DIME: Time-aware Dynamic Binary Instrumentation Using Rate-based Resource Allocation

Pansy Arafa   Hany Kashif   Sebastian Fischmeister

Dept. of Electrical and Computer Engineering
University of Waterloo, Canada
{parafa, hkashif, sfischme}@uwaterloo.ca

Abstract

Program analysis tools are essential for understanding programs, analyzing performance, and optimizing code. Some of these tools use code instrumentation to extract information at runtime. The instrumentation process can alter program behavior such as timing behavior and memory consumption. Time-sensitive programs, however, must meet specific timing constraints and thus require that the instrumentation process, for instance, bounds the timing overhead. Time-aware instrumentation techniques try to honor the timing constraints of such programs. All previous techniques, however, support only static source-code instrumentation methods. Hence, they become impractical beyond microcontroller code for instrumenting large programs along with all their library dependencies.

In this work, we propose DIME, a time-aware dynamic binary instrumentation technique that adds an adjustable bound on the timing overhead to the program under analysis. We implement DIME using the dynamic instrumentation framework, Pin. Quantitative evaluation of the three implementation alternatives shows an average reduction of the instrumentation overhead by 12, 7, and 3 folds compared to native Pin. Instrumenting the VLC media player and a laser beam stabilization experiment demonstrate the practicality and scalability of DIME.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging – Tracing

General Terms Experimentation, Performance

Keywords Instrumentation, Tracing, Profiling

1. Introduction

Program analysis tools are critical for understanding program behavior. Software developers need such tools to analyze their programs and identify performance bottlenecks [33]. Profiling is one approach for program analysis. Examples of program profiling include time and space profiling, collecting runtime statistics on instruction and function usage, and function call tracing. Profilers instrument programs to collect the required profiling information.

Since real-time software is time-sensitive and needs to obey timing constraints, real-time systems require specialized program analysis tools. Instrumentation naturally causes perturbation to the program under analysis. This perturbation might result in violations of timing constraints. Software-based instrumentation methods [17, 22] insert code in the original program to enable tracing/monitoring, which results in modifying the program's timing behavior. Hardware-based tracing methods [23, 30] are also known for causing significant perturbation to the program being traced [26]. Furthermore, hardware-based tracing methods collect low-level data and, hence, require higher-level support to provide traces at a higher-level of abstraction [24, 25]. Dynamic instrumentation approaches [6, 21, 29] modify the program during its execution causing large runtime overheads. Consequently, recent work [11, 16] focuses on time-aware instrumentation techniques that honor a program's timing constraints.

Time-aware instrumentation extracts information from a program without violating its timing constraints. Fischmeister et al. [11] 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). Kashif et al. [16] use program transformation techniques to increase the effectiveness of time-aware instrumentation. They achieve this by creating locations in the program where instrumentation can be inserted while having minimal effect on the program's worst-case behavior. In [15], Kashif et al. propose INSTEP; a static instrumentation framework for preserving extra-functional properties. INSTEP uses the input program and a user-specified instrumentation intent to derive a partial program. Then it solves an optimization problem based on cost models and constraints to generate the instrumented program. Other works propose frameworks for optimizing other program aspects such as code size and energy consumption [20, 27].

To our knowledge, 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 [3] 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 paper, we propose DIME, a time-aware dynamic binary instrumentation (DBI) technique as an extension to Pin [21]. The idea is to enable dynamic instrumentation of program binaries while still bounding the overhead of the instrumentation process. DIME, as a tool for instrumenting soft real-time applications, is practical, scalable, and supports multi-threaded applications. The contributions of this work are as follows.

1. Introduce the concept of time-aware DBI by employing rate-based resource allocation methods.
2. Present three implementations of DIME using Pin [21].
3. Quantitatively demonstrate the areas of applicability of the different approaches.
4. Demonstrate the general applicability and scalability of DIME on two case studies.

2. Background

This section overviews some necessary background information.

2.1 Binary Instrumentation

Instrumentation of binary executables can happen either statically before the program runs or dynamically during 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. Static binary instrumentation tools include EEL [19], ATOM [33], and Morph [38]. Static instrumentation methods are based on static analysis and cannot react to changes in application behavior at run time. DBI, as opposed to static binary instrumentation, does not require any preprocessing 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 code they can analyze, and, for example, cannot instrument dynamically generated code.

Multiple tools exist that support dynamic binary instrumentation. DBI tools that use code transformation during program execution include Dyninst [8] and Vulcan [10]. Most of these instrumentation tools suffer from transparency issues, i.e., they modify native behavior of the program under analysis [7]. Other DBI tools have software code caches and dynamically compile binaries such as Pin [21], DynamoRIO [6], and Valgrind [29]. We choose Pin as the underlying DBI framework to implement DIME because Pin is easily extensible and has lower overhead compared to other tools. This is discussed in detail in the following sections.

2.2 DBI Using Pin

Pin is a DBI framework developed by Intel [21]. Pin provides a cross-platform API for building program-analysis tools, and it 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. 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 [21, 31, 34].

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.

2.3 DBI Overhead

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 [21].

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 [9]:

- *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 [21] that the application slow down due to dynamic instrumentation by Pin [21], DynamoRIO [6], and Valgrind [29] 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 [34], 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 [29]. 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.

2.4 Time-aware Instrumentation

Time-aware instrumentation [11, 15, 16] is a mechanism for instrumenting programs while preserving functional correctness and timing constraints. The main idea is instrumenting a program at code locations that do not modify the program's WCET (or some other timing constraint) and at the same time preserves the program's original behavior. Instrumentation of a program using time-aware instrumentation techniques shifts the program's execution time profile towards its deadline. The authors in [16] 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 [15] develop an instrumentation framework, INSTEP, for preserving multiple competing extra-functional properties. 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. All previous work on time-aware instrumentation is based on static source-code instrumentation techniques. In this work, we also focus on time-aware instrumentation but using dynamic binary instrumentation methods.

2.5 Rate-based Resource Allocation

A rate-based system allows a specific task to run according to a previously defined rate, for example, *"x milliseconds per second"* [14]. 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 execution times, varying execution times, or varying executing rates. These models guarantee full resource utilization and overload-avoidance in a resource-constrained system. They also enable direct mapping of timing and importance constraints into priority values.

3. Overview of DIME

DIME is a dynamic time-aware binary instrumentation tool. 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 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.

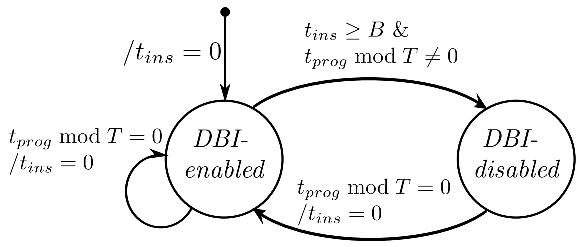


Figure 1: State-machine for DIME's operation.

Figure 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 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.

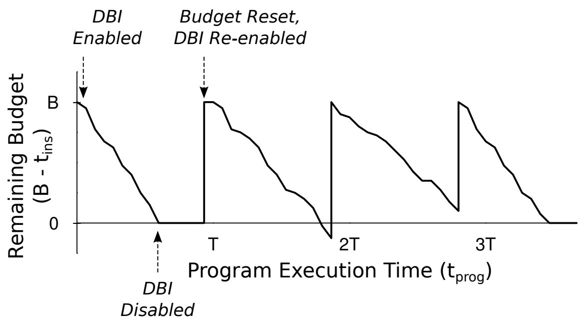


Figure 2: Rate-based DBI.

Figure 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.

4. Implementation Using Pin

This section describes the implementation of DIME. We mentioned, in Section 2.3, that instrumentation routines and dynamic compilation have a small overhead compared to the execution of the instrumentation code in the analysis routines [21]. 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 1: Handling of instrumentation budget in DIME.

Listing 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 5 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 data a 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 the original DBI (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. Continuous budget checking on the other hand 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.

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

4.1 *Trace Version*

Trace Version checks for budget at each instrumentation points 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 next instruction to be executed.

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 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.

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.

```
void instrumentation(...){
   InsertCall(budget_check);
   if(version == V_BASE) {
      //check switching to V_INSTRUMENT
      InsertVersionCase(1,V_INSTRUMENT);
   }
   else if(version == V_INSTRUMENT){
      //check switching to V_BASE
      InsertVersionCase(0,V_BASE);
   }
   switch(version) {
      case V_BASE:
      break; //Do Nothing
   case V_INSTRUMENT:
      ...
      InsertCall(analysis);
      ...
      break;
   }
}
```

Listing 2: Instrumentation routine of *Trace Version*.

4.2 *Trace Version Conditional*

Trace Version Conditional 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 *Trace Version Conditional* implementation of DIME also makes use of Pin's trace versioning APIs.

Trace Version Conditional 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 outlines the implementation of *Trace Version Conditional*. 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*. 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 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.

Trace Version Conditional 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, *Trace Version Conditional* has lower budget utilization compared to *Trace Version* because it postpones using the instrumentation budget till the start of a new trace.

```
void instrumentation(...){
   if(version == V_BASE &&
               budget_reset == true) {
      budget_reset = false;
      //check switching to V_INSTRUMENT
      InsertVersionCase(1,V_INSTRUMENT);
   }
   else if(version == V_INSTRUMENT){
      InsertCall(budget_check);
      //check switching to V_BASE
      InsertVersionCase(0,V_BASE);
   }
   ...
}
...
void sig_handler(...){
   budget_var = B;
   budget_reset = true;
}
```

Listing 3: Implementation of *Trace Version Conditional*.

4.3 *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 4: Instrumentation routine of *Trace Conditional*.

Trace Conditional performs all its budget checking in the instrumentation routine without any analysis-routine calls. Listing 4 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 *Trace Version Conditional*. 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

Trace Version Conditional). *Trace Conditional* also loosely honors the instrumentation budget compared to *Trace Version* and *Trace Version Conditional*. The is because switching to the *DBI-disabled* state only occurs at the beginning of a new trace.

4.4 Qualitative Comparison

Table 1 provides a qualitative comparison of the different implementations of DIME. The *Trace Version* and *Trace Version Conditional* 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 *Trace Version Conditional* and *Trace Conditional*). *Trace Version Conditional* 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 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.

5. Performance Evaluation

The qualitative evaluation of Section 4.4 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. 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 6 discuss the instrumentation coverage and applicability domains.

5.1 Experimental Setup

We experiment with the SPEC2006 C benchmark suite [13] which consists of integer and floating point benchmarks. SPEC2006 is a common benchmark for evaluating performance of dynamic instrumentation tools [6, 21, 29]. 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 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. Memory locking and stack-prefaulting prevent page faults and stack faults from introducing non-deterministic behavior. 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 6 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:

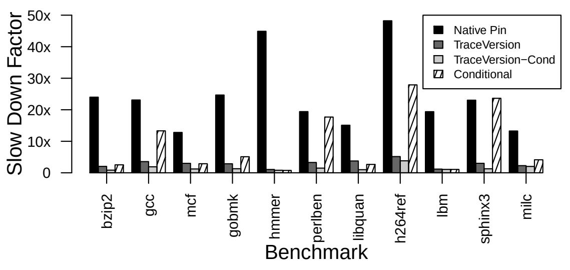
- **Slow down factor of the dynamically instrumented program:** The slow down factor is the ratio of the execution time of the instrumented benchmark running on top of Pin to the execution time of the natively running benchmark. This metric highlights the overhead reduction of DIME compared to native 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.

5.2 Experimental Results

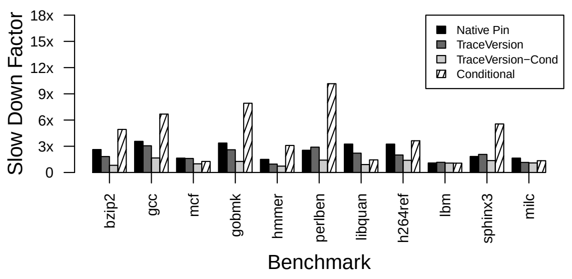
DIME, on average, outperforms native Pin in terms of overhead in the heavy-weight analysis-routine tools: dcache and topopcode. Figures 3a and 3d show the slow-down factors of Pin and DIME implementations with dcache and topopcode tools, respectively. The average slow down of native Pin with dcache is 24.3x, and with topopcode is 29.6x. *Trace Version* and *Trace Version Conditional* 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

Table 1: Qualitative comparison of the different DIME implementations.

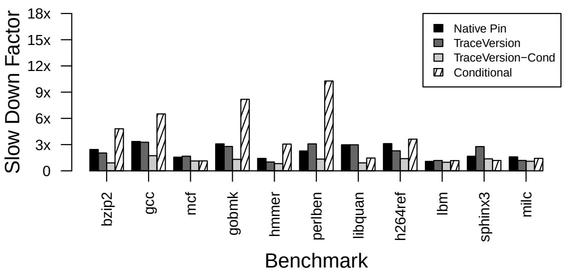
	Trace Version	***Trace Version Conditional***	***Trace Conditional***
Analysis-routine call for budget checking	Yes in both states	Only in *DBI-enabled* state	No
Switch to *DBI-disabled* state	Immediate	Immediate	Delayed till start of new trace
Switch to *DBI-enabled* state	Immediate	Delayed till start 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



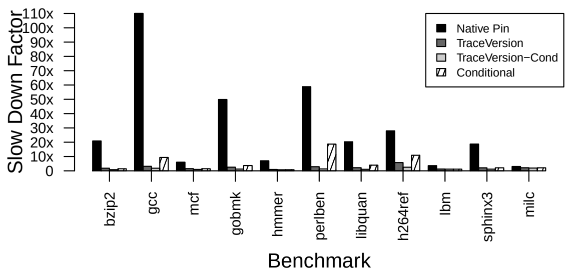
(a) Dcache



(b) Inscount



(c) Regmix



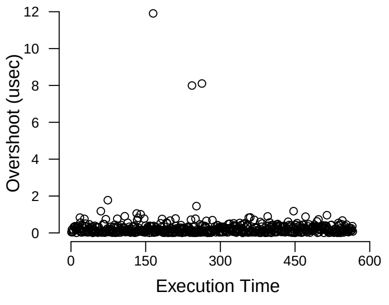
(d) Topopcode

Figure 3: Slow-down factors of native Pin and the three implementations of DIME.

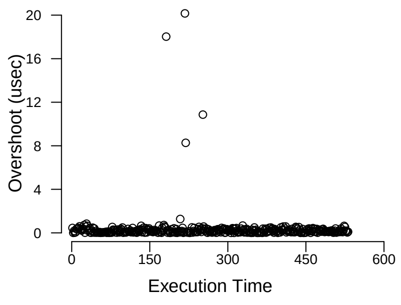
topopcode, respectively. This reflects the higher budget checking overhead of *Trace Version* over *Trace Version Conditional*. *Trace Conditional* has a slow down of 9.2x with dcache and 5x with topopcode. This is due to the frequent overshoots of *Trace Conditional* beyond the instrumentation budget.

Trace Version Conditional 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 *Trace Version Conditional* maintain an average slow down of 1.9x and 1.1x, respectively, as shown in Figure 3b. Figure 3c presents the slow down with regmix, which are 2.2x, 2.2x, and 1.1x for native Pin, *Trace Version*, and *Trace Version Conditional*, 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 [21]. 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 overhead. The results for *Trace Conditional*, in Figures 3b and 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 *Trace Version Conditional* is lower than that of *Trace Version*, since *Trace Version Conditional* has a lower budget-checking overhead (as discussed in Section 4.4). Additionally, both *Trace Version* and *Trace Version Conditional* incur lower slow down in average compared to *Trace Conditional*.

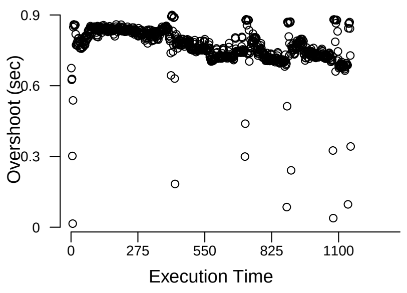
Trace Version and *Trace Version Conditional* have a very low overshoot value (order of microseconds) compared to *Trace Conditional*. Figure 4 shows the overshoot magnitude for the three implementations of DIME over the execution time of the mcf benchmark while instrumenting it using the DIME version 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 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 4a and 4b present the values of overshoots for *Trace Version* and *Trace Version Conditional*, respectively. The values for the most frequent overshoots lie below 2 usec for both *Trace*



(a) *Trace Version*



(b) *Trace Version Conditional*



(c) *Trace Conditional*

Figure 4: Overshoots of the three implementations of DIME with the mcf benchmark and dcache tool.

Version and *Trace Version Conditional*. 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 *Trace Version Conditional* which always maintains a lower slow down compared to Pin. Both *Trace Version* and *Trace Version Conditional* 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. Both *Trace Version* and *Trace Version Conditional* strictly respect the instrumentation budget. *Trace Version Conditional* is well-suited for applications that require very low instrumentation overhead. *Trace Version* has a higher raw instrumentation coverage compared to *Trace Version Conditional* but with a relatively higher overhead.

6. 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 [28].

6.1 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 [3]. For this case study, we use the latest version of the VLC media player, v2.0.5. As mentioned in Section 1, 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 [32]. 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 the unmodified pintool and from the partial traces generated by DIME [32]. The time period T is set to one second throughout this case study. Table 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

Table 2: Results for the VLC case study.

		TV.	TV-Cond.	Tr.-Cond.
1	Max budget w.o. pauses	14%	38%	22%
2	CC coverage in one run	93.2% 90.2%	92.8% 90.9%	83.4% 75.0%
3	Runs for 98% of CC tree	4	4	5
4	Runs for 99% of CC tree	5	5	6
5	Raw coverage at min. budget	23.6%	17.6%	62.8%

recorded and are available for viewing[1]. The original video has 599 blocks that VLC decodes for viewing frames. VLC, using the unmodified pintool, 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 *Trace Version Conditional* extract 90% of the call context tree in one run. Table 2 at row 1 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. Table 2 at row 5 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 *Trace Version Conditional*. This is because *Trace Conditional* loosely obeys the budget and *Trace Version Conditional* has a delayed switch to the *DBI-enabled* state.

6.2 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 [2]. Laser beam stabilization is an important technology currently used in manufacturing equipment, surveillance, aircraft targeting, etc. The experiment consists of a stationary

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

Table 3: Results for the LBS case study.

	Average Displac. (mm)	Displac. Variance (mm)	Stability Budget (%)	Memory Pattern (%)
Orig. LBS	0.022	0.0002	N/A	N/A
Native Pin	2.988	0.487	N/A	100
TV.	0.509	0.881	0.3%	<1%
TV-Cond.	0.588	0.531	0.6%	<1%
Trace-Cond.	0.129	0.183	9.0%	78.1%

laser beam source pointing 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 [1] and Valgrind [29]. 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. The unmodified pintool fails to stabilize the laser beam which has a jittery response all over the detector. The instability of the unmodified version of the pintool is visible from the average displacement shown in Table 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 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 *Trace Version Conditional* 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. Assessing the effect of the program structure on the different implementations of DIME is part of our future work.

7. Discussion

This section discusses some current limitations of DIME.

Instrumentation Parameters and Program Structure: In this work, we assessed the effect of the instrumentation and analysis-routine complexity on DIME. The results show that program structure as well affects the performance of DIME and consequently affects the choice of DIME's implementation to use. Varying the instrumentation period T can also have an effect on the performance of DIME and choice of implementation. Assessing the effect of changing these factors is part of our future work. The results, however, demonstrate the applicability of DIME for different programs at the default instrumentation period of one second.

Pin Specific: DIME is based on Pin which mainly targets IA-32 and Intel64 architectures. Pin supports multiple operations 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 this work. Note also that Pin is a popular and well-supported instrumentation tool with over 300 000 downloads and 700 citations [9]. Otherwise, time-aware DBI using rate-based resource allocation is a concept that can be integrated into other DBI frameworks.

8. Related Work

Naik and Palsberg [27] present a framework for code-size-aware compilation. They formulate register allocation as an Integer Linear Programming (ILP) problem. Lee et al. [20] introduce a framework to balance the tradeoffs between code size, execution time, and energy consumption when developing an embedded system. These frameworks are compile-time techniques for balancing tradeoffs and optimizing certain code properties.

Wallace and Hazelwood [36] 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. Moseley et al [25] 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 [37, 39]. DIME can be extended to utilize these parallelization techniques.

Upton et al. [35] 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. [18] optimize the instrumentation code to decrease the DBI overhead. They reduce the cost of instrumentation probes that intercept program execution, in addition to the number of instrumentation points, and the cost of each point. These instrumentation approaches ignore the program's temporal constraints. They, however, can complement DIME to achieve better performance and instrumentation coverage.

Arnold and Ryder [4] 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. 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 the performance of time-sensitive applications [5]. Other sampling-based techniques have been proposed for performance optimizations [12]. These techniques either apply optimizations specific to the instrumentation objective or use compiler-specific information to perform optimizations. Again, these can be used to complement DIME which is a generic approach to time-aware instrumentation.

9. Conclusion

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. In this work, we introduce 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. Our future work includes assessing the effect of the program structure on the performance of DIME as well as investigating the effect of varying the instrumentation budget and period.

References

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

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

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

[4] 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 (PLDI)*, 2001.

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

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)*, 2003.

[7] D. Bruening, Q. Zhao, and S. Amarasinghe. Transparent Dynamic Instrumentation. *SIGPLAN Not.*, 47(7), Mar. 2012.

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

[9] T. Devor. Pin Tutorial. CGO'12, San Jose, California, 2012.

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

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

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

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

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

[15] H. Kashif, P. Arafa, and S. 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.

[16] H. Kashif and S. Fischmeister. Program transformation for time-aware instrumentation. In *Proc. of the 17th IEEE Intl. Conf. on Emerging Technologies & Factory Automation (ETFA)*, Sep. 2012.

[17] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A Run-Time Assurance Approach for Java Programs. *Form. Methods Syst. Des.*, 24(2), 2004.

[18] N. Kumar, B. R. Childers, and M. L. 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.

[19] J. R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. *SIGPLAN Not.*, 30, 1995.

[20] S. Lee, I. Shin, W. Kim, I. Lee, and S. L. Min. A Design Framework for Real-time Embedded Systems with Code Size and Energy Constraints. *ACM Trans. Embed. Comput. Syst.*, 7(2), Jan. 2008.

[21] 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. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2005.

[22] 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.

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

[24] P. Mork. Techniques for Debugging Parallel Programs. Technical report, University of Miskolc.

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

[26] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney. We have it Easy, but do we have it Right? *IEEE Intl. Symp. on Parallel and Distributed Processing*, 2008.

[27] M. Naik and J. Palsberg. Compiling with Code-size Constraints. *ACM Trans. Embed. Comput. Syst.*, 3(1), Feb. 2004.

[28] M. Naik, H. Yang, G. Castelnuovo, and M. Sagiv. Abstractions from Tests. *SIGPLAN Not.*, 47(1), Jan. 2012.

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

[30] W. Omre. Debug and Trace for Multicore SoCs. Technical report, ARM, 2008.

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

[32] M. Serrano and X. Zhuang. Building Approximate Calling Context from Partial Call Traces. In *Proc. of the 7th Annual IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO)*, 2009.

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

[34] G.-R. Uh, R. Cohn, B. Yadavalli, R. Peri, and R. Ayyagari. Analyzing Dynamic Binary Instrumentation Overhead. 2007.

[35] 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.

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

[37] Z. Yu, W. Zhang, and X. 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.

[38] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System Support for Automatic Profiling and Optimization. *SIGOPS Oper. Syst. Rev.*, 31, 1997.

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