QDIME: QoS-aware Dynamic Binary Instrumentation

Pansy Arafa, Guy Martin Tchamgoue, Hany Kashif, and Sebastian Fischmeister
Dept. of Electrical and Computer Engineering
University of Waterloo, Canada
{parafa, gmtchamg, hkashif, sfischme}@uwaterloo.ca

***Abstract*—Software systems with quality of service (QoS), such as database management systems and web servers, are ubiquitous. Such systems must meet strict performance requirements. Instrumentation is a useful technique for the analysis and debugging of QoS systems. Dynamic binary instrumentation (DBI) extracts runtime information to comprehend system's behavior and detect performance bottlenecks. However, existing DBI tools are intrusive; adding unacceptable delay to the program execution. Such delay alters the performance requirements and degrades the overall quality and the user experience of the system. Moreover, the delay may change the system behavior, thus, producing misleading run-time information.**

This paper presents QDIME, a QoS-aware dynamic binary instrumentation technique that respects system's performance requirements. QDIME takes a user-defined QoS threshold as an input and periodically gathers QoS feedback from the system under analysis to decide its instrumentation budget.

We implemented QDIME on top of PIN, a popular DBI framework. We evaluated QDIME with *Gzip*, *MySQL* server, *Apache* HTTP server, and *Redis*. The experiments show that QDIME respects the user-defined QoS threshold and, thus, improves the performance of the monitored application by manifolds. QDIME is able to provide up to 100% instrumentation coverage with an average of 92% when compared to PIN. Moreover, QDIME reduces the slow-down factor of the instrumented application by 1.41, 5.67, and 10.26 folds for *Sys-trace*, *Call-trace*, and *Branch-profile* respectively. A release of QDIME is available for download at https://github.com/pansy-arafa/qdime.

I. Introduction

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 need 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 [14]. These requirements include the total volume of computation, end-to-end delay, error rate, response time, and jitter [35]. For example, the production database management system at Facebook should be able to handle roughly 60 million queries per second [16], and its key-value store, billions of requests per second [32]. In this paper, 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.

Program analysis tools are critical for understanding the run-time behavior of programs. Developers need such tools to analyze programs and identify performance bottlenecks [37]. Instrumentation is the process of inserting analysis code inside the program to extract runtime information. Static instrumentation preprocesses the program to insert analysis code into the program before execution [37], [23]. On the other side, dynamic binary instrumentation (DBI) weaves analysis code into the program's binary during execution [29], [31], [12]. Although DBI incurs high runtime overhead, it does not require program preprocessing and can handle dynamically loaded libraries and dynamically generated code.

DBI frameworks have been widely adopted in software analysis and security applications [29], [12], [31]. However, these tools are well-known for being intrusive; imposing high overhead to the monitored program. For instance, our experiments showed that extracting the branch profile of *Gzip* with Pin incurs a slowdown of about 102×. The perturbation added by the instrumentation may alter the overall QoS of a system drastically degrading the end-user's experience. However, many applications are not tolerant to more than a few percent of latency degradation as this hurts the QoS and causes distortions that thwart meaningful analysis [13].

In this paper, we present Qdime, a QoS-aware DBI technique that guarantees a certain QoS to the program under analysis. For this purpose, 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. Instrumentation *budget* is the amount of time, per period, during which instrumentation is enabled. Periodically, Qdime determines its instrumentation *budget* as a function of the QoS metric. The function is constructed such 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 P, the budget, B, is computed to satisfy $0 \leq B \leq P$.

To accurately determine the budget, the values of the QoS metric are periodically extracted from the instru-

mented 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 easily achievable with applications such as *MySQL*, *Gzip*, *Apache* server, and *Redis*. QDIME also has the option to log small pieces of data about the information it extracts. These data are used and updated across multiple runs of the same application to guarantee the unicity of the extracted information.

We implemented QDIME on top of PIN instrumentation framework [29]. The evaluation on four popular real-world applications shows the performance, practicality, scalability, and effectiveness of QDIME. Moreover, QDIME not only guarantees a QoS to the monitored application but also reduces the runtime overhead by manyfold and provides up to 100% of coverage w.r.t. PIN. In summary, QDIME offers the following features:

- *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.
- *Asymptotic Completeness*: The re-execution and redundancy suppression feature of QDIME allows it to extract only new information through multiple runs, and also to extract full traces with limited runs.
- *Flexibility*: The QoS metrics can be of any type and from any source including system-level metrics such as CPU and memory consumption. Also, the thresholds, period and constraints are entirely defined by the developer, who can then tune them to adjust the trade-off between information gain, QoS, and overhead.
- *Practicality*: producing a high 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*: The generic nature of QDIME makes it possible to implement the technique on top of any existing instrumentation framework. Further, QDIME does not depend on any system or hardware level features.

The remainder of this paper is organized as follows: Section II provides an overview of DBI techniques. Section III describes QDIME and presents its design architecture. Section IV provides some implementation details of QDIME. In Section V, we evaluate QDIME on some commonly used applications. Section VI discusses some of the limitations of QDIME. Finally, Sections VII and VIII list the related work and the conclusion respectively.

II. BACKGROUND

In this section, we provide a brief description of DBI techniques with emphasis on PIN framework upon which we built a prototype of QDIME.

A. Dynamic Binary Instrumentation

Software instrumentation is essential for designing program analysis tools such as profilers and debuggers. The instrumentation acts by augmenting the original program with user-generated extra code for analysis. Binary instrumentation can be done either *statically* or *dynamically*. Static binary instrumentation (SBI) tools such as EEL [27], ATOM [37], or PEBIL [28], use static analysis techniques to modify a binary before execution. This behavior limits the scope of SBI techniques to the only code they can statically access as they lack a global view of the program.

DBI, contrarily to SBI, requires no preprocessing of a binary and possesses a complete view of the program at runtime including shared libraries, plug-ins, and dynamically generated code. Accordingly, the DBI approach is more practical, flexible, and scalable. DBI frameworks intercept a program and inject the instrumentation code at the designated points of interest during execution. Based on the mechanism of this task, DBI frameworks can be classified into two groups. In the first group are tools like DYNINST [10] and VULCAN [15] that use code transformation, and thus, suffer from transparency issues. The second group includes tools like PIN [29], DYNAMORIO [12], and VALGRIND [31] that use a Just-in-Time (JIT) compiler [11] to dynamically recompile the binary and run it from a software code cache. DBI frameworks usually propose a set of custom application programming interfaces (APIs) to help developers provide the (1) *analysis routine*, which determines the collection and the processing of the runtime data, and the (2) *instrumentation routine*, which specifies where to inject calls to the analysis routine.

Runtime overhead constitutes the principal drawback of DBI frameworks and mainly originates from both the instrumentation itself and the user-defined analysis routine. However, the instrumentation process, which involves injecting the user code into the binary, uses different optimization techniques [11], [22] and represents only a minor source of overhead [29]. Uh *et al.* [38] showed that a typical DBI uses only about 1.4% of its time to generate and optimize code dynamically. On the other hand, DBI spends about 4.6% of the time to process the collected data and 88% to execute the dynamically generated code. Thus, the main contributor to the runtime overhead is the user-defined analysis code. Contrarily to the instrumentation overhead that occurs only when a new code is discovered, the analysis routine runs every time an instrumentation point is encountered. The complexity and invocation frequency of the analysis function directly impact the overhead and thus the QoS. Unfortunately, while much has been done to lower the instrumentation overhead, less attention has so far been paid to that of the user-defined code [13], [41]. Moreover, none has been done to support programs with strict QoS requirements. We focus on this type of systems in this work.

B. Instrumentation with Pin

Pin [29] is a state-of-the-art cross-platform DBI framework. Pin is portable and supports any commodity hardware with the IA-32 or the x86-64 instruction-set architecture. The framework transparently allows an instrumented program to maintain the same instruction and data addresses as well as the same register and memory values with its uninstrumented counterpart. Pin provides a rich set of APIs that allow developers to build various analysis tools called *Pintools*.

Pin takes full control of a program by injecting itself into its binary, thus, forcing the program to run inside its virtual machine. It then uses its JIT compiler to translate, instrument, and execute the program. The unit of compilation is a *trace*, which represents 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 its execution, Pin compiles the first trace and generates a modified trace. The generated trace is almost identical to the original but enables Pin to regain control when a branch exits the trace. Before moving to the next trace, the current trace is saved into a cache code to speed-up further executions of the same sequence of code. Whenever the JIT compiler fetches a new code, the pintool is allowed to instrument the code before compilation. The instrumentation can run at different levels of granularity, e.g., *image*, *trace*, *routine*, and *instruction* level, each indicating when Pin executes the instrumentation routine.

The Trace Version APIs [5] of Pin provide a way to implement and switch between multiple types of instrumentation on traces. For example, a trace may have both heavyweight and lightweight instrumentation and Pin can switch between the two at runtime based on a dynamic test defined by the user. From an implementation point of view, trace versions are user-defined numbers with zero being the default for all traces. Switching to a new version causes the execution to continue at a new trace starting with the current instruction. To illustrate, assume that the current trace A consists of 10 instructions and has the default trace version zero. Then, after instrumenting five instructions, the dynamic test results in switching the trace version to one. Pin will split the trace A into two traces A and B such that trace A ends at the fifth instruction and trace B starts at the sixth instruction. Pin will, then, set the trace version of B to one and continue the execution accordingly.

When compared to other DBI frameworks, Pin is flexible, easier to use, and has relatively lower overhead. Using a lightweight basic block counter, Luk *et al.* [29] reported a slowdown of 2.5×, 5.1×, and 8.3×, respectively for Pin [29], DynamoRio [12], and Valgrind [31], with the SPECint benchmark on a Linux platform. For the above-mentioned reasons, we choose to implement Qdime on top of Pin to support the dynamic binary instrumentation of software with strict QoS requirements. However, Qdime is a generic approach, not restricted to Pin, and can be ported to other instrumentation frameworks as well.

```
1  void analysis(...){
2     start = get_time();
3     //Analysis code
4     Budget -= get_time() - start;
5  }
6  void instrumentation(...){
7    if(redundancy_off)
8       is_new = check_log();
9    if(is_new || !(redundancy_off)){
10      version = insert_call(check_budget())
11      if(version == instrument){
12         //enable instrumentation
13         //insert analysis routines
14      }
15      else{
16         //disable instrumentation
17      }
18      if(redundancy_off)
19         update_log();
20   }
21 }
22 void budget_handler(int sig){
23    Budget = compute_Budget(qos);
24 }
```

Listing 1: Instrumenting with Qdime

III. QoS-Aware Instrumentation

This section provides details about Qdime, its algorithm and design architecture.

A. Overview

It is important for instrumentation tools to guarantee a certain QoS to the monitored program for responsiveness and usability. Qdime is a DBI tool designed for this purpose. To instrument a program, the user first defines the QoS metric to be monitored by Qdime at run-time. Further, the user is required to set a *threshold* and the constraints that are to be satisfied by the metric to guarantee an acceptable QoS. Finally, the user defines the instrumentation period for Qdime.

Listing 1 shows the core algorithm of Qdime. At run-time, Qdime decides to either enable instrumentation (i.e., augment the program with analysis code) or disable instrumentation (i.e., prevent the injection of the analysis code into the program). Thus, to meet the QoS requirements, Qdime trades off information, QoS, and overhead. Using the trace-version APIs, Qdime checks its *budget* at each instrumentation point as shown in Line 10. Thus, the instrumented code executes as long as Qdime has the budget to do so and as a consequence, the QoS of the program degrades. The analysis routine at Line 1 consumes the budget at each call by decrementing its value by the total time it takes to execute. 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

mechanism of budget checking and version switching in QDIME follows the same model of [8].

QDIME uses a signal handler that fires periodically to replenish the budget (Listing 1, Line 22). The function `compute_budget()` generates a new budget w.r.t. 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. Figure 1 shows how the budget is periodically replenished during the extraction of system calls from *Apache*. For instance around 10 sec, when *Apache* processes about 4,000 requests/sec, which is above the threshold of 2,000 requests/sec, the budget is set to approximately 35% of the 1 sec instrumentation period. However, around 55 sec, the QoS drops to 3,050 requests/sec forcing QDIME to adjust its budget to only about 20% to meet the threshold.

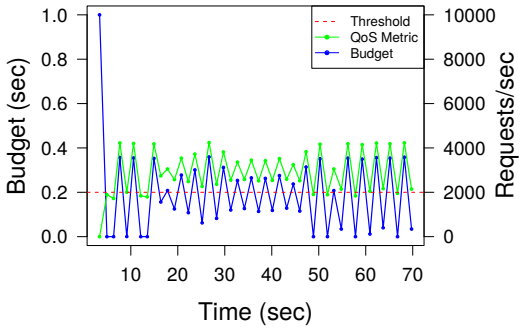


Fig. 1: *Apache*: budget vs. QoS metric

B. Redundancy Suppression

Instrumentation tools may generate an amount of runtime information that, however, contain many redundant entries. The extracted information can form several gigabytes of data. Many researchers have reported the complexity of understanding software systems using runtime information due to its sheer size and complexity [18], [19]. One reason is the redundancy of traces which occurs due to, for example, repetition of sequence of events. In [18], 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 extracted information is the same regardless of the number of times the instruction is executed or instrumented. Examples of such tools are code-coverage analyzers and memory profilers.

To respect the QoS threshold, QDIME may disable instrumentation, and hence, extract partial traces compared to PIN. We enrich QDIME with a *Redundancy Suppression* feature allowing it to prohibit the instrumentation of a previously instrumented code [7]. This feature provides three advantages; (1) reduction of the number of extracted traces to facilitate program comprehension, (2) increasing instrumentation coverage, and (3) decreasing runtime overhead. Although useful, the redundancy suppression is an *optional* feature, which the user can turn off when not needed.

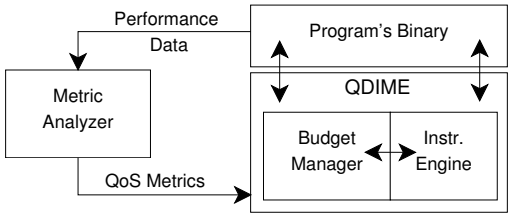


Fig. 2: Architecture of QDIME

With the redundancy suppression feature enabled, QDIME utilizes the instrumentation budget to extract unique, i.e., non-redundant run-time information. To be able to identify the instrumented pieces of code, QDIME saves their relative addresses into a simple log file as shown in Listing 1, Line 19. QDIME checks the log before instrumenting a new trace to ensure the uniqueness of the extracted information (Listing 1, Line 8). Multiple runs of QDIME may be required to, optimally, achieve full coverage. In this case, the redundancy log file is passed across runs, allowing QDIME to reveal only new information during each run. We refer to this extraction process as *asymptotic completeness*.

C. 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.

The architecture of QDIME is as shown in Figure 2. The Metric Analyzer periodically accesses 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 P, although this is not a requirement. As shown in Listing 1, Line 23, the Budget Manager of QDIME periodically reads the QoS values

from the shared memory to compute the budget. This architecture allows QDIME to handle any application out-of-the-box, without any modification of the binary.

IV. Implementation

This section describes QDIME's implementation details.

A. Basic Details

We implemented a prototype of QDIME on top of PIN as a C/C++ library, that any Pintool can load. QDIME relies on the Trace Version APIs [5] provided by PIN to transparently switch instrumentation on and off with minimal overhead. The switching decision is made whenever QDIME reaches an instrumentation point. A release of QDIME is available for download at https://github.com/pansy-arafa/qdime.

For the redundancy suppression feature, QDIME utilizes a hash-table as the redundancy log. After instrumenting a trace, QDIME saves the trace relative address to the log. QDIME searches the log before the instrumentation of a trace. If τ_i is the current trace and τ_j is the trace already in the hash table log L, QDIME instruments τ_i only if $address(\tau_i) \neq address(\tau_j), \forall \tau_j \in L$. To avoid adding to the runtime overhead, both steps, saving to and searching the log, take place in the instrumentation routine (not the analysis routine). According to comparative evaluations of the hash-table versus other data structures such as binary search tree (BST), a hash-table provides the following:

- Negligible overhead and simple implementation.
- Fast logging and searching (average case: constant; worst case: linear in the size of the table).
- Negligible false negatives. False negatives will occur, if QDIME prohibits instrumentation of an uninstrumented trace.
- Possible false positives, i.e., allowing instrumentation of a previously instrumented trace. False positives will happen, if the current trace τ_i was previously instrumented as part of a previous trace τ_j. Note that a trace may be part of another trace due to the creation of new traces through version switching as explained earlier in Section II-B.

The evaluation of the redundancy-suppression data-structure showed that there is a trade-off between the ratios of false positives and false negatives. A high ratio of false negatives may dramatically decrease the instrumentation coverage, i.e., the amount of extracted information. Although the hash-table log may generate a percentage of redundant information, it is safer and preferable due to the low ratio of false negatives. An extended discussion of the mentioned evaluation is available in [7].

QDIME registers a `SIGVTALRM` signal, which fires at each instrumentation period P to replenish the budget (Listing 1, Line 22). We implemented the function `get_time()` of Listing 1, Line 2 as an inlined assembly function using `rdtsc` to return the value of the timestamp counter.

B. Budget Function

We implemented the budget function `compute_Budget()` of Listing 1, Line 23, as a map of $\mathbb{R}^+$ into $[0, P]$, where P 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.

In this paper, we design the budget function as a proportional controller which periodically determines the percentage of the QoS to safely use for instrumentation. If the value of the QoS, $v(t)$, extracted from the program at time t is above the threshold, there will be a QoS gain given by $g(t) = v(t) - T$, where T represents the user-defined threshold value. The gain function $g(t)$ depicts the health of the monitored program w.r.t. the defined threshold and represents the extra portion of the QoS to utilize for instrumentation without violating the threshold. The function of Equation 1 encodes these observations and sets the budget to zero whenever the QoS constraint is violated, i.e., QoS below the threshold in this case. Otherwise, it always produces a budget $B < P$. Instead of zero, the user could also choose a default value to use when QDIME fails to meet the threshold condition. Since respecting the QoS threshold is the primary objective, QDIME adopts a conservative budget function by choosing the tuning factor of $1/(v(t) + T)$ in Equation 1. As later shown in Section V-C, the budget function of Equation 1 enables QDIME to successfully avoid threshold violations while providing both high coverage and reduced overhead. However, one could opt for a more aggressive tuning parameter, considering the overhead of the analysis routine, such as $1/(M - T)$ if the QoS upper-bound value M is known or even $1/v(t)$.

$$b(t) = \begin{cases} 0 & \text{if } v(t) < T \\ \dfrac{v(t) - T}{v(t) + T} \times P & \text{otherwise} \end{cases} \quad (1)$$

V. 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 V-C are geometric means.

A. 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 [33]. 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 least 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.
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 [36].
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. 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.
- *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 threshold to provide a guaranteed QoS to the end-user, (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 coverage.

B. Benchmark Applications

This subsection describes the benchmarks and the experimentation objectives. We fixed the instrumentation period to $P = 1$ sec for all the applications. Different values of the instrumentation period and the QoS threshold will be discussed in Section VI.

***Gzip* Compression Utility** *Gzip* is a widely used data-compression application of about 59,000 lines of code (LOC) adopted by the GNU project [3]. Our experiments aim to instrument *Gzip* 1.4 while maintaining a compression rate threshold of 1 MB/s. *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 LOC [4]. For the experiments, we installed *MySQL* 5.5.43 and SysBench 0.4.12, a benchmarking tool for databases [25]. 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 1,000 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 1,785,428 LOC 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].

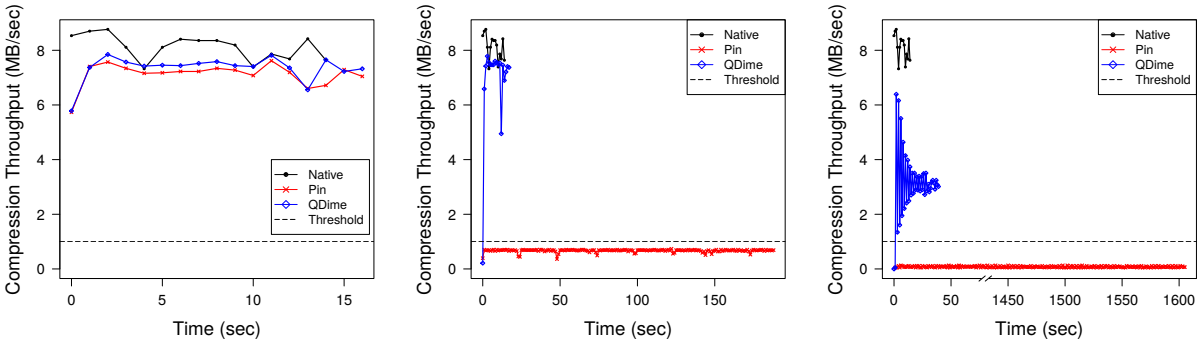
Although QDIME is capable of handling an application's child processes, the *Apache* child processes refused the attachment of both PIN and QDIME. Therefore, the experiments run *Apache* in debug mode, which forces the server to run as a single process. To respect a QoS threshold of 3,000 requests per 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 [6]. *Redis* consists of roughly 20,000 LOC. Many well-known projects such as Twitter, GitHub, and Pinterest, rely on *Redis*.

We installed *Redis* server 3.0.3 and used *redis-benchmark* utility to generate load on the server. We configured *redis-benchmark* 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.

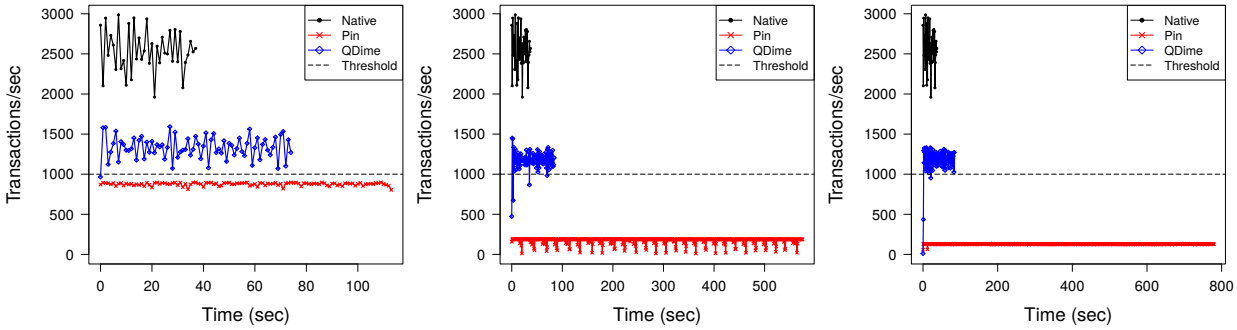
C. Experimental Results

QDIME respects the QoS performance threshold of the application, while PIN drastically alters the QoS of the application. Figures 3, 4, 5, and 6 show the QoS metrics over time for *Gzip*, *MySQL*, *Apache*, and *Redis*, respectively,



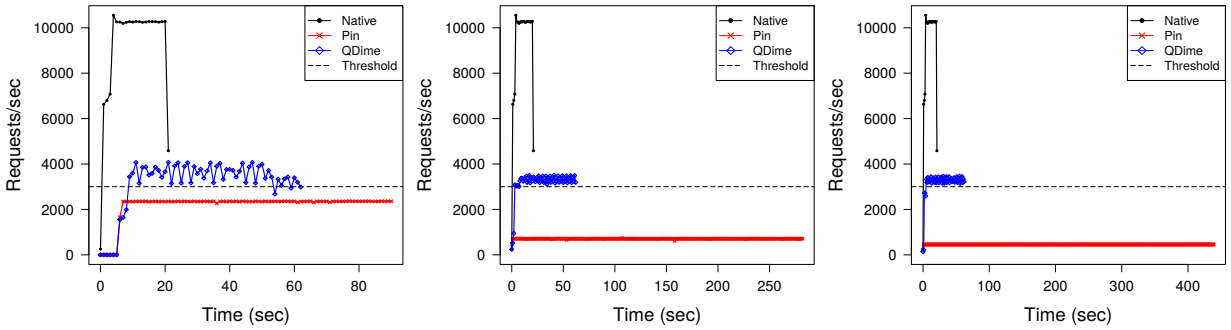
(a) *Sys-trace* (b) *Call-trace* (c) *Branch-profile*

Fig. 3: ***Gzip***: QoS performance metric over time



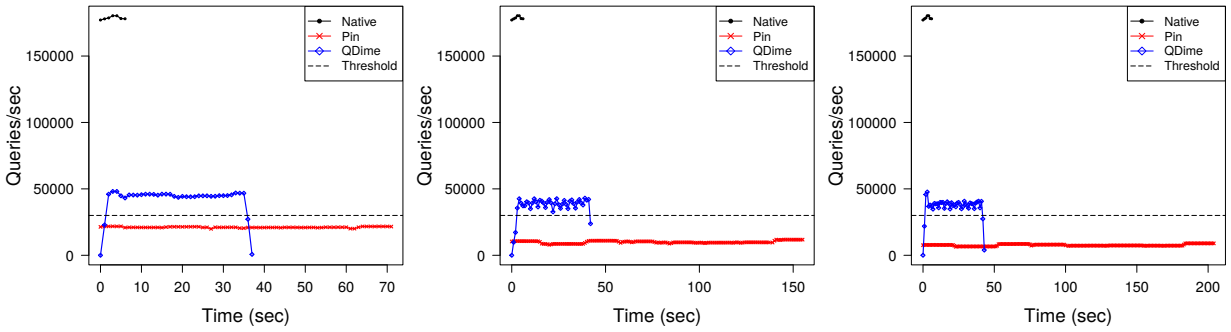
(a) *Sys-trace* (b) *Call-trace* (c) *Branch-profile*

Fig. 4: ***MySQL***: QoS performance metric over time



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

Fig. 5: ***Apache***: QoS performance metric over time



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

Fig. 6: ***Redis***: QoS performance metric over time

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. In this settings, we turned on the redundancy suppression feature of QDIME. For increased readability, the mentioned figures show only the first experimental repetition for each application and tool. Moreover, the box-plots in Figures 7, 8, 9, and 10 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, and instrumentation on top of QDIME. ***Gzip*:** While PIN maintains a high average compression rate of 7.00 MB/sec with the lightweight *Sys-trace*, it degrades the average QoS down to 646.72 KB/sec with *Call-trace*, and 73.60 KB/sec with *Branch-profile*. However, as seen in Figures 3 and 7, QDIME always respects the threshold of 1 MB/sec with an average compression rate of 7.33 MB/sec with *Sys-trace*, 5.91 MB/sec with *Call-trace*, and 2.89 MB/sec with *Branch-profile*. Natively, *Gzip* has an average compression rate of 8.26 MB/sec.

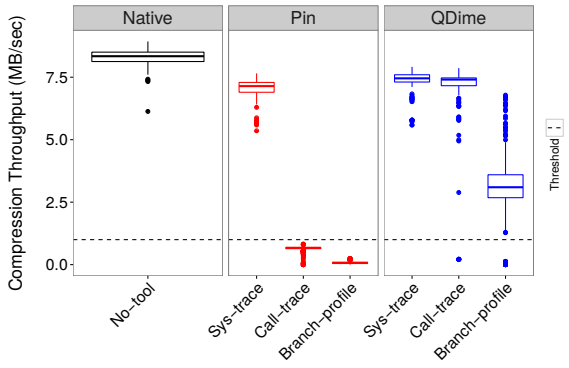


Fig. 7: Summary of ***Gzip*** QoS-metric values

***MySQL*:** The un-instrumented execution of *MySQL* server maintains an average of 2,544 TPS as shown in Figures 4 and 8. 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 1,000 TPS with an average of 1,310, 1,165, and 1,158 TPS respectively for the three analysis tools.

***Apache*:** Figures 5 and 9 show that the native execution of the *Apache* server along with its workload has an average of 8,302 RPS. PIN cannot respect the server's QoS and produces only an average of 2,343 RPS with *Sys-trace*, 689 RPS with *Call-trace*, and 452 RPS with *Branch-profile*. QDIME maintains an average QoS of 3,514 RPS with *Sys-trace*, 3,009 RPS with *Call-trace*, and 3,014 RPS with *Branch-profile*.

***Redis*:** Natively, *Redis* maintains a QoS of about 177,667 QPS on average as in Figures 6 and 10. The QoS remains below threshold with PIN and drops from 21,448 QPS on average with *Sys-trace* to 9,859 QPS with *Call-trace* and 7,461 QPS with *Branch-profile*. Contrarily to PIN, QDIME meets the defined threshold of 30,000 QPS

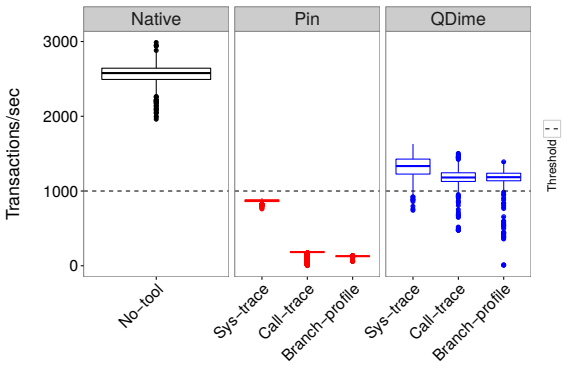


Fig. 8: Summary of ***MySQL*** QoS–metric values

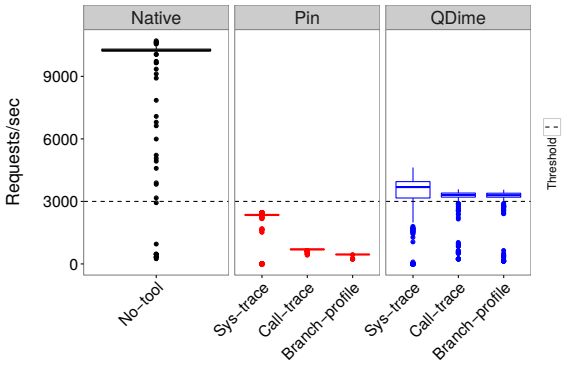


Fig. 9: Summary of ***Apache*** QoS-metric values

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*.

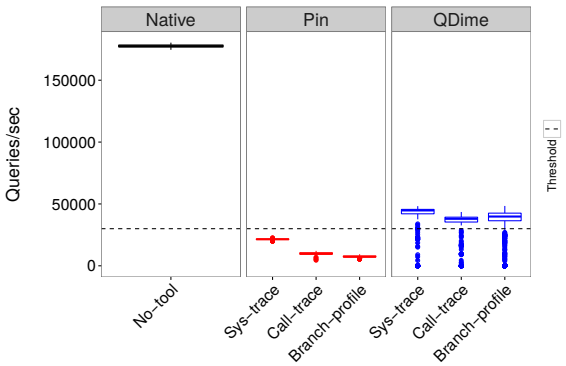


Fig. 10: Summary of ***Redis*** QoS-metric values

As for the slowdown factor 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 11 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 13), 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.

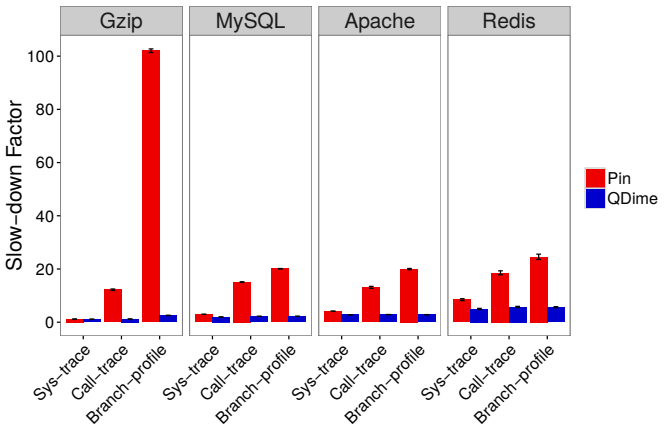


Fig. 11: Slowdown factors of the instrumented applications

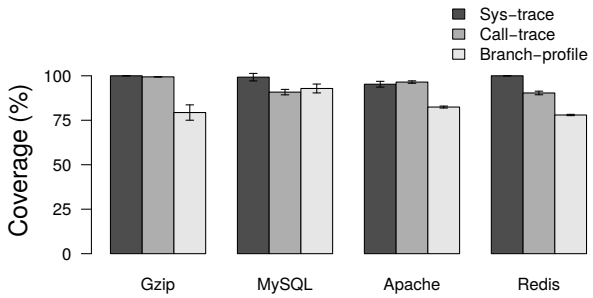


Fig. 12: Instrumentation coverage of QDime

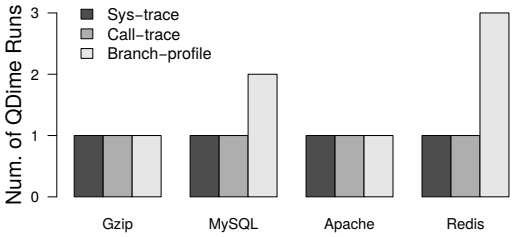


Fig. 13: Number of QDime runs

QDime is a useful and practical DBI technique since it, not only, respects the QoS thresholds and reduces the runtime overhead, but also provides high instrumentation coverage in a low number of runs. QDime, on average, conveys 92% of the runtime information compared to Pin. Figure 12 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 13 highlights the number of runs required by QDime to reach the coverage of Figure 12. Only *Branch-profile*, the heaviest analysis tool, consumed two runs for *MySQL* and three runs for *Redis* to extract 93% and 78%, respectively.

Summary: Table I 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. 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. Finally, the runtime information collected by QDime represents an average of 92% of that of Pin, with a minimum of 78% and a maximum of 100%.

VI. Discussion

QDime is a generic method that is portable to DBI frameworks other than Pin. QDime can instrument any QoS application whose security permits the attachment of a DBI framework. For example, the child processes of the *Google Chrome* browser executes within a restrictive environment and accordingly prohibits the Pin attachment. Also, the applicability of QDime is not limited to the analysis tools used in this paper.

Redundancy suppression, which is an optional feature of QDime, is useful for (1) analysis tools that extract useless repetitions, and (2) applications that maintain negligible variance among runs. Without redundancy suppression, QDime may require a higher number of runs to collect sufficient instrumentation coverage. For instance, without redundancy suppression, QDime extracted 69.88% of call traces with *Gzip* in one run (vs. 99.4% with redundancy suppression enabled).

We evaluated various threshold values to identify the highest that QDime can support in our settings. Table II presents the maximum possible thresholds along with respective coverage. Beyond these values, QDime starts to violate the thresholds. With higher thresholds, QDime extracts from 70% to 100% of the coverage for all the applications. The only exception is *Branch-profile* and *MySQL* where the highest threshold of 1,200 TPS restricted the coverage to 55.5%. However, such a trade-off between the QoS level and the extracted coverage is expected especially for a heavy-weight analysis tool.

TABLE I: Summary table

Applications		*Gzip*			*MySQL*			*Apache*			*Redis*		
Analysis Tools		**Sys**	**Call**	**Branch**	**Sys**	**Call**	**Branch**	**Sys**	**Call**	**Branch**	**Sys**	**Call**	**Branch**
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×	102.03×	2.96×	15.07×	20.09×	4.13×	13.09×	19.96×	8.47×	18.56×	24.57×
QDIME	# Runs	1	1	1	1	1	2	1	1	1	1	1	3
	Coverage	**100%**	**99.40%**	**79.34%**	**99.23%**	**90.82%**	**92.86%**	**95.26%**	**96.46%**	**82.42%**	**100%**	**90.34%**	**77.96%**

TABLE II: Max. threshold values with resp. coverage

Application		**Sys-trace**	**Call-trace**	**Branch**
Gzip	**Thr.**	7.50 MB/sec	7.50 MB/sec	7.00 MB/sec
	Cov.	78.57%	69.70%	81.14%
MySQL	**Thr.**	1,500 TPS	1,200 TPS	1,200 TPS
	Cov.	94.59%	92.57%	55.56%
Apache	**Thr.**	4,000 RPS	3,400 RPS	3,400 RPS
	Cov.	93.10%	95.94%	82.30%
Redis	**Thr.**	48,000 QPS	44,000 QPS	44,000 QPS
	Cov.	100%	89.22%	71.55%

Finally, we assessed the impact of the instrumentation-period value on QDIME's performance by varying the instrumentation period from 1 to 10 seconds. Table III summarizes the results for *Call-trace*. In general, smaller periods allow fine-grained decision making and are therefore to be favored at the expense of larger ones. The experiments show that increasing the instrumentation period increases the time at which QDIME decides its budget leading to frequent threshold violations. With *Gzip*, QDIME always respects the threshold when the instrumentation period $P < 7$ sec. In this case, there is a drop in coverage with threshold violations when $P > 6$ sec. For *MySQL* and *Apache*, the threshold violations start when $P > 3$ sec and for *Redis*, when $P > 2$ sec but without coverage sacrifice.

TABLE III: Coverage and threshold violations points

Threshold		***Gzip***	***MySQL***	***Apache***	***Redis***
Respected	**Per.**	**1 – 6 sec**	**1 – 3 sec**	**1 – 3 sec**	**1 – 2 sec**
	Cov.	99.40%	94.75%	96.06%	90.62%
Violated	**Per.**	**7 – 10 sec**	**4 – 10 sec**	**4 – 10 sec**	**3 – 10 sec**
	Cov.	70%	94.75%	96.06%	90.62%

VII. Related Work

This section then compares the existing DBI techniques to QDIME. Thanks to their flexibility and scalability, the recent years have seen a wide adoption and development of many DBI frameworks such as DynamoRio [12], Dyninst [10], Valgrind [31], Vulcan [15], Pin [29], and Pemu [40]. These tools all suffer from the same overhead problem, sacrificing the overall QoS of the monitored program. Thus, many works [30], [41] leverage parallelism to reduce the overhead of DBI frameworks. However, while the output of these frameworks highly depends on the number of cores available and the amount of code that can effectively be parallelized, QDIME can still benefit from these techniques to provide even better throughput on multicore platforms. Several other optimization techniques [26], [39], [42] have been proposed with the same objective, reducing the instrumentation overhead of DBI. However, none of them have so far considered the QoS of the application under analysis. Similar to QDIME, Cho *et al.* [13] and Arafa *et al.* [8] also propose to control both the instrumentation duration and frequency to switch the instrumentation on and off periodically. However, their techniques can guarantee only the timing property of programs.

Sampling has been widely used to reduce the runtime overhead by trading off the amount of extracted information [9], [21], [17], [24], [34]. Arnold and Ryder [9] collects low-overhead frequency profiles using a compiler-controlled counter-based sampling to switch between instrumented and uninstrumented code. Hirzel and Chilimbi [21] further extended this work to allow longer samples with lower overhead. To maximize the coverage of infrequently executed code, Hauswirth and Chilimbi [20] propose an adaptive profiling scheme that samples code regions at a rate inversely proportional to their execution frequency. Fischmeister and Ba [17] introduced a sampling-based monitoring approach for time-sensitive applications along with techniques to statically determine sampling periods. While sampling-based techniques mostly focus on performance profiling and optimization, QDIME also considers the overall QoS of applications.

VIII. Conclusion

This paper presents QDIME, a QoS-aware DBI technique that guarantees a certain QoS to the program under analysis. The evaluation on four real-world applications shows that QDIME respects the QoS threshold while maintaining an average coverage of 92%. QDIME also reduces the slow-down factors of the instrumented applications. These results make QDIME a powerful tool for instrumenting QoS-based applications and enable the design of dynamic analysis tools with QoS guarantees. For the future work, we are investigating new budget functions; for example utilizing adaptive learning approach and handling thread-specific constraints.

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

[5] Pin Trace-version APIs. https://goo.gl/qNQ6tu.

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

[7] P. Arafa, H. Kashif, and S. Fischmeister. Redundancy Suppression in Time-Aware Dynamic Binary Instrumentation. *preprint*. arXiv:1703.02873.

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

[9] M. Arnold and B. G. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *Proc. of the ACM SIGPLAN Conf. on Programming language design and implementation*. ACM, 2001.

[10] A. R. Bernat and B. P. Miller. Anywhere, Any-time Binary Instrumentation. In *Proc. of the Workshop on Program Analysis for Software Tools*. ACM, 2011.

[11] I. Böhm, T. J. Edler von Koch, S. C. Kyle, B. Franke, and N. Topham. Generalized Just-in-time Trace Compilation Using a Parallel Task Farm in a Dynamic Binary Translator. *SIGPLAN Not.*, 46(6), June 2011.

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

[13] H. K. Cho, T. Moseley, R. Hank, D. Bruening, and S. Mahlke. Instant Profiling: Instrumentation Sampling for Profiling Datacenters Applications. In *Int'l Symp. on Code Gen. and Optimization*, Feb 2013.

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

[15] A. Edwards, A. Srivastava, and H. Vo. Vulcan: Binary Transformation in a Distributed Environment. Technical report, Microsoft Research, April 2001.

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

[17] S. Fischmeister and Y. Ba. Sampling-based Program Execution Monitoring. *SIGPLAN Not.*, 45(4), 2010.

[18] A. Hamou-Lhadj and T. 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.

[19] A. Hassan, D. Martin, P. Flora, P. Mansfield, and D. 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.

[20] M. Hauswirth and T. M. Chilimbi. Low-overhead Memory Leak Detection Using Adaptive Statistical Profiling. *SIGPLAN Not.*, 39(11), Oct. 2004.

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

[22] N. Jia, C. Yang, J. Wang, D. Tong, and K. Wang. SPIRE: Improving Dynamic Binary Translation Through SPC-indexed Indirect Branch Redirecting. *SIGPLAN Not.*, 48(7), Mar. 2013.

[23] H. Kashif, P. Arafa, and S. Fischmeister. INSTEP: A Static Instrumentation Framework for Preserving Extra-Functional Properties. In *International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, Aug 2013.

[24] B. Kasikci, T. Ball, G. Candea, J. Erickson, and M. Musuvathi. Efficient Tracing of Cold Code via Bias-free Sampling. In *Proc. of the USENIX Annual Technical Conference*. USENIX Association, 2014.

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

[26] N. Kumar, B. R. Childers, and M. L. Soffa. Low Overhead Program Monitoring and Profiling. In *Proc. of the 6th ACM workshop on Program Analysis for Software Tools and Engineering*. ACM, 2005.

[27] J. R. Larus and E. Schnarr. EEL: Machine-independent Executable Editing. *SIGPLAN Not.*, 30(6), June 1995.

[28] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely. PEBIL: Efficient Static Binary Instrumentation for Linux. In *Symp. on Performance Analysis of Systems Software*. IEEE, 2010.

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

[30] T. Moseley, A. Shye, V. Reddi, D. Grunwald, and R. Peri. Shadow Profiling: Hiding Instrumentation Costs with Parallelism. In *Intl. Symp. on Code Gen. and Optimization*. ACM, Mar. 2007.

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

[32] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, April 2013.

[33] A. Oliveira, J.-C. Petkovich, T. Reidemeister, and S. Fischmeister. DataMill: Rigorous Performance Evaluation Made Easy. In *Proc. of the 4th ACM/SPEC International Conference on Performance Engineering*. ACM, April 2013.

[34] H. Pirzadeh, S. Shanian, A. Hamou-Lhadj, L. Alawneh, and A. Shafiee. Stratified Sampling of Execution Traces: Execution Phases Serving As Strata. *Sci. Comput. Program.*, 78(8), Aug. 2013.

[35] B. Sabata, S. Chatterjee, M. Davis, J. Sydir, and T. Lawrence. Taxonomy for QoS Specifications. In *Proc. of the International Workshop on Object-Oriented Real-Time Dependable Systems*, 1997.

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

[37] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. *SIGPLAN Not.*, 29(6), June 1994.

[38] G.-R. Uh, R. Cohn, B. Yadavalli, R. Peri, and R. Ayyagari. Analyzing Dynamic Binary Instrumentation Overhead. In *Proc. of the Workshop on Binary Instrumentation and Applications*, 2006.

[39] D. Upton, K. Hazelwood, R. Cohn, and G. Lueck. Improving Instrumentation Speed via Buffering. In *Proc. of the Workshop on Binary Instrumentation and Applications (WBIA)*, 2009.

[40] J. Zeng, Y. Fu, and Z. Lin. PEMU: A Pin Highly Compatible Out-of-VM Dynamic Binary Instrumentation Framework. In *Proc. of the Int'l Conference on Virtual Execution Environments*, VEE'15. ACM, 2015.

[41] Q. Zhao, I. Cutcutache, and W.-F. Wong. PiPA: Pipelined Profiling and Analysis on Multicore Systems. *ACM Trans. Archit. Code Optim.*, 7(3), 2010.

[42] Y. Zheng, L. Bulej, C. Zhang, S. Kell, D. Ansaloni, and W. Binder. Dynamic Optimization of Bytecode Instrumentation. In *Proc. of the Workshop on Virtual Machines and Intermediate Languages*. ACM, 2013.