets; CPU, IO, and memory intensive applications. ANOVA statistical tests judges the effects of the time-period and its interaction with the budget value on the performance of DIME. The value of the time-period has negligible effect on the runtime overhead suggesting the efficiency of DIME's implementation. The parameters of DIME have significant impact on the unique instrumentation coverage. In general, unique instrumentation coverage is negatively correlated with the time-period value. However, there exist exceptions due to the rate of trace redundancies in the instrumented applications. Finally, the raw instrumentation coverage has a positive correlation with the time-period value of DIME.

Chapter 6

QoS-Aware Dynamic Binary Instrumentation

In this chapter, we present QDIME: a QoS-aware dynamic binary instrumentation tool [19]. QDIME respects user-defined thresholds and constraints to guarantee an acceptable system performance during instrumentation. Four case studies of QDIME running *Gzip*, *MySQL*, *Apache*, and *Redis*, demonstrate its practicality and scalability.

6.1 Motivation

The number of businesses relying on systems with quality of service (QoS) has been rapidly increasing. Software systems behind these businesses must meet strict performance requirements to satisfy the needs of both the provider and the end-users. These systems include web servers, database management systems, multimedia applications, web browsers, and hypervisors. QoS performance requirements refer to the extra-functional (non-functional) aspects of the system that can affect the user's experience [34]. These requirements include the total volume of computation, end-to-end delay, error rate, response time, and jitter [87]. For example, the production database management system at Facebook should be able to handle roughly 60 million queries per second [36], and its key-value store, billions of requests per second [80]. In this chapter, the term QoS denotes all extra-functional aspects of a system which may be used by end-users to judge the quality of a service.

Instrumentation is useful to profile and debug QoS systems. Many QoS performance requirements are time-related, for example, response time (task completion time), jitter,

and task synchronization [87]. Instrumentation of QoS systems can alter these performance requirements due to the timing delay introduced by instrumentation. This can result in undesired degradation in overall system performance. Also, that delay can change the system behavior; thus the extracted tracing data can be misleading. Consider for example profiling a media player while playing a video, instrumentation will add delay to the execution. The media player will fail to decode a number of frames and will drop these frames resulting in an unwatchable video. Moreover, different code portions are executed compared to the expected execution, and therefore, extracted traces fail to represent the expected system behavior. Another example is profiling a web server. Instrumentation can dramatically increase the response time, i.e., the time between sending a request by the user and receiving a response by the web server.

A QoS-system developer should be able to profile the expected execution of the system while maintaining the desired performance guarantees. As discussed in Sections 3.4 and 4.6, DIME allows instrumentation of QoS systems while keeping acceptable performance level. However, DIME requires the developer to specify the allowed amount of instrumentation time per time period. For many QoS systems, it is complicated to map the performance metric to instrumentation time. In other words, what value for instrumentation time will result in acceptable performance measures? The answer to this question can vary based on the QoS system type (e.g., web server, control application, media player) and the system's performance metrics. Note that QoS systems, in general, may consist of millions of lines of code and can have a high number of libraries dependencies. Accordingly, dynamic instrumentation is more practical and scalable than static instrumentation to profile and debug these systems. Thus, a flexible, customizable, scalable QoS-aware DBI tool is required.

6.2 Overview of QDIME

QDIME is a QoS-aware DBI technique that guarantees a certain QoS level to the program under analysis [19]. As a modified version of DIME, QDIME considers customized QoS performance constraints instead of timing constraints. QDIME allows the user to define a QoS related metric with a threshold value. To meet the performance requirements, QDIME periodically switches instrumentation on and off. Recall that, the instrumentation budget is the amount of time, per period, during which instrumentation is enabled. QDIME is responsible to periodically determines the instrumentation budget as a function of the current QoS-metric value. The function guarantees that the budget varies with the quality of the application and eventually falls to zero when the QoS constraints are violated. Thus,

in every period T, the budget, B, is computed to satisfy $0 \leq B \leq T$.

QDIME consumes its budget by subtracting from its value the time it takes to execute the analysis routine. Thus, it instruments as long as $B > 0$, with a degraded latency as a result. Once QDIME fully consumed the budget, i.e., $B \leq 0$, it disables instrumentation, which produces an increased QoS. To accurately determine the budget, the values of the QoS metric are periodically extracted from the instrumented program and fed back into QDIME. Ideally, this metric extraction should be transparent and unintrusive, requiring no or little modifications to the application. Fortunately, most existing applications already output different statistics that can directly be used by QDIME. In our experiments, we show that this is achievable with applications such as *MySQL*, *Gzip*, *Apache* server, and *Redis*. QDIME also utilizes the redundancy suppression feature discussed in Chapter 4 to guarantee the unicity of the extracted information. The following summarizes the features of QDIME:

- *QoS Guarantees*: QDIME takes into account the performance requirements to limit the latency degradation of the program.
- *Low Overhead*: QDIME guarantees a reduced runtime overhead by switching the instrumentation on and off.
- *High Unique Coverage*: The redundancy suppression feature of QDIME allows it to extract only new information through multiple runs.
- *Flexibility*: The QoS metric, along with its source, is customizable by the user of QDIME. Also, the thresholds, the time period, and the constraints are entirely defined by the user. He can then tune the parameters to adjust the trade-off between information gain, QoS, and overhead.
- *Practicality*: producing a high unique coverage with a significantly reduced overhead allows QDIME to enable the instrumentation of QoS-based applications and the design of QoS-aware analysis tools.
- *Portability*: QDIME is a generic technique that can be implemented on top of other instrumentation frameworks than PIN. Further, QDIME is independent of system's hardware-level features.

To instrument a program atop of QDIME, the user sets the instrumentation period and defines the QoS metric of the program. Further, the user is required to set the threshold and the constraints that are to be satisfied by the metric to guarantee an acceptable QoS.

6.3 Design Architecture

QDIME periodically monitors the QoS state to ensure that the quality of the instrumented program does not degrade to violate the threshold. Thus, QDIME needs to maintain a constant knowledge on the evolution of the QoS metric throughout the entire instrumentation process. QDIME achieves this by periodically extracting the QoS data from the program under analysis. To make this extraction process transparent and unintrusive, QDIME relies on performance data generated by the program itself. This reliance is possible because most, if not all, programs with QoS requirements already provide a mechanism to expose internal performance statistics. QDIME parses and consumes this data to make its instrumentation decisions. Even for programs that do not propose such a mechanism, in our experiments, we found it straightforward to add an extension that exposes QoS statistics.

Figure 6.1 shows the architecture of QDIME. The application runs on top of QDIME. The metric analyzer periodically collects the performance statistics generated by the program. It then parses and performs any user-defined computation on the data to generate QoS metrics in a format accessible by QDIME. Finally, the analyzer writes the QoS data in a memory area it shares with QDIME. Ideally, the frequency at which the metric analyzer updates the QoS metric should match the instrumentation period T, although this is not a requirement. This architecture allows QDIME to handle applications out-of-the-box, without binary modification.

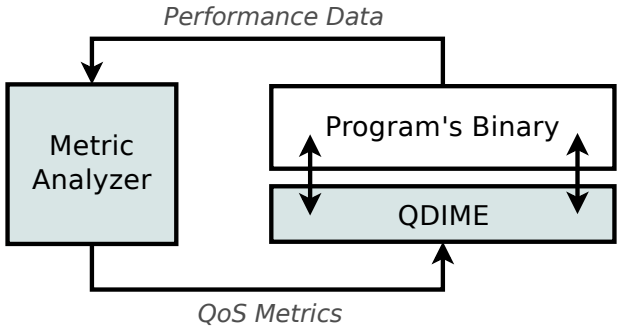


Figure 6.1: Architecture of QDIME

6.4 Budget Function

Similar to DIME, QDIME replenishes the instrumentation budget at the beginning of each time period T. QDIME calculates the value of the budget as a function of the QoS-metric

value, contrary to DIME which resets the budget to a constant value. QDIME's budget function `compute_Budget()` maps $\mathbb{R}^+$ into [0,T], where T is the instrumentation period. In general, the budget function can be built as a decreasing function using the insight that a program produces high QoS when the budget is set to zero, i.e., when instrumentation is disabled. Also, the QoS decreases as the budget increases.

Equation 6.1 represents the budget function of QDIME such that (1) $q(t)$ is the QoS-metric value of the instrumented program at time t, (2) X is the user-defined QoS threshold, and (3) T is the time-period value. The function calculates the budget based on the QoS slack with respect to the user-defined threshold. The QoS slack at time t is the difference between the current QoS level and the threshold; $q(t)-X$. This value indicates the health of the instrumented program and can be used to safely determine the budget value at time t.

The budget at time t is set to zero whenever the QoS constraint is violated, i.e., QoS below the threshold X in this case. Otherwise, the function always produces a budget $b(t) < T$. Instead of zero, the user could also choose a default value to use when QDIME fails to meet the threshold condition. To get the budget value in seconds, the function multiplies the QoS slack by the time-period value and by a tuning factor. Since respecting the QoS threshold is the primary objective, QDIME adopts a conservative budget function by choosing the tuning factor of $1/(q(t)+X)$. As later shown in Section 6.6.3, the budget function of Equation 6.1 enables QDIME to successfully avoid threshold violations while providing both high unique coverage and reduced overhead.

$$b(t) = \begin{cases} 0 & \text{if } q(t) < \text{X} \\ \dfrac{q(t)-X}{q(t)+X} \times \text{T} & \text{otherwise} \end{cases} \tag{6.1}$$

The example in Figure 6.2 shows how the periodic replenishment of QDIME budget during the extraction of system calls from *Apache* server. The x-axis shows the execution time of *Apache* in seconds. The left y-axis represents the budget value in seconds, while the right y-axis shows the QoS-metric value (number of requests per second). For instance around t=10 sec, when *Apache* processes about 4,000 RPS, which is above the threshold of 2,000 RPS, the budget is set to approximately 35% of the one-second instrumentation period. Around t=55 sec, the QoS drops to 3,050 RPS forcing QDIME to adjust its budget to only about 20% to meet the threshold.

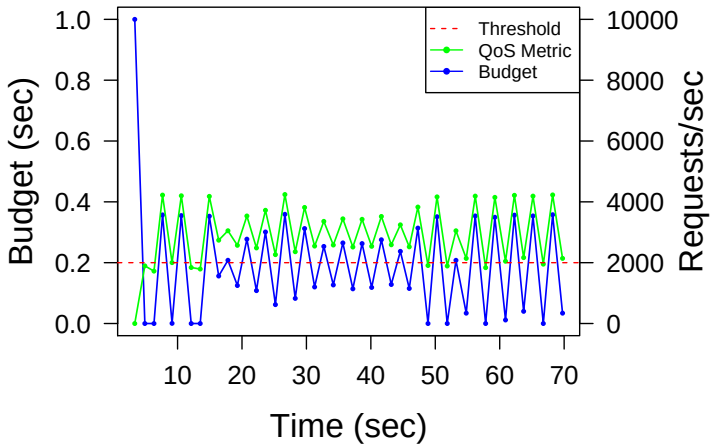


Figure 6.2: Apache: instrumentation budget vs. QoS metric

6.5 Implementation

The implementation of QDIME complies with that of DIME explained in Chapter 3. Listing 6.1 shows the core algorithm of QDIME. The mechanism of budget checking and version switching in the instrumentation routine follows the *Trace Version* implementation of DIME (Line 9). At each instrumentation point, QDIME can either allow or prohibit the insertion of the analysis routine based on the dynamic budget check at Line 13. During instrumentation, the time consumed by the analysis routine is subtracted from the available budget (Line 6). Whenever the budget falls to or below zero, QDIME turns off instrumentation, i.e., prevents insertion and execution of analysis routines, leading to increased QoS and speedup as a result.

The signal handler of QDIME (Line 36) fires periodically at T to execute the function `compute_Budget()`. This function reads the QoS-metric value from the shared memory and replenishes the budget according to the equation listed in Section 6.4. The function `compute_budget()` generates a new budget with respect to the health status of the program. Thus, QDIME instruments aggressively when the system is healthy enough by producing a higher budget, but also limits its intrusion by generating a lower budget as the QoS degrades. If the user-defined threshold is violated, the budget is set to zero to prevent further instrumentation.

```
1  void analysis(...){
2      time_start = get_time();
3      //Execute instrumentation code
4      ...
5      time_end = get_time();
6      budget -= time_end - time_start;
7  }
8  ...
9  void instrumentation(...){
10     is_new = check_redundancy_log();
11     if(is_new){
12         For each instrumentation point{
13             InsertCall(budget_check);
14             if(version == V_BASE) {
15                 //check switching to V_INSTRUMENT
16                 InsertVersionCase(1,V_INSTRUMENT);
17             }
18             else if(version == V_INSTRUMENT){
19                 //check switching to V_BASE
20                 InsertVersionCase(0,V_BASE);
21             }
22             switch(version) {
23                 case V_BASE:
24                     break; //Do Nothing
25               case V_INSTRUMENT:
26                     ...
27                     InsertCall(analysis_routine);
28                     ...
29                     break;
30             }
31         }
32         update_redundancy_log();
33     }
34 }
35 ...
36 void sig_handler(...){
37     budget = compute_Budget(qos);
38 }
```

Listing 6.1: Instrumenting with QDIME

QDIME utilizes the redundancy suppression feature described in Chapter 4 to extract unique, i.e., non-redundant run-time information. As shown in (Line 10), QDIME searches the hash-table log before the instrumentation of a trace. If τ_i is the current trace and τ_j is the trace already in the log L, QDIME instruments τ_i only if $address(\tau_i) \neq address(\tau_j), \forall \tau_j \in L$. After instrumenting a trace, QDIME saves the trace relative address to the log. Multiple runs of QDIME may be required to, optimally, achieve full coverage. In this case, the log file is passed across runs, allowing QDIME to reveal only new information during each run. Although useful, the redundancy suppression is an optional, and not a fundamental, feature of QDIME.

Note that QDIME is a fully-implemented analysis tool[1]. It shares the same work flow, use cases, and limitations as those of DIME (listed in Section 5.4).

6.6 Performance Evaluation

This section describes the experimental setup and discusses the results of QDIME instrumenting four popular real-world applications. Note that all the average values mentioned in Section 6.6.3 are geometric means.

6.6.1 Experimental Setup

The experimentation environment consists of two workstations each hosting a 64-bit quad-core i7-2600 processor clocked at 3.4 GHz with 16 GB of RAM and 8 MB of cache. Each workstation runs Ubuntu 12.04 patched with the real-time kernel v3.2.0-23 that converts Linux into a fully preemptive kernel. We prevent task migration between cores, lock each core to its maximum frequency, and run the experiments with increased scheduling priority. Although these settings are not necessary for QDIME, they lead to less variance in the results [81]. We used the following analysis tools taken from the PIN toolkit version 2.14-71313 and compiled using `gcc` 4.6.3 with -O3 optimization level. The tools are ordered from the less to the most intrusive.

1. *Sys-trace*: extracts system function calls. The system-call traces are important for debugging and discovering performance bottlenecks in a program.

[1] QDIME is available for download at https://github.com/pansy-arafa/qdime

2. *Call-trace*: outputs the list of function calls with corresponding instruction addresses. Call traces are used to build call-context trees, which are useful in performance analysis and runtime optimizations [88].

3. *Branch-profile*: prints out the jump, call, and return instructions in addition to the source and destination addresses. The output of this tool is useful for exploring code coverage for example.

To evaluate its applicability and scalability, we run four real-world applications on top of QDIME. We repeated each experiment ten times (1) natively, (2) with PIN, and (3) with QDIME. Since QDIME aims to maximize the coverage while respecting the QoS threshold and minimizing the overhead, our experiments empirically evaluate the following metrics:

- *QoS performance metric:* the values of the user-defined QoS metric over time. This metric measures the ability of QDIME to respect the user-defined threshold during instrumentation.

- *Slowdown factor of the instrumented application*: the ratio of the execution time of the instrumented application to its native execution time. This metric evaluates the runtime overhead of QDIME.

- *Unique instrumentation coverage:* defined as the ratio of the amount of non-redundant information extracted by QDIME to that extracted by PIN. To illustrate, a 100% coverage means that QDIME extracts the same number of non-redundant traces as PIN. This metric highlights the ability of QDIME to extract quality information.

Based on the above, our experiments are to verify that (1) instrumenting with PIN highly degrades the quality of the application, (2) QDIME always maintains a higher QoS w.r.t. that of PIN, (3) QDIME is able to meet the defined QoS threshold, (4) QDIME reduces the application's slow-down factor introduced by DBI, and (5) QDIME is capable of extracting sufficient information, i.e., provide high unique coverage.

6.6.2 Benchmark Applications

This subsection describes the benchmarks and the experimentation objectives. In general, QDIME can instrument QoS applications whose security permits the attachment of a DBI framework. For example, the child processes of the *Google Chrome* browser execute within a restrictive environment and accordingly prohibit the PIN attachment. The instrumentation

period in the experiments is set to $T = 1sec$ for all the applications. Different values of the QoS threshold will be discussed later in Section 6.6.3.

***Gzip* Compression Utility** *Gzip* is a widely used data-compression application of about 59,000 lines of code adopted by the GNU project [3]. Our experiments aim to instrument *Gzip* v1.4 while maintaining a compression rate threshold of 1 MB/sec. *Gzip* compresses the Linux kernel 4.1.1 file whose size on the hard disk is 569 MB and generates a compressed file of size 121 MB using the default compression flags. QDIME's metric analyzer periodically monitors the size of the output file to compute the compression rate.

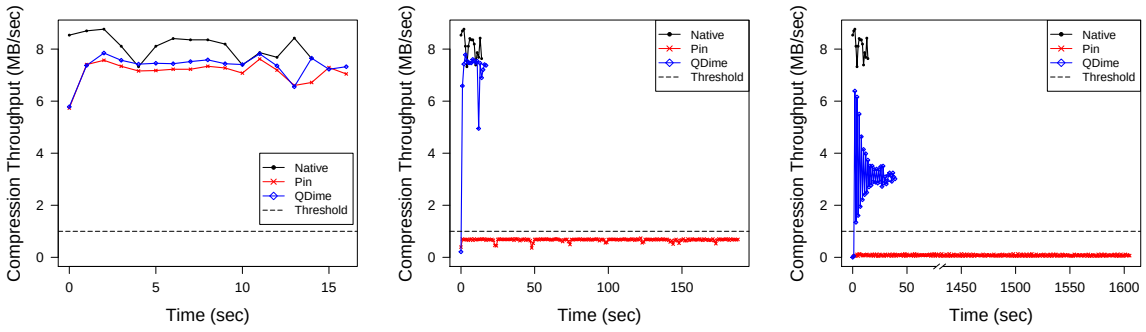
***MySQL* Server** *MySQL* is a popular, fast, scalable, and reliable relational database management system. *MySQL* is a multi-threaded system of over 1.5 million lines of code [6]. For the experiments, we installed *MySQL* 5.5.43 and SysBench 0.4.12, a benchmarking tool for databases [62]. We configured SysBench to run an OLTP test with a *MySQL* database of 1 000 000 records. The test simulates 10 users performing a total of 100 000 requests. This setup results in the execution of a total of 2 100 000 database queries. The goal of the experiments is to instrument *MySQL* server, under the above-mentioned workload, and keep a QoS threshold of 1000 transactions per second (TPS). The metric analyzer utilizes *MySQL* APIs to access the server status variables and compute the number of TPS.

***Apache* HTTP Server** *Apache* [2] is a powerful and popular web server of about 1 700 000 lines of code that implements the latest HTTP protocols. The experiments intend to instrument an active *Apache* server, i.e., version 2.4.16 in this case. The workload consists of 200 000 HTTP requests from 10 concurrent users generated with the *Apache* benchmarking tool `ab 2.3` [1].

Although QDIME is capable of handling an application's child processes, the *Apache* child processes refused the attachment of both PIN and QDIME. Therefore, the experiments run *Apache* in debug mode, which forces the server to run as a single process. To respect a QoS threshold of 3000 requests per second (RPS), the metric analyzer monitors *Apache*'s log files to determine the number of requests executed per second.

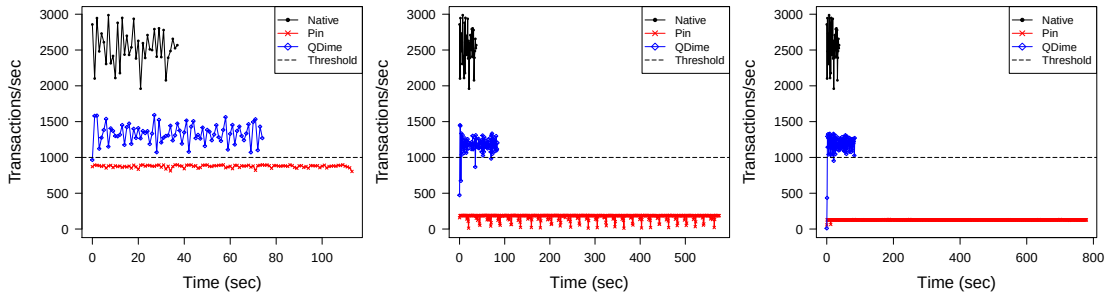
***Redis* Data Structure Server** Written in ANSI C, *Redis* is an in-memory, NoSQL database management system, with optional persistence capabilities [11]. *Redis* consists of roughly 20 000 lines of code. Many well-known projects such as Twitter, GitHub, and Pinterest, rely on *Redis*.

We installed *Redis* server 3.0.3 and used *redis-benchmark* to generate load on the server. *Redis-benchmark* is configured to run a series of 10 tests with 50 parallel connections, each with a maximum of 200 000 queries. The QoS performance threshold is set to 30 000 queries per second (QPS). The metric analyzer uses `Hiredis`, a C client library for *Redis*, to access run-time statistics from the server.



(a) *Sys-trace* (b) *Call-trace* (c) *Branch-profile*

Figure 6.3: Gzip: QoS performance metric over time

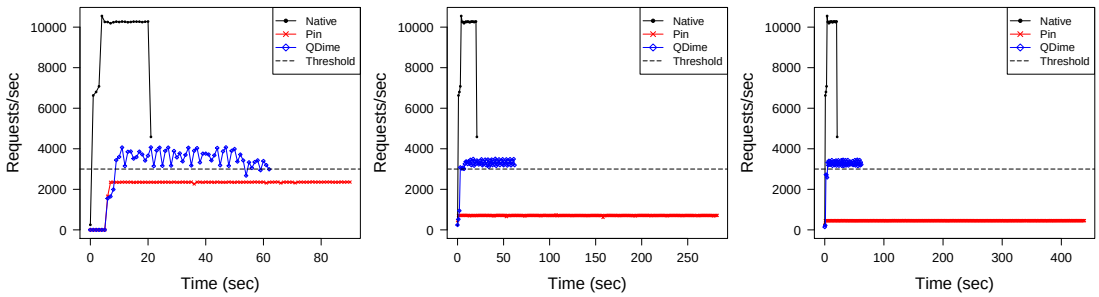


(a) *Sys-trace* (b) *Call-trace* (c) *Branch-profile*

Figure 6.4: MySQL: QoS performance metric over time

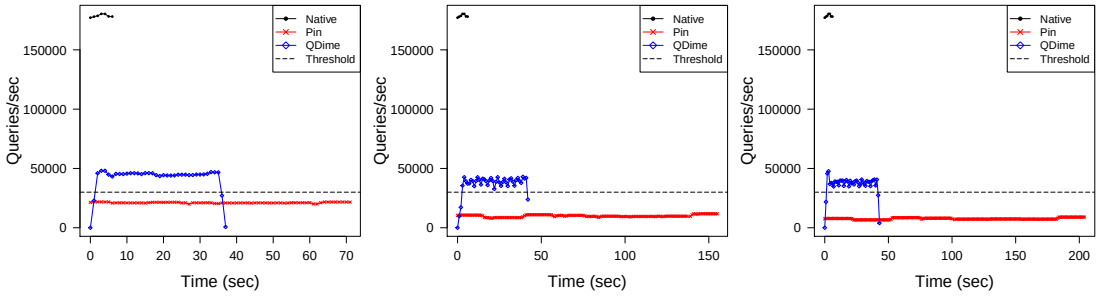
6.6.3 Experimental Results

QDIME respects the QoS performance threshold of the application, while PIN drastically alters the QoS of the application. Figures 6.3, 6.4, 6.5, and 6.6 show the QoS metrics over time for *Gzip*, *MySQL*, *Apache*, and *Redis*, respectively, for each analysis tool. The execution time in seconds is on the x-axis, while the y-axis represents the QoS metric. The higher the QoS values, the better it is for the overall performance of the application. We repeated each experiment ten times natively, with PIN, and with QDIME. For increased readability, the mentioned figures show only the first experimental repetition for each application and tool. Moreover, the box-plots in Figures 6.7a, 6.7b, 6.7c, and 6.7d summarize the QoS values for the ten repetitions and, if applicable, for all QDIME runs. The y-axis represents the QoS metric values during native execution, instrumentation on top of PIN,



(a) *Sys-trace* (b) *Call-trace* (c) *Branch-profile*

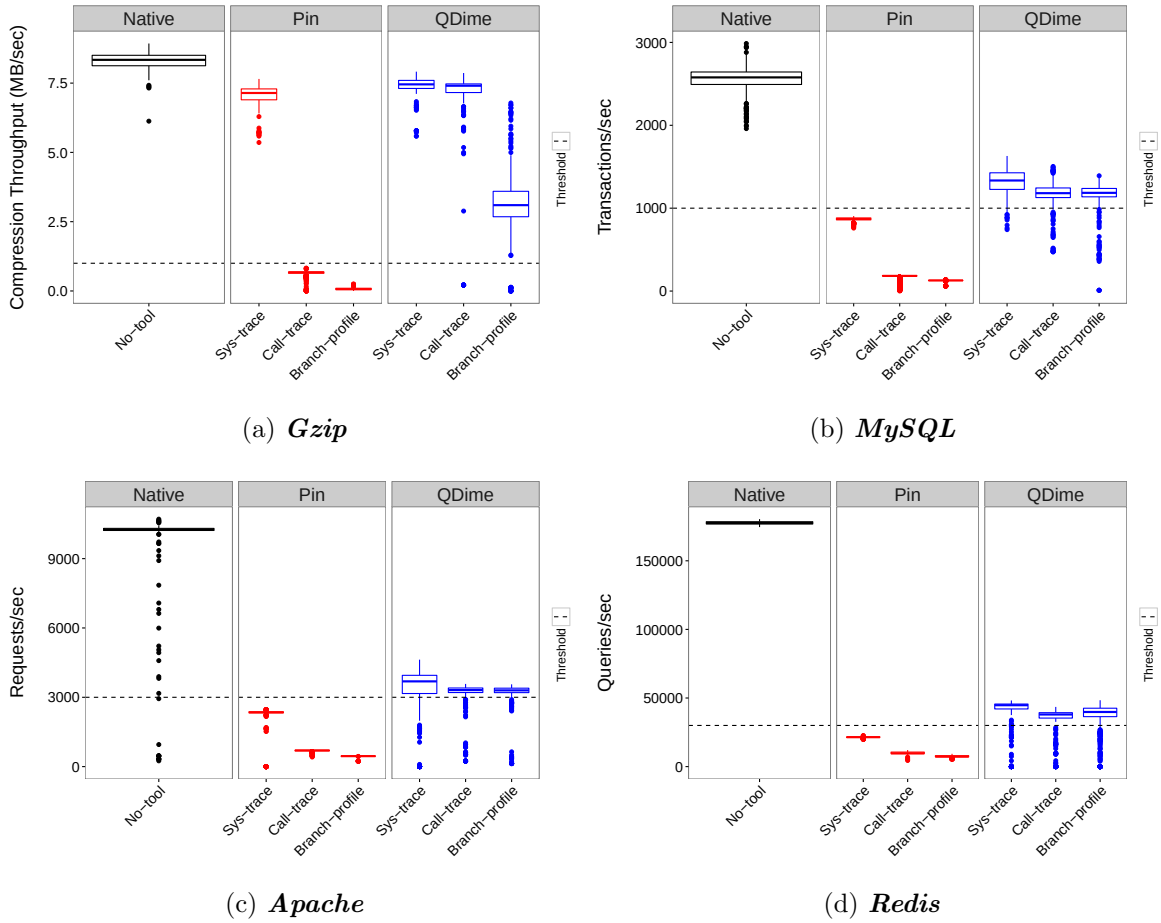
Figure 6.5: Apache: QoS performance metric over time



(a) *Sys-trace* (b) *Call-trace* (c) *Branch-profile*

Figure 6.6: Redis: QoS performance metric over time

and instrumentation on top of QDIME.



(a) ***Gzip*** (b) ***MySQL***

(c) ***Apache*** (d) ***Redis***

Figure 6.7: Summary of QoS-metric values

***Gzip*:** While PIN maintains a high average compression rate of 7 MB/sec with the lightweight *Sys-trace*, it degrades the average QoS down to 646.7 KB/sec with *Call-trace*, and 73.6 KB/sec with *Branch-profile*. However, as seen in Figures 6.3 and 6.7a, QDIME always respects the threshold of 1 MB/sec with an average compression rate of 7.3 MB/sec with *Sys-trace*, 5.9 MB/sec with *Call-trace*, and 2.8 MB/sec with *Branch-profile*. Natively, *Gzip* has an average compression rate of 8.2 MB/sec.

***MySQL*:** The uninstrumented execution of *MySQL* server maintains an average of 2544 TPS as shown in Figures 6.4 and 6.7b. PIN produces only an average of 868, 161,

and 128 TPS for *Sys-trace*, *Call-trace*, and *Branch-profile*, respectively. On the other side, QDIME respects the threshold of 1000 TPS with an average of 1310, 1165, and 1158 TPS respectively for the three analysis tools.

***Apache*:** Figures 6.5 and 6.7c show that the native execution of the *Apache* server along with its workload has an average of 8302 RPS. PIN cannot respect the server's QoS and produces only an average of 2343 RPS with *Sys-trace*, 689 RPS with *Call-trace*, and 452 RPS with *Branch-profile*. QDIME maintains an average QoS of 3514 RPS with *Sys-trace*, 3009 RPS with *Call-trace*, and 3014 RPS with *Branch-profile*.

***Redis*:** Natively, *Redis* maintains a QoS of about 177 667 QPS on average as in Figures 6.6 and 6.7d. The QoS remains below threshold with PIN and drops from 21 448 QPS on average with *Sys-trace* to 9859 QPS with *Call-trace* and 7461 QPS with *Branch-profile*. Contrarily to PIN, QDIME meets the defined threshold of 30 000 QPS with all the analysis tools by maintaining an average QoS of 41 811 QPS with *Sys-trace*, 34 918 QPS with *Call-trace*, and 36 440 QPS with *Branch-profile*.

As for the slowdown factors of the instrumented applications, QDIME always outperforms PIN. On average, QDIME reduces the runtime overhead by 1.41× with *Sys-trace*, 5.67× with *Call-trace*, and 10.26× with *Branch-profile*. Figure 6.8 presents the average slowdown factors of the instrumented applications on top of PIN and QDIME w.r.t. the application's native execution time. The x-axis lists the analysis tools, whereas the y-axis shows the average slowdown factors. QDIME reduces the slowdown of PIN with *Sys-trace* from 2.96×, 4.14×, and 8.47× to 1.96×, 2.81×, and 4.97× for *MySQL*, *Apache*, and *Redis*, respectively. Similarly, *Call-trace*'s overhead drops from 12.20×, 15.07×, 13.09×, and 18.56× with PIN to 1.19×, 2.20×, 2.87×, and 5.74× with QDIME respectively for *Gzip*, *MySQL*, *Apache*, and *Redis*. With *Branch-profile*, PIN slows down *Gzip* by a factor of 102.03×. Thanks to the adaptive budget function and redundancy suppression of QDIME, this slowdown is reduced to 2.56×. Similarly, the slowdown factor of *Apache* with *Branch-profile* on top of QDIME is only 2.84× compared to 19.96× atop of PIN.

Although QDIME with *Branch-profile* requires multiple runs to achieve high coverage with *MySQL* and *Redis* (Figure 6.9b), the total execution times of QDIME runs remains lower than that of PIN. PIN introduces slowdown factors of 20.09× and 24.56× for *MySQL* and *Redis*, respectively. QDIME reduces these values to 2.22× and 5.63× for one run. The instrumentation of *MySQL* atop of PIN consumes 785 sec on average, whereas the combined execution times for the two runs with QDIME is 175 sec at most. Similarly, *Redis* runs three times on top of QDIME for a total of 147 sec, while PIN takes 209 sec on average.

QDIME, not only, respects the QoS thresholds and reduces the runtime overhead, but also provides high unique instrumentation coverage in a low number of runs. QDIME, on

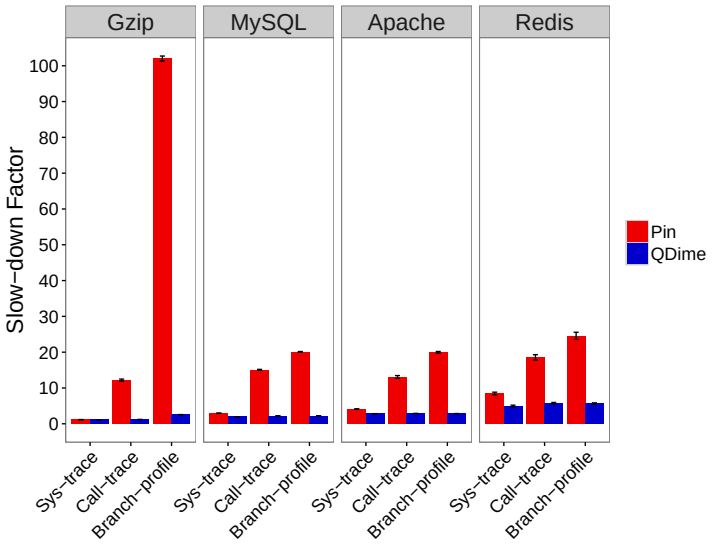
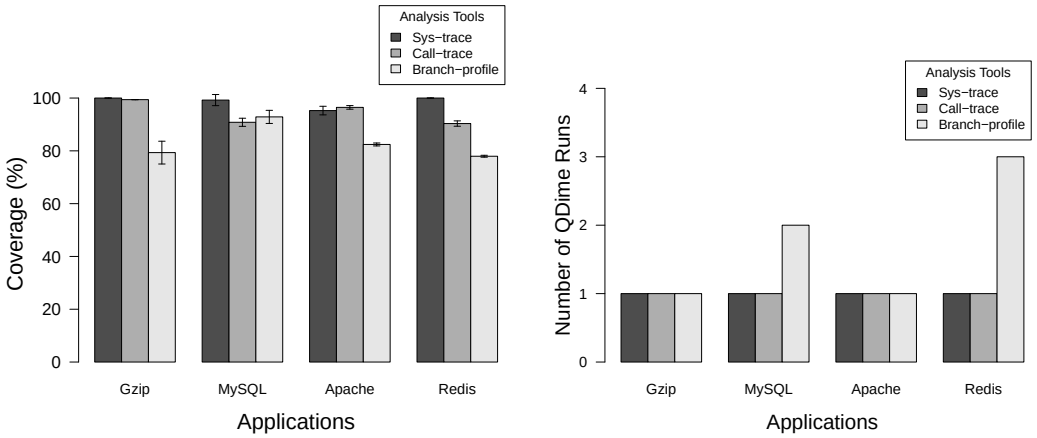


Figure 6.8: Slowdown factors of the instrumented applications.

average, conveys 92% of the unique runtime information compared to Pin. Figure 6.9a plots the average QDime coverage for each analysis tool. The x-axis lists the applications, while the y-axis shows the instrumentation coverage of QDime w.r.t. that of Pin. In general, QDime maintains high coverage by being able to extract up to 100% coverage in a single run with some applications and tools. In our experiments, the lowest coverage of QDime is with *Redis*, as only 78% of the *Branch-profile* information is extracted. Figure 6.9b highlights the number of runs required by QDime to reach the coverage of Figure 6.9a. Only *Branch-profile*, the heaviest analysis tool, consumed two runs for *MySQL* and three runs for *Redis* to extract 93% and 78%, respectively.

Table 6.1 summarizes the experimental results. As shown earlier, QDime always respects the QoS threshold of the instrumented application leading to a higher overall system performance as compared to Pin. Even with highly intrusive analysis tools like *Branch-profile*, QDime provides the application under instrumentation with a guaranteed QoS. Also, QDime reduces the application's slowdown factors introduced by DBI while extracting high instrumentation coverage. Using QDime, the slowdown factors of the instrumented applications dropped by 1.41× with *Sys-trace*, 5.67× with *Call-trace*, and 10.26× with *Branch-profile* on average w.r.t. Pin. The unique runtime information collected by QDime represents an average of 92% of that of Pin, with a minimum of 78% and a maximum of 100%.



(a) Unique Instrumentation Coverage (b) Number of Runs

Figure 6.9: QDIME coverage

Threshold Values. Another set of experiments evaluate the performance of QDIME with various threshold values to identify the highest possible in our settings. Increasing the threshold value limits the instrumentation budget of QDIME, and accordingly, results in a lower coverage. Table 6.2 lists the maximum possible thresholds along with the respective coverage values for the four benchmark applications. Beyond these values, QDIME starts to violate the thresholds. With higher thresholds, QDIME extracts from 70% to 100% of the unique coverage for all the applications. The only exception is *MySQL* with *Branch-profile* where the highest threshold of 1200 TPS restricted the coverage to 55.5%. Such a trade-off between the QoS level and the extracted coverage is expected especially for a heavy-weight analysis tool.

Table 6.1: Summary of QDIME experimental results

Applications		***Gzip***			***MySQL***			***Apache***			***Redis***		
Analysis Tools		**Sys**	**Call**	**Branch**	**Sys**	**Call**	**Branch**	**Sys**	**Call**	**Branch**	**Sys**	**Call**	**Branch**
QoS: QDIME > PIN**?**		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Threshold		**1 MB/sec**			**1000 TPS**			**3000 RPS**			**30000 QPS**		
Threshold	QDIME	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Satisfied?	PIN	✓	×	×	×	×	×	×	×	×	×	×	×
Slowdown	QDIME	**1.13×**	**1.19×**	**2.56×**	**1.96×**	**2.20×**	**2.22×**	**2.81×**	**2.87×**	**2.84×**	**4.97×**	**5.74×**	**5.63×**
	PIN	**1.17×**	**12.20×**	**102×**	**2.96×**	**15.07×**	**20.09×**	**4.13×**	**13.09×**	**19.96×**	**8.47×**	**18.56×**	**24.57×**
# Runs	QDIME	**1**	**1**	**1**	**1**	**1**	**2**	**1**	**1**	**1**	**1**	**1**	**3**
Coverage	QDIME	**100%**	**99.40%**	**79.34%**	**99.23%**	**90.82%**	**92.86%**	**95.26%**	**96.46%**	**82.42%**	**100%**	**90.34%**	**77.96%**

Table 6.2: Maximum QDIME threshold values with respective unique coverage

Applications	***Gzip***			***MySQL***			***Apache***			***Redis***		
Analysis Tools	**Sys**	**Call**	**Branch**	**Sys**	**Call**	**Branch**	**Sys**	**Call**	**Branch**	**Sys**	**Call**	**Branch**
Max. Threshold	**7.5**	**7.5**	**7 MB/sec**	**1500**	**1200**	**1200 TPS**	**4000**	**3400**	**3400 RPS**	**48000**	**44000**	**44000 QPS**
Coverage	**78.5%**	**69.7%**	**81.1%**	**94.5%**	**92.5%**	**55.5%**	**93.1%**	**95.9%**	**82.3%**	**100%**	**89.2%**	**71.5%**

6.7 Summary

This chapter discusses the concept of QDIME, a QoS-aware DBI technique that guarantees a certain QoS to the application under analysis. QDIME accepts a QoS metric with a designated threshold and constraints to be satisfied for an acceptable overall performance. QDIME periodically collects performance data from the instrumented program and decides the instrumentation budget accordingly. The evaluation on four popular real-world applications shows the practicality, scalability, and effectiveness of QDIME. QDIME respects the user-defined QoS threshold while maintaining an average unique coverage of 92%. Also, QDIME reduces the slow-down factors of the instrumented applications. These results make QDIME a useful tool for instrumenting QoS-based applications and enable the design of dynamic analysis tools with QoS guarantees. This is the first work that considers QoS specifications during program instrumentation.

Chapter 7

Conclusion and Future Work

Program analysis using instrumentation is widely used for understanding program behavior and identifying performance bottlenecks. Instrumentation naturally introduces perturbation to the program under analysis. Such perturbation can alter the timing behavior of the program. Real-time systems must respect extra-functional constraints especially the timing properties. Thus, real-time systems require specialized program instrumentation techniques. Time-aware instrumentation preserves a program's logical correctness and respects its timing constraints. Current approaches for time-aware instrumentation rely on static and source-code instrumentation techniques. They require the availability of the source code of the program including all libraries dependencies. Moreover, a WCET analysis is performed before and after instrumentation. Static time-aware instrumentation is sound and effective for hard real-time systems, but impractical for soft real-time systems due to its restrict assumptions.

In this thesis, we introduce the theory, method, and tools for time-aware dynamic binary instrumentation technique realized in DIME tool. DIME bounds the runtime overhead of the instrumentation process to respect the program's timing constraints. Chapter 3 discusses the design of DIME along with its three implementations of DIME. These implementations differ in their budget checking overhead, strictness of respecting budget, and overshoots beyond the budget. The evaluation of DIME shows an average reduction in overhead by 12, 7, and 3 folds compared to native Pin. Two cases studies of DIME instrumenting a media player and a control application demonstrate the scalability and applicability of DIME.

DIME aims, not only to limit the instrumentation overhead, but also to maximize the instrumentation coverage. Instrumentation frameworks, in general, may extract an amount

of runtime information that contains many redundancies. In Chapter 4, we propose a redundancy suppression technique to increase the unique instrumentation coverage. DIME avoids collecting redundant information and prohibits the instrumentation of previously instrumented code regions. DIME was able to extract 97% of the call context tree of the VLC video player while playing a high-definition video. DIME was also used for the branch profiling of the PostgreSQL database management system and was able to extract 97% of the unique instrumentation information in three runs.

Moreover, Chapter 5 examines the relation between the operation parameters of DIME and its performance. A set of experiments is conducted to evaluate the impact of the time period and the instrumentation budget on the runtime overhead and the instrumentation coverage of DIME.

Chapter 6 presents the method for QoS-aware dynamic instrumentation technique and its tool. QDIME satisfies user-defined performance thresholds and constraints to maintain an acceptable QoS level to the program under analysis. The evaluation of QDIME revealed its practicality and scalability to instrument complex applications.

The proposed tools and approaches, in this thesis, can be further extended to increase the effectiveness of time-aware dynamic instrumentation. The following is some of the potential improvements and future work in this area:

1. *Further investigation on increasing the unique instrumentation coverage:* the test results of the unique coverage in Chapter 5 are inconclusive. Thus, further experiments are required to consider additional input factors such as the program structure.

2. *Adapting redundancy suppression to different definitions of unique coverage:* Using the redundancy suppression feature, DIME prohibits the extraction of repeated runtime information. Some program-comprehension techniques may benefit from the suppression of *consecutive* repetitions for example.

3. *Reconstruction of program execution using the extracted information by DIME:* An interesting research objective is the reconstruction of program execution path using the partial traces for the goal of program comprehension or bug detection.

4. *Combining program sampling algorithms with DIME:* It is possible to use the framework of DIME to implement sampling techniques for time-sensitive systems. At a periodic rate, DIME may allow a sampling algorithm to collect information only if the budget is not consumed. For example, using adaptive bursty sampling can increase the ability of DIME to instrument cold code. This research direction can allow the program-sampling work to be transferred safely to the world of real-time systems.

References

[1] ab: Apache HTTP Server Benchmarking Tool. http://httpd.apache.org/docs/2.2/programs/ab.html.

[2] Apache HTTP Server. http://httpd.apache.org/.

[3] GZIP Compression Utility. http://www.gnu.org/software/gzip/.

[4] Intel Developer Zone. http://software.intel.com/.

[5] Iozone filesystem benchmark. http://www.iozone.org/.

[6] MySQL Database Management System. http://www.mysql.com/.

[7] Pgbench: Benchmarking Tool for PostgreSQL. http://wiki.postgresql.org/wiki/Pgbench.

[8] PostgreSQL Global Development Group. http://www.postgresql.org/.

[9] QNX. http://www.qnx.com/.

[10] Quanser. http://www.quanser.com/.

[11] Redis Data Structure Server. http://redis.io/.

[12] Stress-ng benchmark. http://kernel.ubuntu.com/ cking/stress-ng/.

[13] VLC Media Player. http://www.videolan.org/vlc/index.html.

[14] Luca Abeni and Giorgio Buttazzo. Integrating Multimedia Applications in Hard Real-Time Systems. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 4–13, Dec 1998.

[15] Luca Abeni and Giorgio Buttazzo. Resource Reservation in Dynamic Real-Time Systems. *Real-Time Systems*, 27(2):123–167, 2004.

[16] Jennifer M Anderson, Lance M Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R Henzinger, Shun-Tak A Leung, Richard L Sites, Mark T Vandevoorde, Carl A Waldspurger, and William E Weihl. Continuous Profiling: Where Have All The Cycles Gone? *ACM Transactions on Computer Systems (TOCS)*, 15(4):357–390, 1997.

[17] Pansy Arafa, Hany Kashif, and Sebastian Fischmeister. DIME: Time-aware Dynamic Binary Instrumentation Using Rate-based Resource Allocation. In *Proc. of the 13th International Conference on Embedded Software (EMSOFT)*, Montreal, Canada, Sept 2013.

[18] Pansy Arafa, Daniel Solomon, Samaneh Navabpour, and Sebastian Fischmeister. Debugging Behaviour of Embedded-Software Developers: An Exploratory Study. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Raleigh, USA, 2017.

[19] Pansy Arafa, Guy M. Tchamgoue, Hany Kashif, and Sebastian Fischmeister. QDIME: QoS-aware Dynamic Binary Instrumentation. In *IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Banff, Canada, 2017.

[20] Matthew Arnold and Barbara G. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *Proc. of the ACM SIGPLAN Conf. on Programming language design and implementation (PLDI)*, 2001.

[21] Borzoo Bonakdarpour, Samaneh Navabpour, and Sebastian Fischmeister. Sampling-based Runtime Verification. In *Proc. of the 17th Intl. Conf. on Formal Methods (FM)*, Jun. 2011.

[22] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, 2003.

[23] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent Dynamic Instrumentation. *SIGPLAN Not.*, 47(7), Mar. 2012.

[24] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.*, 14(4), Nov. 2000.

[25] Prashanth P. Bungale and Chi-Keung Luk. PinOS: A Programmable Framework for Whole-system Dynamic Instrumentation. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 137–147, New York, NY, USA, 2007. ACM.

[26] M. Burrows, Ú Erlingsson, S.-T. A. Leung, M. T. Vandevoorde, C. A. Waldspurger, K. Walker, and W. E. Weihl. Efficient and Flexible Value Sampling. *SIGPLAN Not.*, 35(11):160–167, November 2000.

[27] Bryan M Cantrill, Michael W Shapiro, and Adam H Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association, 2004.

[28] Chabbi, Milind and Liu, Xu and Mellor-Crummey, John. Call paths for pin tools. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 76:76–76:86, New York, NY, USA, 2014. ACM.

[29] Albert M. K. Cheng. *Real-Time Systems: Scheduling, Analysis, and Verification*. Wiley-Interscience, 2002.

[30] Bas Cornelissen and Leon Moonen. Visualizing Similarities in Execution Traces. In *Proceedings of the 3rd Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, pages 6–10, 2007.

[31] Bas Cornelissen and Leon Moonen. *On Large Execution Traces and Trace Abstraction Techniques*. Delft University of Technology, Software Engineering Research Group, 2012.

[32] Joachim Denil, Hany Kashif, Pansy Arafa, Hans Vangheluwe, and Sebastian Fischmeister. Instrumentation and Preservation of Extra-functional Properties of Simulink Model. In *Proceedings of the Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, Alexandria, USA, 2015.

[33] Tevi Devor. Pin Tutorial. CGO'12, San Jose, California, 2012.

[34] G. Dobson, R. Lock, and I. Sommerville. QoSOnt: a QoS Ontology for Service-centric Systems. In *Software Engineering and Advanced Applications, 2005. 31st EUROMICRO Conference on*, pages 80–87, Aug 2005.

[35] Andrew Edwards, Hoi Vo, and Amitabh Srivastava. Vulcan: Binary Transformation in a Distributed Environment. Technical report, 2001.

[36] Facebook. Mysql tech talk 11.2.10, Feb 2010. http://livestre.am/rIpq.

[37] S. Fischmeister and P. Lam. Time-Aware Instrumentation of Embedded Software. *IEEE Transactions on Industrial Informatics*, 2010.

[38] Sebastian Fischmeister and Yanmeng Ba. Sampling-based Program Execution Monitoring. *SIGPLAN Not.*, 45(4), 2010.

[39] Sebastian Fischmeister and Patrick Lam. On Time-Aware Instrumentation of Programs. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 305–314. IEEE, 2009.

[40] Nathan Froyd, John Mellor-Crummey, and Rob Fowler. Low-overhead Call Path Profiling of Unmodified, Optimized Code. In *Proc. of the 19th Annual Intl. Conf. on Supercomputing (ICS)*, 2005.

[41] M.P. Gallaher and B.M. Kropp. The Economic Impacts of Inadequate Infrastructure for Software Testing. National Institute of Standards & Technologg Planning Report, 2002.

[42] Anjana Gosain and Ganga Sharma. A Survey of Dynamic Program Analysis Techniques and Tools. In *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*, pages 113–122. Springer, 2015.

[43] Abdelwahab Hamou-Lhadj and Timothy C Lethbridge. Measuring Various Properties of Execution Traces to Help Build Better Trace Analysis Tools. In *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, pages 559–568. IEEE, 2005.

[44] Ahmed Hassan, Daryl Martin, Parminder Flora, Paul Mansfield, and Dave Dietz. An Industrial Case Study of Customizing Operational Profiles Using Log Compression. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 713–723. IEEE, 2008.

[45] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead Memory Leak Detection Using Adaptive Statistical Profiling. *SIGOPS Oper. Syst. Rev.*, 38(5), Oct. 2004.

[46] John L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer*, 33(7), 2000.

[47] Martin Hirzel and Trishul Chilimbi. Bursty Tracing: A Framework for Low-Overhead Temporal Profiling. In *In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*. ACM, 2001.

[48] J.K. Hollingsworth, O. Niam, B.P. Miller, Zhichen Xu, M.J.R. Goncalves, and Ling Zheng. Mdl: a language and compiler for dynamic program instrumentation. In *Parallel Architectures and Compilation Techniques., 1997. Proceedings., 1997 International Conference on*.

[49] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *Proc. of the 3rd Conf. on USENIX Windows NT Symp. (WINSYM)*, 1999.

[50] Bart Jacob, Paul Larson, B Leitao, and SAMM da Silva. SystemTap: Instrumenting the Linux Kernel for Analyzing Performance and Functional Problems. *IBM Redbook*, 2008.

[51] Kevin Jeffay and David Bennett. A Rate-based Execution Abstraction for Multimedia Computing. pages 64–75, 1995.

[52] Kevin Jeffay and Steve Goddard. A Theory of Rate-based Execution. In *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No.99CB37054)*, pages 304–314, 1999.

[53] Kevin Jeffay and Steve Goddard. Rate-based Resource Allocation Models for Embedded Systems. In *Proc. of the 1st Intl. Workshop on Embedded Software*, 2001.

[54] Yogi Joshi, Guy M. Tchamgoue, and Sebastian Fischmeister. Runtime Verification of LTL on Lossy Traces. In *32nd ACM Symposium on Applied Computing (SAC)*, pages 1379–1386, Marrakech, Morocco, 2017.

[55] Hany Kashif, Pansy Arafa, and Sebastian Fischmeister. INSTEP: A Static Instrumentation Framework for Preserving Extra-functional Properties. In *Proc. of the 19th IEEE Intl. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Aug. 2013.

[56] Hany Kashif and Sebastian Fischmeister. Program Transformation for Time-aware Instrumentation. In *Proc. of the 17th IEEE Intl. Conf. on Emerging Technologies & Factory Automation (ETFA)*, Sep. 2012.

[57] Hany Kashif, Johnson Thomas, Hiren Patel, and Sebastian Fischmeister. Static Slack-based Instrumentation of Programs. In *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–8, Sept 2015.

[58] Baris Kasikci, Thomas Ball, George Candea, John Erickson, and Madanlal Musuvathi. Efficient Tracing of Cold Code via Bias-Free Sampling. In *USENIX Annual Technical Conference*, pages 243–254, 2014.

[59] Irvin R. Katz and John R. Anderson. Debugging: An Analysis of Bug-Location Strategies. *Hum.-Comput. Interact.*, 3, 1987.

[60] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A Run-Time Assurance Approach for Java Programs. *Form. Methods Syst. Des.*, 24(2), 2004.

[61] A Ko and B Myers. A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems. *Journal of Visual Languages & Computing*, 16(1-2), 2005.

[62] Alexey Kopytov. SysBench Manual. http://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf.

[63] Naveen Kumar, Bruce R. Childers, and Mary Lou Soffa. Low Overhead Program Monitoring and Profiling. In *Proc. of the 6th ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE)*, 2005.

[64] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. *SIGPLAN Not.*, 30, 1995.

[65] J.R. Larus. Efficient Program Tracing. *Computer*, 26(5), 1993.

[66] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug Isolation via Remote Program Sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 141–154, New York, NY, USA, 2003. ACM.

[67] Daniel Lohmann, Wolfgang Schroder-Preikschat, and Olaf Spinczyk. Functional and Non-functional Properties in a Family of Embedded Operating Systems. In *Proceedings of the 10th IEEE International Workshop On Object-Oriented Real-Time Dependable Systems (WORDS)*. IEEE Computer Society, 2005.

[68] Gregory Lueck, Harish Patil, and Cristiano Pereira. PinADX: An Interface for Customizable Debugging with Dynamic Instrumentation. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 114–123, New York, NY, USA, 2012. ACM.

[69] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2005.

[70] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: Effective Sampling for Lightweight Data-Race Detection. *SIGPLAN Not.*, 44(6), 2009.

[71] J. M. Mellor-Crummey and T. J. LeBlanc. A Software Instruction Counter. In *Proc. of the 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1989.

[72] Ying Meng, Ok-Kyoon Ha, and Yong-Kee Jun. Dynamic Instrumentation for Nested Fork-join Parallelism in OpenMP Programs. In *Future Generation Information Technology*, pages 154–158. Springer, 2012.

[73] Douglas C Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, 2008.

[74] Linda J. Moore and Angelica R. Moya. Non-Intrusive Debug Technique for Embedded Programming. In *Proc. of the 14th Intl. Symp. on Software Reliability Engineering (ISSRE)*, 2003.

[75] Peter Mork. Techniques for Debugging Parallel Programs. Technical report, University of Miskolc.

[76] T. Moseley, A. Shye, V.J. Reddi, D. Grunwald, and R. Peri. Shadow Profiling: Hiding Instrumentation Costs with Parallelism. In *Intl. Symp. on Code Generation and Optimization(CGO)*, Mar. 2007.

[77] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter Sweeney. We Have It Easy, But Do We Have It Right? *IEEE Intl. Symp. on Parallel and Distributed Processing*, 2008.

[78] Mayur Naik, Hongseok Yang, Ghila Castelnuovo, and Mooly Sagiv. Abstractions from Tests. *SIGPLAN Not.*, 47(1), Jan. 2012.

[79] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *SIGPLAN Not.*, 42(6), Jun. 2007.

[80] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, April 2013.

[81] Augusto Oliveira, Jean-Christophe Petkovich, Thomas Reidemeister, and Sebastian Fischmeister. DataMill: Rigorous Performance Evaluation Made Easy. In *Proc. of the 4th ACM/SPEC International Conference on Performance Engineering*. ACM, April 2013.

[82] William Omre. Debug and Trace for Multicore SoCs. Technical report, ARM, 2008.

[83] David A Patterson. *Computer Architecture: a Quantitative Approach*. Elsevier, 2011.

[84] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Proc. of the USENIX Windows NT Workshop*, 1997.

[85] Amitabha Roy, Steven Hand, and Tim Harris. Hybrid Binary Rewriting for Memory Access Instrumentation. In *Proc. of the 7th ACM SIGPLAN/SIGOPS Intl. Conf. on Virtual Execution Environments (VEE)*, 2011.

[86] Arkaitz Ruiz-Alvarez and Kim Hazelwood. Evaluating the Impact of Dynamic Binary Translation Systems on Hardware Cache Performance. In *IEEE Intl. Symp. on Workload Characterization (IISWC)*, 2008.

[87] B. Sabata, S. Chatterjee, M. Davis, J.J. Sydir, and T.F. Lawrence. Taxonomy for QoS Specifications. In *Object-Oriented Real-Time Dependable Systems, 1997. Proceedings., Third International Workshop on*, pages 100–107, Feb 1997.

[88] Mauricio Serrano and Xiaotong Zhuang. Building Approximate Calling Context from Partial Call Traces. In *Proc. of the Intl. Symp. on Code Gen. and Optimization*, 2009.

[89] Beth Simon, Dennis Bouvier, Tzu-Yi Chen, Gary Lewandowski, Robert McCartney, and Kate Sanders. Common Sense Computing (Episode 4): Debugging. *Computer Science Education*, 18(2), 2008.

[90] J. Sincero, W. Schroder-Preikschat, and O. Spinczyk. Approaching Non-functional Properties of Software Product Lines: Learning from Products. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, 2010.

[91] Marco Spuri and Giorgio C. Buttazzo. Efficient Aperiodic Service under Earliest Deadline Scheduling. In *Real-Time Systems Symposium, 1994., Proceedings.*, pages 2–11, Dec 1994.

[92] Amitabh Srivastava and Alan Eustace. ATOM: A System for Building Customized Program Analysis Tools. *SIGPLAN Not.*, 39, 1994.

[93] Ariel Tamches and Barton P. Miller. Fine-grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Proc. of the 3rd Symp. on Operating Systems Design and Implementation (OSDI)*, 1999.

[94] G. M. Tchamgoue and S. Fischmeister. Lessons Learned on Assumptions and Scalability with Time-aware Instrumentation. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–7, Oct 2016.

[95] Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, and Ravi Ayyagari. Analyzing Dynamic Binary Instrumentation Overhead. 2007.

[96] Dan Upton, Kim Hazelwood, Robert Cohn, and Greg Lueck. Improving Instrumentation Speed via Buffering. In *Proc. of the Workshop on Binary Instrumentation and Applications (WBIA)*, 2009.

[97] Steven Wallace and Kim Hazelwood. SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance. In *Intl. Symp. on Code Generation and Optimization (CGO)*, Mar. 2007.

[98] Chadd C Williams and Jeffrey K Hollingsworth. Interactive Binary Instrumentation. In *Second International Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*. Citeseer, 2004.

[99] Zhibin Yu, Weifu Zhang, and Xuping Tu. MT-Profiler: A Parallel Dynamic Analysis Framework Based on Two-stage Sampling. In *Proc. of the 9th Intl. Conf. on Advanced Parallel Processing Technologies (APPT)*, 2011.

[100] Junyuan Zeng, Yangchun Fu, and Zhiqiang 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, New York, NY, USA, 2015. ACM.

[101] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. System Support for Automatic Profiling and Optimization. *SIGOPS Oper. Syst. Rev.*, 31, 1997.

[102] Qin Zhao, Ioana Cutcutache, and Weng-Fai Wong. PiPA: Pipelined Profiling and Analysis on Multicore Systems. *ACM Trans. Archit. Code Optim.*, 7(3), Dec. 2010.

APPENDICES

Appendix A

Parameter Tuning Experiments: Detailed Results

This appendix lists the analysis of variance (ANOVA) tables of the experiments discussed in Chapter 5. An ANOVA tests the equality of the means of the data populations associated to different levels of the input factors. The probability of the equality of the means is called the P-value. The first column of an ANOVA table lists the sources of variation including the residuals. The residuals denote the variability in the response variable that is unexplained by the input factors. Column 2 shows the degrees of freedom (DF) of each factor. Note that the summation of the degrees of freedom of all sources of variation should equal to the number of input data points minus one [73]. The third and the fourth columns calculate the sum of squares (SS) and the mean square (MS) respectively. The sum of squares represents the total variation in the response that can be attributed to the associated input factor. The mean square (MS) is the division result of the sum of squares (SS) by the associated degrees of freedom (DF). The F-ratio column, which is the fifth, shows the ratio of the mean square to the residual mean square. The P-value, in column 6, is calculated based on the F-ratio and the degrees of freedom (DF). The last column of the ANOVA table concludes the significance of the the associated input factor by comparing the P-value to the significance level. A P-value that is lower than the significance level of 0.01 confirms the significance of the effect of the input factor.

The number of input data points to each ANOVA test equals to the product of (1) the number of benchmark programs/inputs, (1) the number of replications, and (2) the number of levels' combinations of T and B. As mentioned before in Section 5.2, the experiments run ten replications for each benchmark set such that each replication includes the four combinations of T and B levels. Also, the number of programs and inputs per each

benchmark set is mentioned in Section 5.2. To sum up, SPEC's ANOVA tests have 1320 input data points (recall that some of the 12 SPEC programs have multiple inputs). The experiments of IOzone and Stress-ng include 13 and 10 benchmark programs respectively. Thus, the number of the input data points to IOzone's ANOVA tests is 520, whereas there are 400 input data points to Stress-ng's ANOVA tests.

Tables A.1, A.2, and A.3 are dedicated for the slow-down factors of SPEC, IOzone, and Stress-ng programs in order. The second three tables A.4, A.5, and A.6 show the results of the unique instrumentation coverage of the three benchmark sets, while the last three tables A.7, A.8, and A.9 present these of the raw coverage of the same benchmarks.

Slow-down Factors

Table A.1: ANOVA table: slow-down factors of SPEC benchmark

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F Ratio	P Value	Significant?
Time Period (T)	1	70	70.1	18.782	0.0000158	Yes
Budget (B)	1	2076	2075.7	556.243	$< 2 \times 10^{-16}$	Yes
Benchmark Program (P)	11	9796	890.6	238.654	$< 2 \times 10^{-16}$	Yes
Interaction Factor (T:B)	1	3	2.8	0.753	0.386	No
Residuals	1305	4870	3.7			

Table A.2: ANOVA table: slow-down factors of IOzone benchmark

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F Ratio	P Value	Significant?
Time Period (T)	1	0.00360	0.003605	36.48	2.99×10^{-10}	Yes
Budget (B)	1	0.00013	0.000133	1.35	0.24580	No
Benchmark Program (P)	12	0.07682	0.006401	64.79	$< 2 \times 10^{-16}$	Yes
Interaction Factor (T:B)	1	0.00132	0.001323	13.39	0.00028	Yes
Residuals	504	0.04979	0.000099			

Table A.3: ANOVA table: slow-down factors of Stress-ng benchmark

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F Ratio	P Value	Significant?
Time Period (T)	1	0.0	0.05	13.33	0.000298	Yes
Budget (B)	1	0.1	0.07	19.14	0.0000157	Yes
Benchmark Program (P)	9	769.5	85.50	23051.36	$< 2 \times 10^{-16}$	Yes
Interaction Factor (T:B)	1	0.0	0.05	12.52	0.000452	Yes
Residuals	387	1.4	0.00			

Unique Instrumentation Coverage

Table A.4: ANOVA table: unique coverage of SPEC benchmark

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F Ratio	P Value	Significant?
Time Period (T)	1	7897	7897	122.7	$< 2 \times 10^{-16}$	Yes
Budget (B)	1	147001	147001	2283.5	$< 2 \times 10^{-16}$	Yes
Benchmark Program (P)	11	101159	9196	142.9	$< 2 \times 10^{-16}$	Yes
Interaction Factor (T:B)	1	7466	7466	116.0	$< 2 \times 10^{-16}$	Yes
Residuals	1305	84010	64			

Table A.5: ANOVA table: unique coverage of IOzone benchmark

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F Ratio	P Value	Significant?
Time Period (T)	1	1247	1247	134.833	$< 2 \times 10^{-16}$	Yes
Budget (B)	1	89266	89266	9649.446	$< 2 \times 10^{-16}$	Yes
Benchmark Program (P)	12	325	27	2.923	0.000617	Yes
Interaction Factor (T:B)	1	87863	87863	9497.821	$< 2 \times 10^{-16}$	Yes
Residuals	504	4662	9			

Table A.6: ANOVA table: unique coverage of Stress-ng benchmark

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F Ratio	P Value	Significant?
Time Period (T)	1	2344	2344	313.08	$< 2 \times 10^{-16}$	Yes
Budget (B)	1	231142	231142	30871.88	$< 2 \times 10^{-16}$	Yes
Benchmark Program (P)	9	3737	415	55.46	$< 2 \times 10^{-16}$	Yes
Interaction Factor (T:B)	1	2974	2974	397.25	$< 2 \times 10^{-16}$	Yes
Residuals	387	2898	7			

Raw Instrumentation Coverage

Table A.7: ANOVA table: raw coverage of SPEC benchmark

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F Ratio	P Value	Significant?
Time Period (T)	1	0.008	0.008	0.420	0.517	No
Budget (B)	1	21.810	21.810	1107.781	$< 2 \times 10^{-16}$	Yes
Benchmark Program (P)	11	31.448	2.859	145.212	$< 2 \times 10^{-16}$	Yes
Interaction Factor (T:B)	1	0.004	0.004	0.209	0.647	No
Residuals	1305	25.693	0.020			

Table A.8: ANOVA table: raw coverage of IOzone benchmark

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F Ratio	P Value	Significant?
Time Period (T)	1	15.54	15.54	1052.36	$< 2 \times 10^{-16}$	Yes
Budget (B)	1	72.14	72.14	4884.66	$< 2 \times 10^{-16}$	Yes
Benchmark Program (P)	12	3.28	0.27	18.53	$< 2 \times 10^{-16}$	Yes
Interaction Factor (T:B)	1	11.77	11.77	796.89	$< 2 \times 10^{-16}$	Yes
Residuals	504	7.44	0.01			

Table A.9: ANOVA table: raw coverage of Stress-ng benchmark

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F Ratio	P Value	Significant?
Time Period (T)	1	2089	2089	22.93	2.4×10^{-6}	Yes
Budget (B)	1	5559	5559	61.02	5.36×10^{-14}	Yes
Benchmark Program (P)	9	29638	3293	36.15	$< 2 \times 10^{-16}$	Yes
Interaction Factor (T:B)	1	1401	1401	15.38	0.000104	Yes
Residuals	387	35256	91			