FADATest: Fast and Adaptive Performance Regression Testing of Dynamic Binary Translation Systems

Jin Wu
Harbin Institute of Technology
China

Jian Dong
Harbin Institute of Technology
China

Ruili Fang
University of Georgia
USA

Wen Zhang
University of Georgia
USA

Wenwen Wang
University of Georgia
USA

Decheng Zuo
Harbin Institute of Technology
China

ABSTRACT

Dynamic binary translation (DBT) is the cornerstone of many important applications. In practice, however, it is quite difficult to maintain the performance efficiency of a DBT system due to its inherent complexity. Although performance regression testing is an effective approach to detect potential performance regression issues, it is not easy to apply performance regression testing to DBT systems, because of the *natural differences* between DBT systems and common software systems and the *limited availability* of effective test programs. In this paper, we present FADATest, which devises several novel techniques to address these challenges. Specifically, FADATest automatically generates *adaptable* test programs from existing real benchmark programs of DBT systems according to the runtime characteristics of the benchmarks. The test programs can then be used to achieve highly *efficient* and *adaptive* performance regression testing of DBT systems. We have implemented a prototype of FADATest. Experimental results show that FADATest can successfully uncover the same performance regression issues across the evaluated versions of two popular DBT systems, QEMU and Valgrind, as the original benchmark programs. Moreover, the testing efficiency is improved significantly on two different hardware platforms powered by x86-64 and AArch64, respectively.

CCS CONCEPTS

• **Software and its engineering** → **Source code generation**; **Software performance**; **Simulator / interpreter**.

KEYWORDS

Performance regression testing, DBT, Test program generation

ACM Reference Format:
Jin Wu, Jian Dong, Ruili Fang, Wen Zhang, Wenwen Wang, and Decheng Zuo. 2022. FADATest: Fast and Adaptive Performance Regression Testing of Dynamic Binary Translation Systems. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510169

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510169

1 INTRODUCTION

Dynamic binary translation (DBT) is a key enabling technology for many important applications, such as whole program analysis [6, 33], hardware simulation [21, 32], heterogeneous computation offloading [38], and software fuzz testing [9, 23]. In essence, a DBT system dynamically translates executable binary code from a *guest* instruction set architecture (ISA) to a *host* ISA, which can be different from or same as the guest ISA. By executing the generated host code on a physical host machine, the DBT system can either emulate the semantics of the guest application or enhance its functionality, e.g., execution tracing for performance analysis.

Despite the vital importance of DBT, it is still quite challenging to develop and maintain an *efficient* DBT system in practice. Typically, the translation process in a DBT system maps a guest instruction into one or more host instructions, which together emulate the semantics of the guest instruction. Given the semantic gaps between different ISAs, the translated host binary code often suffers from significant code explosion. For example, an x86-64 instruction may be translated into dozens of AArch64 instructions due to the differences between the two ISAs. Therefore, in general, the performance of a guest application running with DBT is remarkably *worse* than its native performance. Besides, to support code translations across various ISAs in one system, DBT developers usually have to create a huge code base. Even worse, to keep pace with the rapidly evolving hardware architectures, existing DBT systems need to be updated frequently to support emerging machine instructions. These factors inevitably render the inherent difficulty of maintaining the performance efficiency of a DBT system.

To give an example, QEMU [3] is a well-known DBT system and has been widely used in many research projects and real-world products [29]. It has around 2.8M lines of source code. Every day, an average of 10 commits are added to the code base by different developers, embodying 25 source lines revised in each commit on average. As a consequence, a tiny and seemingly innocuous modification to the large code base may introduce an unexpected impact on the performance of the system. Indeed, we have observed multiple performance regression issues between two successive release versions, which enclose an average of 2188 code commits. This apparently makes it quite difficult and time-consuming to debug and fix the performance issues.

Therefore, in this paper, we advocate that it is necessary and imperative to conduct *performance regression testing* during the *daily* development of a DBT system. This will allow DBT developers to notice unexpected performance regressions at a very early stage

and thus reduce the tremendous engineering effort needed to fix the problems. Performance regression testing has been demonstrated to be a practical approach to detect potential performance inconsistencies between two different versions of a software system. It has been extensively adopted during software development cycles in commercial companies [1, 11, 17], and a large amount of research work has been devoted to studying and enhancing performance regression testing [8, 13, 14, 16, 22, 24–27, 30, 35].

However, unfortunately, performance regression testing of DBT systems faces several unique challenges. First, the performance of a DBT system is usually evaluated using industry-standard and classical benchmark suites, such as SPEC CPU 2017 [34] and PARSEC [5]. Due to the aforementioned performance overhead incurred by DBT, it takes an *extremely long* time, even with most recent DBT optimizations applied [15, 36, 37, 40–42], to complete the testing of an entire benchmark suite. For instance, QEMU needs more than three hours to finish the execution of a *single* benchmark program in SPEC CPU 2017. Note that, although these benchmark suites are shipped with input data sets in different scales, the smaller data sets are primarily used for correctness verification rather than performance testing. Second, different from most software systems, a DBT system takes as input *executable binary code* instead of regular program data. This makes it extraordinarily difficult to generate effective test inputs for performance regression testing, as each test input needs to be a binary code in guest ISA compiled from a test program. Simply using artificial test programs would not help because they will *not* be able to expose the same performance issues as real benchmark programs. Last but not least, DBT systems usually run on diverse platforms, ranging from servers, to desktops, mobile and embedded devices. Given the dramatically different computing power of these platforms, it is obviously unreasonable to use a set of fixed test programs to conduct performance regression testing of a DBT system on all of these platforms, because this may lead to either inaccurate testing results on high-performance platforms or poor testing efficiency on low-power platforms.

To address the above challenges and make DBT performance regression testing possible and practical, we propose FADATest in this paper. FADATest aims to realize *fast* and *adaptive* performance regression testing of DBT systems. To this end, FADATest first intelligently captures runtime characteristics of real benchmark programs through dynamic program profiling. Based on the collected information, FADATest then automatically generates *adaptable* test programs to preserve the characteristics of the benchmark programs. The generated test programs are able to accurately simulate the behaviors and performance results of original benchmark programs, while the execution times of the test programs are significantly shorter. Therefore, the test programs can replace the original benchmark programs to achieve efficient performance regression testing of DBT systems. Furthermore, the test size of each test program generated by FADATest can be easily scaled up/down on demand to fit the target hardware platform of a DBT system.

We have implemented a research prototype of FADATest. To evaluate the effectiveness of FADATest, we employ two major performance benchmark suites of DBT systems, SPEC CPU 2017 and PARSEC. Specifically, we utilize both the original benchmark programs and the generated test programs to test the performance of two widely-used DBT systems, QEMU and Valgrind, across different release versions. Experimental results show that the performance results of the generated test programs strongly match with those of the original benchmark programs. In other words, all performance regression issues in the evaluated versions that can be detected by the original benchmark programs are also successfully discovered by the test programs generated by FADATest. More importantly, the testing efficiency is enhanced significantly, compared to the original benchmark programs. This demonstrates the capability of FADATest to conduct efficient performance regression testing of DBT systems. In addition, the evaluation results on a low-power AArch64 platform show that the adaptability of the generated test programs allows FADATest to achieve the high testing efficiency on this hardware platform without loss of testing accuracy.

In summary, this paper makes the following contributions:

- We present FADATest, which integrates novel techniques to realize efficient and adaptive performance regression testing of DBT systems. FADATest automatically generates test programs for performance regression testing according to the dynamic characteristics of real benchmarks.
- We implement a research prototype of FADATest. The prototype supports two mainstream ISAs on the market, x86-64 and AArch64. Our implementation also overcomes several technical obstacles caused by executing special hardware instructions in the generated test programs.
- We conduct comprehensive experiments to evaluate the effectiveness of FADATest. The evaluation results show that FADATest can successfully reveal performance regression issues of DBT systems. Compared to the original benchmark programs, the testing efficiency is significantly improved.

2 BACKGROUND AND MOTIVATION

How DBT Works? In general, a DBT system translates guest binary code at the *basic block* (or block) granularity. Each block contains a sequence of guest instructions with at most one branch instruction at the end of the block. That is, a block has only one entry and one exit and the execution of a block is sequential. The translated host binary code is saved to a software *code cache* to mitigate the translation overhead as a block may be executed multiple times in the same execution. Once the translation of a block is completed, the execution flow is transferred to the code cache so that the translated host binary code can be executed. Therefore, the execution of a DBT system is typically interleaved by the code translation and the execution of the translated host binary code, until all dynamically discovered blocks are translated.

High Performance Overhead of DBT. Due to the significant semantic gaps between different ISAs, the translated host binary code often suffers from substantial code explosion, which undoubtedly leads to heavy performance overhead. To give an example, Figure 1 shows the normalized execution times of QEMU (version 5.0.0) and Valgrind (version 3.16.0) with benchmark programs in PARSEC on two different hardware platforms: x86-64 and AArch64. More details about the configuration of the two platforms can be found in Section 5. The performance baseline is the execution time of the corresponding benchmarks running natively on the x86-64 platform. As we can see from the figure, QEMU introduces an average of 6× and 36× performance slowdown for the x86-64 and AArch64

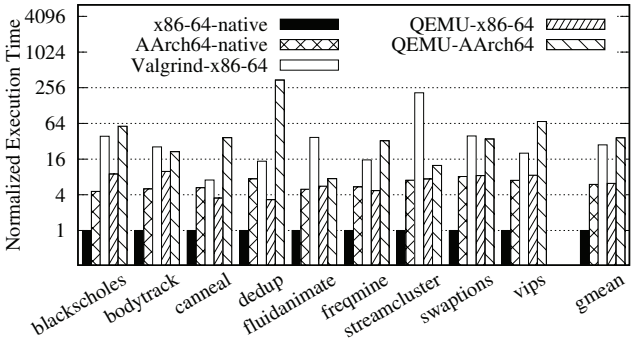

Figure 1: Normalized execution times of QEMU and Valgrind on x86-64 and AArch64 platforms. The baseline is the native benchmark execution time on the x86-64 platform.

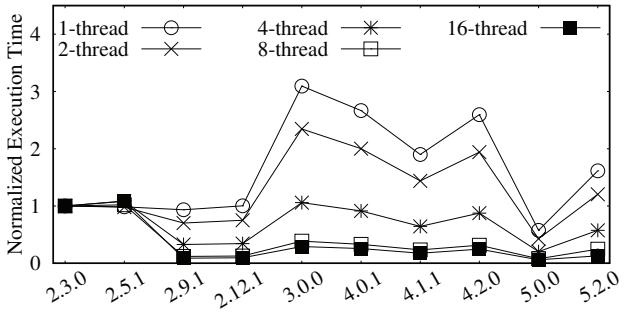

Figure 2: Normalized execution times of different versions of QEMU with the `swaptions` benchmark in PARSEC under different numbers of threads. The baseline is the execution time of QEMU (version 2.3.0) of each thread configuration.

platforms, respectively. Similarly, Valgrind introduces an average of 28× performance slowdown on the x86-64 platform. We failed to collect the data of Valgrind on the AArch64 platform due to the unbearably long execution time. This result demonstrates the slow execution of programs running atop a DBT system. From this figure, we can further conclude that the performance of the benchmarks is much worse on the AArch64 platform (i.e., QEMU-AArch64). This is mainly because the computing power of the AArch64 processor is much lower than the x86-64 processor on our platforms. On the other side, this shows the difficulty to achieve a similar testing efficiency when using the same programs to test the performance of a DBT system on different hardware platforms.

Performance Regressions of DBT. Given the inherent complexity of modern ISAs, developers have to manually create a huge code base to support various guest and host instructions in a DBT system. For example, one of the source files related to translating AArch64 instructions in QEMU contains around 15K lines of source code. Besides, to improve the translation quality, as well as support emerging hardware instructions, such as single instruction multiple data (SIMD) instructions, frequent changes to the source code are very common during the development and maintenance process. This unavoidably leads to performance variance across different versions of a DBT system. For example, Figure 2 illustrates the performance results of different release versions of QEMU with the `swaptions` benchmark in PARSEC under different numbers of threads. As shown in the figure, the performance of QEMU is not consistent across different versions, e.g., the performance is decreased significantly from 2.12.1 to 3.0.0. Though the performance is increased later on in 4.0.1, it is not clear whether the previous performance degradation is fixed or not. Therefore, in order to monitor and maintain the performance efficiency of a DBT system, it is necessary and urgent to conduct performance regression testing during the daily development of the DBT system.

3 FADATEST

In general, there are several principles for designing a practical performance regression testing approach for DBT systems:

- **High Testing Efficiency**. Given the heavy performance overhead incurred by DBT systems, the testing time should be sufficiently short. This allows the testing approach to be used during the daily development of a DBT system to rapidly discover performance regression issues.
- **High Testing Accuracy**. The testing approach is expected to have a strong capability to accurately uncover potential performance regressions of a DBT system. This means representative DBT performance benchmark suites can be replaced to avoid extremely long testing times.
- **High Testing Adaptability**. Since a DBT system may run on a broad range of hardware platforms, it is necessary and required that the testing approach can be adapted seamlessly to the actual target platform of the DBT system without loss of the high testing efficiency and accuracy.
- **Low Manual Effort**. It is typically not acceptable if the testing approach needs to put additional engineering effort on DBT developers. Moreover, the testing approach should be very easy to use and helpful so that DBT developers would like to adopt it in their development cycles.

The key to realize practical performance regression testing of DBT systems is to generate short yet effective test programs, which can be used to test DBT performance regularly. A naïve approach is to randomly sample the execution of a selected benchmark program to collect a list of basic blocks and then assemble them together to form a test program. Though this approach can potentially shorten the testing time, it does not comply with the above principles. For example, it is hard to determine an appropriate sampling rate so that the generated test program can achieve acceptable testing efficiency and accuracy. Also, the test programs generated using this approach are fixed and thus hard to be scaled up/down to fit different hardware platforms. In contrast, the design of FADATest is compliance with these principles. It combines several novel techniques to realize fast, accurate, and adaptive performance regression testing of DBT systems. Next, we present a high-level overview of FADATest, followed by a detailed description of each component.

3.1 System Overview

FADATest takes several steps to generate a test program from a seed DBT performance benchmark program. The seed benchmark program may come from an existing benchmark suite, e.g., SPEC

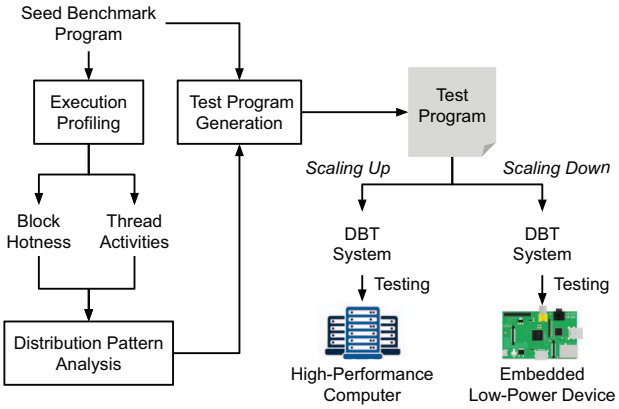

Figure 3: System overview of FADATest.

CPU 2017 or PARSEC, or provided by DBT developers with specific performance testing purposes. Figure 3 shows the high-level workflow of FADATest.

As shown in the figure, FADATest first collects runtime characteristics of the seed benchmark program through online execution profiling. This enables FADATest to understand the dynamic behavior of the seed program. Next, FADATest performs an in-depth analysis on the gathered information to determine the distribution patterns of the program. This allows FADATest to generate a distribution pattern to summarize how the basic blocks are distributed across different threads and how the number of threads influences such a distribution. With this distribution pattern, the final step of FADATest is to generate the test program. More specifically, FADATest includes all blocks that may affect the performance of the DBT system in the test program. Besides, the test program is parameterized in two dimensions. First, the dynamic execution counters of blocks can be adjusted to increase/decrease the testing time. Second, the number of threads can be changed to test specific thread settings of the DBT system. Both of them can be easily tuned through the options of the test program provided to users. This way, the generated test programs can test DBT performance on different hardware platforms, as shown in the figure.

3.2 Program Execution Profiling

In theory, we can extract a test program *statically* from a seed benchmark program, at either source code level or binary code level. However, this approach may lead to *inaccurate* testing results due to the lack of the important dynamic execution characteristics of the seed benchmark program. To avoid this problem, the first step of FADATest is to collect the execution profile of the seed benchmark program. To this end, FADATest executes the seed program with the standard input data set provided by the benchmark suite. For example, FADATest uses the *reference* input to collect the runtime characteristics of benchmark programs in SPEC CPU 2017. Note that SPEC CPU 2017 also provides the *test* and *train* inputs, which are smaller than *reference*, but they are generally not intended for performance testing and thus not considered by FADATest. During the execution of the seed program, FADATest collects two major types of profiling information: *block hotness* and *thread activities*.

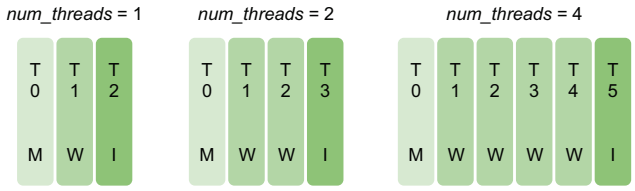

Figure 4: The actual number of threads created by the benchmark `bodytrack` in PARSEC is different from the number of threads specified by the input parameter *num_threads*. M: main thread, W: worker thread, I: I/O thread.

Block Hotness Profiling. Here, the "hotness" of a basic block means the times the block is executed dynamically under the provided input data set. The reason why FADATest collects the hotness information at the basic block level is because basic block is the translation unit of DBT systems. A *hot* basic block typically implies that the block is executed frequently and thus more likely to account for a *higher* proportion of the entire execution time of the DBT system, compared to a *cold* block. FADATest creates a *hotness counter* for every basic block translated by the DBT system and increases the counter by one each time when the translated host binary code of the block is executed. Note that the hotness counter is *thread private*, which means each thread has its own block hotness data. This allows FADATest to perform a *thread-aware* distribution pattern analysis in the following step.

Thread Activity Profiling. Thread activity information includes the specific time points at which a thread is created and terminated. Instead of using the absolute time, FADATest employs relative times, which eliminate any potential inaccuracies caused by uncertainty factors, such as thread synchronization and scheduling. Specifically, when a thread is created/terminated, FADATest marks down the corresponding execution point of its parent thread, i.e., the number of basic blocks that have been executed in the parent thread. With this thread activity information, the test program generated by FADATest is able to exhibit a similar thread behavior to the seed program. This is important as the generated test program may be used to test DBT performance with various thread settings, which cannot be profiled *completely* in advance. For example, a test program generated based on the execution profiles of 1, 2, and 4 threads may be used to test the performance of 8 threads or more.

3.3 Distribution Pattern Analysis

The purpose of the distribution pattern analysis is to determine how the execution of a basic block is distributed among different threads, so that the generated test program can faithfully simulate the thread behaviors of the seed benchmark program. In particular, the distribution pattern analysis aims to answer the following two questions. First, how is the execution of a basic block changed when the number of threads is scaled up/down? Second, what is the exact distribution of a basic block among different threads under a specific number of threads? To answer these questions, FADATest conducts a comprehensive distribution pattern analysis based on the profiled block hotness and thread activity information.

Algorithm 1: Thread Grouping Analysis

Input: *HotnessList* - The list of block hotness arrays, where each array corresponds to a block and is indexed by thread id
Output: *ResultGroups* - The result thread groups

```
1  NumThreads ← GetNumThreads(HotnessList);
2  ResultGroups ←<< 1, 2, ..., NumThreads >>;
3  foreach hotness array HA in HotnessList do
4      HasZero ← FALSE;
5      HasNoneZero ← FALSE;
6      foreach group g in ResultGroups do
7          for thread id tid in g do
8              if HA[tid] = 0 then
9                  HasZero ← TRUE;
10             else
11                 HasNoneZero ← TRUE;
12             end
13         end
14         if HasZero and HasNoneZero then
15             (g1, g2) ← SplitGroup(g);
16             ResultGroups.Remove(g);
17             ResultGroups.Append(g1, g2);
18         end
19     end
20 end
21 return ResultGroups;
```

Thread Grouping Analysis. Generally, a multi-threaded program exports to users an adjustable parameter, e.g., *num_threads*, which allows users to specify the number of threads for the execution of the program. Intuitively, the value of *num_threads* is the same as the actual number of threads created by the program. But, our observations on the multi-threaded benchmark programs from PARSEC show that the actual number of the created threads is probably *greater* than the specified value of *num_threads*. For example, `bodytrack` is a PARSEC benchmark. When we run this benchmark with a specified number of threads 1, i.e., $num\_threads = 1$, the actual number of the created threads is 3. Similarly, if the specified number of threads is 2 and 4, the actual number of the created threads is 4 and 6, respectively. Figure 4 illustrates this behavior. Further investigation shows that this benchmark always creates a main thread, an I/O thread, and *num_threads* worker threads. In other words, the main thread and the I/O thread do *not* scale when the value of *num_threads* increases. Therefore, the first step of the distribution pattern analysis in FADATest is to understand how the threads of the seed benchmark program are grouped and how the number of threads in each group scales.

To this end, FADATest analyzes the profiled basic block hotness data to identify the thread grouping pattern. This is inspired by the key observation that the threads in the same group should execute same/similar basic blocks. By searching for *group-private* blocks, which are executed by one group but not others, FADATest can rapidly identify thread groups in an execution profile. FADATest represents the thread grouping pattern using a two-dimensional array: <<α_1, β_1>, <α_2, β_2>, ..., <α_n, β_n>>, where $\alpha_i \times num\_threads + \beta_i$ is the actual number of threads in the group i, $1 \leq i \leq n$.

Table 1: Thread grouping patterns identified by FADATest for seed benchmark programs in PARSEC.

	Thread Grouping Pattern
blackscholes	<<0, 1>, <1, 0>>
bodytrack	<<0, 1>, <1, 0>, <0, 1>>
canneal	<<0, 1>, <1, 0>>
dedup	<<0, 1>, <1, 0>, <1, 0>, <1, 0>, <1, 0>, <0, 1>>
fluidanimate	<<0, 1>, <1, 0>>
freqmine	<<0, 1>, <1, -1>>
streamcluster	<<0, 1>, <1, 0>, <1, 0>, <1, 0>, <1, 0>, <1, 0>, <0, 1>>
swaptions	<<0, 1>, <1, 0>>
vips	<<0, 1>, <1, 0>>

Algorithm 1 describes how FADATest identifies thread groups. Initially, there is only one thread group. The analysis attempts to split the initial group into multiple groups by scanning the hotness data of all basic blocks to identify group-private blocks. A group is split into two groups if at least one thread in the group executes some blocks that are not executed by any other thread(s). By applying this algorithm to multiple execution profiles of the same seed benchmark program with different numbers of threads, e.g., 1, 2, and 4, FADATest can obtain multiple thread grouping results. With these grouping results, FADATest can further infer the thread grouping pattern by analyzing the relationship between the total number of threads in each group and the value of *num_threads*.

Recall the `bodytrack` benchmark in Figure 4, the thread grouping pattern identified by FADATest is <<0, 1>, <1, 0>, <0, 1>>. We further list the thread grouping patterns identified by FADATest for seed benchmark programs from PARSEC in Table 1. The table shows that every benchmark has more than one thread group. `streamcluster` even has seven groups. This explains the necessity of the thread grouping analysis. Indeed, without this information, it will be extremely hard, if not impossible, to correctly scale up the number of threads in the generated test program and match the thread behaviors of the seed benchmark program.

Block Hotness Pattern Analysis. With the thread grouping information, FADATest next analyzes the distribution patterns of basic block hotness. FADATest achieves this through two steps. First, FADATest identifies the block hotness patterns at the thread group level, i.e., the relationship between the total execution counters of a basic block inside a group and the number of threads in the group. Second, FADATest figures out the distribution patterns of basic block hotness among threads inside a group. Note that FADATest analyzes the hotness distribution pattern for *each* basic block, as different blocks often exhibit different behaviors.

Given a thread group G with m threads, FADATest creates a simple yet accurate model to compute the total execution count of a block in G: $\theta \times m + \delta$, where θ and δ are the model parameters and can be calculated based on the method of least squares approximation using the profiled execution counts of the block in G with different thread numbers. Suppose the actual execution count of the block is C_1 and the count computed by the model is C_2. The *error rate* of the model is determined by $\frac{|C_1 - C_2|}{C_1}$. Our experiences with standard multi-threaded DBT performance benchmark programs show that, within an error rate less than 10%, this model is able to

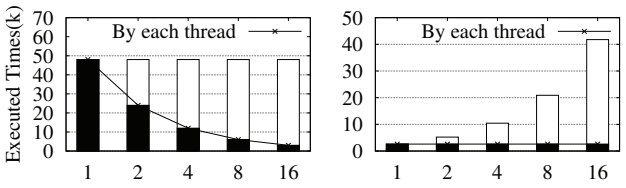

Figure 5: Hotness distribution patterns of two different basic blocks of `bodytrack` in the same thread group. The x-axis is the number of threads in the group.

predict the execution counts of more than 90% basic blocks. For the remaining blocks, they often have very small execution counts and thus can be excluded from the generated test program. To summarize, FADATest can accurately compute the hotness count of a basic block for each thread group through this model.

Next, FADATest analyzes the distribution pattern of a block inside a group. It is worth pointing out that this pattern typically does not change when the number of threads in the group changes. FADATest uses a vector to denote the pattern: $<\gamma_1, \gamma_2, ..., \gamma_m>$, where m is the number of threads in the group and $\sum_{i=1}^{m} \gamma_i = 1$. To simplify the design, FADATest heuristically treats a distribution pattern $<\gamma_1, \gamma_2, ..., \gamma_m>$ as an evenly distributed pattern, i.e., $\gamma_1 = \gamma_2 = ... = \gamma_m = \frac{1}{m}$, if $\forall i \in \{1, ..., m\}$, $\frac{|\gamma_i - 1/m|}{1/m} < 0.1$. This allows FADATest to accurately capture the pattern for more than 95% basic blocks.

Figure 5 shows two different distribution patterns analyzed by FADATest for two basic blocks of the same thread group in the `bodytrack` benchmark. As shown in the figure, the total execution count of the first block does not change as the number of threads in the group increases, while the total execution count of the second block increases. But for both blocks, the total execution counts are evenly distributed among different threads in the group.

3.4 Test Program Generation

FADATest adopts the C language for the test program as it is an efficient system language and it is quite straightforward to integrate the assembly code of basic blocks into the test program. To allow users to easily scale up/down the execution time of the test program, FADATest generates each test program with a *scaling factor* η, which is used to calculate the actual execution counter of a basic block BB for each thread T:

$$ActualExcutionCounter(BB, T) = \frac{HotnessCounter(BB, T)}{\eta} \quad (1)$$

Here, the hotness counter of BB for T is calculated using the distribution pattern derived from the previous analysis. The value of η can be tuned each time when the test program is used to test DBT performance on a new hardware platform. In case the actual execution count of a basic block becomes zero, i.e., η is higher than its hotness counter, the block will be skipped in the testing.

For a multi-threaded test program, a simple execution mechanism is to launch all threads at the beginning of the program. However, this may introduce potential inaccurate testing results. The reason is that the execution of a multi-threaded program usually interleaves multi-threaded execution phases with single-threaded execution phases. Hence, the total execution time of the program should combine the execution times of all phases. In other words, the test program generated by FADATest should preserve both multi-threaded phases and single-threaded phases in the seed benchmark program. Therefore, FADATest integrates the profiled thread activity information into the generated test program. In particular, the relative execution points of thread creation and termination events are the same as those in the seed benchmark program. This way, the multi-threaded test program generated by FADATest can achieve more accurate performance testing results.

Algorithm 2: Test Program Generation

Input: *HotnessList* - The list of block hotness arrays
ThreadGroupList - The specification of thread groups
ThreadActivityList - The specification of thread activities
Output: *TCode* - The generated source code of the test program

```
1  TCode ← NULL;
2  foreach thread group id TGID in ThreadGroupList do
3      TCode ← TCode + GenCodeThreadFuncHead(TGID);
4      foreach basic block BB in HotnessList do
5          TCode ← TCode + GenPayloadBB(BB, TGID);
6      end
7      TCode ← TCode + GenCodeThreadFuncTail(TGID);
8  end
9  TCode ← TCode + GenCodeCtrlFuncHead();
10 foreach thread group id TGID in ThreadActivityList do
11     TCode ← TCode + GenCodeThreadActivity(TGID);
12 end
13 TCode ← TCode + GenCodeCtrlFuncTail(TGID);
14 return TCode;
```

Algorithm 2 shows the process of generating a test program in FADATest. An example of the generated test program is illustrated in Figure 6. The major components of the test program include several functions: `init`, `run_thread_0`, `fini`, and `main`. The `main` function serves as the driver of the test program. The `init` and `fini` functions allocate and deallocate memory resources required to run the test program, respectively. FADATest analyzes the instructions in each basic block to determine the exact memory resource required by the block (see Section 4 for more details). The `run_thread_0` function is the test function, which contains the assembly code of basic blocks. Here, "0" is the index of the thread group. That means, FADATest generates a test function for each thread group, rather than each thread. This design choice allows FADATest to limit the size of the test program and reduce the additional pressure on the code cache of the target DBT system, as all executed blocks will be translated and stored in the code cache.

4 IMPLEMENTATION

We have implemented a prototype of FADATest with the support of two mainstream ISAs on the market: x86-64 and AArch64. The execution profiler is implemented at the binary code level, using the DBT system QEMU [3]. Note that FADATest is *not* tied to any specific DBT system, and, in fact, the generated test programs can be used to test various DBT systems. The distribution pattern

```
#include ...
#define ...
struct timeval *tv_begin, *tv_end; unsigned char *ptr_mem_base;
void run_thread_0(unsigned int *percent);
... //other declarations
void init(void) {
  ptr_mem_base = alignment16(malloc(1858758 * sizeof(uint64_t));
  ptr_double_float = malloc(sizeof(double));
  tv_begin = calloc(sizeof(struct timeval));
  ... //more initialization
}
void run_thread_0(unsigned int *percent) {
  //executes basic blocks according to hotness and scaling factor
  for(long i=0; i< *percent; i++) {
    asm volatile(
      "movq %[input_mem_base], %%r15\n\t" //prepares memory
      "movq $1363, %%r10\n\t" //repeats 1363 times per iteration
      "loop_0_0: movl 0x2f(%%r15), %%edx\n\t"
      "cmpl %%ebx, %%edx\n\t"
      "jne jmp_hit_0_0\n\t" //branches to a provided target
      "test %%r15, %%r15\n\t"
      "jmp_hit_0_0: dec %%r10\n\t"
      "test %%r10, %%r10\n\t"
      "jnz loop_0_0\n\t"
      :
      :[input_mem_base] "m" (ptr_mem_base)
      :"rdx","r10","r15"
    );  ... //more basic blocks
}}  ... //more threads
void fini(void) {  ... /*releases memory*/}
int main(int argc, void *argv) {
  init();
  /*runs threads according to thread activity profiling*/
  run_thread_func_0(80); //executes first 80% of main thread.
  for(int i = 1;i < n_thread;i++){ //runs threads
    pthread_create(&(tid[i-1]), NULL, \
        (void*)run_thread_func_1, &percent);
  }
  for(int i = 1;i < n_thread;i++){
    pthread_join(&(tid[i-1]), NULL);
  }
  run_thread_func_0(20); //executes remaining 20% of main thread.
  fini();return 0;
}
```

Figure 6: A test program generated by FADATest. The code is simplified for demonstration.

analyzer and the test program generator are primarily developed in Python (~ 2.4K LoC). The major obstacle we encountered during the implementation was how to legitimately execute the instructions in the basic blocks included in the generated test programs. We next elaborate our solutions for each type of instructions.

Memory Access Instructions. To correctly execute a memory access instruction, we need to pass an accessible memory address to the instruction. In general, the memory address accessed by an instruction is encoded through four elements and calculated based on the formula: $base + index \times scale + offset$. Here, *base* is required while *index* and $offset$ are optional. *scale* is a constant of 1, 2, 4, or 8. Besides, *base* is typically provided through a register. Thus, our implementation allocates a memory region in advance and passes appropriate values to related registers so that the calculated address falls into the region. Note that memory access instructions in the same block can share one memory region to reduce the overhead caused by memory allocation and data preparation.

Direct Branch Instructions. Since the generated test program does not recover the control flow of the seed program, we need to update the target of a branch instruction. Otherwise, the execution of the branch instruction may experience an error if the branch target is unreachable. Hence, we revise the target of a branch instruction to the instruction immediately following the branch instruction. Furthermore, several instructions are inserted between the branch and the target in case the branch is a conditional branch.

Indirect Branch Instructions. Our implementation also needs to take care of indirect branches. This is achieved by first placing a reachable code address into a memory location and then replacing the operand of an indirect branch with the address of the location. This allows the indirect branch to be executed correctly.

Call/Ret Instructions. In our implementation, call/ret instructions are not executed directly. Instead, a call instruction is decomposed into two instructions. The first one saves the return address and the second one branches to the call target, which is replaced with a reachable code address. Like most DBT systems, ret instructions are handled in the same way as indirect branches.

Floating-Point Instructions. Inappropriate floating-point operations may produce floating-point exceptions, e.g., division by zero. Given that a program will be terminated once a floating-point exception is triggered, we need to avoid such operations. To this end, our implementation analyzes all floating-point instructions in a basic block and carefully prepares the right floating-point data for them so that no exception will be triggered.

SIMD Instructions. Similar to floating-point instructions, our implementation feeds SIMD instructions with appropriate floating-point values to avoid potential floating-point exceptions. In addition, SIMD instructions generally access multiple memory items simultaneously. This requires our implementation to pay special attention to calculate the actual size of the required memory and provide aligned addresses for SIMD instructions.

System Calls. If a block contains a system call, our implementation firstly analyzes the block to determine whether the system call number is defined. In case the instruction that defines the system call number is not found in this block, an additional instruction will be inserted before the system call instruction to define the system call number. We use the `getpid` system call because it does not require a parameter and also has no impact on the system state of the host platform. For simplicity, FADATest does not ensure that the system call parameters are prepared correctly. This is reasonable because a system call may fail even in a normal execution.

ISA-Specific Instructions. Some ISA-specific instructions may have special semantics and access implicit operands. For example, the x86-64 `CPUID` instruction takes as input the value in `RAX` and returns the CPU features to `RAX`, `RBX`, and `RCX`. Our implementation ensures the correct execution of such an instruction in a basic block by analyzing the instructions before it and inserting additional instructions to initialize the input value if necessary.

5 EXPERIMENTAL RESULTS

In this section, we evaluate FADATest. The evaluation aims to answer the following research questions: i) How efficient are the test programs generated by FADATest when used to test DBT performance? ii) Can the test programs generated by FADATest uncover the same performance regression issues of a DBT system as the original seed benchmark programs? iii) How efficient is the test program generation process? The data and source code are available at https://github.com/fadatestdbt/fadatest.git.

Experimental Setup. In our evaluation, we use FADATest to generate test programs from benchmark programs in two standard

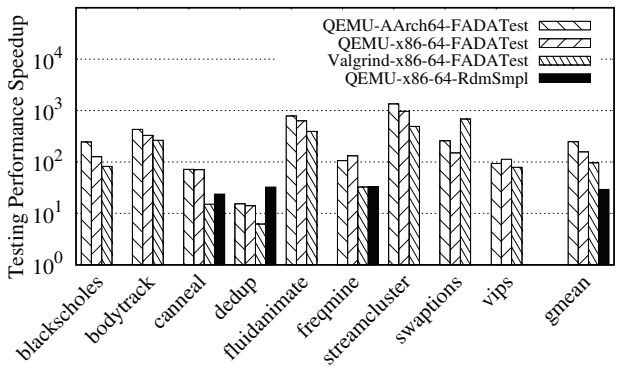

Figure 7: Testing performance speedup achieved by the test programs generated by FADATest and random sampling (RdmSmpl). The performance baseline is the testing times of original benchmarks. Higher is better.

benchmark suites: SPEC CPU 2017 and PARSEC (version 3.0), which have been extensively used to test the performance of DBT systems. For each SPEC CPU 2017 benchmark, FADATest only needs to run it once to collect the execution profile, as SPEC CPU 2017 benchmarks are single-threaded. For PARSEC benchmarks, FADATest runs them with 1, 2, and 4 threads to collect multiple execution profiles for distribution pattern analysis, but tests them with 8 threads to show the capability of FADATest to scale up the number of threads. The generated test programs are employed to test the performance of two representative DBT systems: QEMU and Valgrind, with 8 and 5 historical versions, respectively.

Apart from comparing with the testing results of the original benchmark programs, we also implement a random sampling mechanism and compare it comprehensively with FADATest. However, if the sampling frequency is too high, it will significantly prolong the sampling process. Moreover, it needs extra storage to save the sampled basic blocks. Hence, we adopt a sampling interval of 10,000 basic blocks and limit the entire sampling process to 10 hours for a benchmark program. Even with such a long time, we still can only collect blocks for three PARSEC benchmarks. This also demonstrates the impracticability of the random sampling approach.

To demonstrate the adaptability of the generated test programs, the evaluation covers two hardware platforms, powered by x86-64 and AArch64, respectively. The x86-64 platform is equipped with an Intel i9-9900 CPU at 3.1GHz and 32GB main memory, while the AArch64 platform has a Rockchip RK3399 CPU at 2.0GHz with 4GB memory. On the x86-64 platform, QEMU takes as input AArch64 guest binaries, while on the AArch64 platform, it runs x86-64 guest binaries. However, QEMU fails to run x86-64 binaries of original SPEC CPU 2017 benchmarks on the AArch64 platform. Also, there is no data for Valgrind on AArch64 due to the significantly long execution time of Valgrind. It takes less than 3 minutes to manually tune the scaling factor η for a test program on one platform. The two platforms are occupied exclusively by our evaluation to reduce any potential impacts of random factors. In addition, we run each test 10 times and use the average value of them as the final result.

5.1 Testing Efficiency

Figure 7 shows the testing performance speedup achieved by the test programs generated by FADATest when used for testing the performance of QEMU and Valgrind on the two hardware platforms. The performance baseline is the testing time of the original benchmark programs. The results of random sampling are also included for reference. Since there are multiple versions of QEMU and Valgrind, we measure the testing performance speedup of every version for each benchmark and use the geometric mean as the final speedup result of the benchmark. As shown in the figure, the testing performance is significantly improved by the test programs generated by FADATest compared to the original benchmarks, with an average speedup of 248× for QEMU-AArch64, 156× for QEMU-x86-64, and 96× for Valgrind-x86-64. It is not a surprise that random sampling can also achieve testing performance speedup, as it skips the execution of many basic blocks. On average, it only takes around 35 seconds to complete the testing of a test program generated by FADATest. This shows the capability of FADATest to offer a high testing efficiency for DBT performance testing.

5.2 Testing Effectiveness

QEMU. Figure 8 shows the performance testing results of different versions of QEMU on the x86-64 platform using each original benchmark program in SPEC CPU 2017 and PARSEC and the corresponding test program generated by FADATest. The testing times are normalized to the version 2.9.1, so both of the two lines start from 1. Due to the space limitation, we omit the testing results of QEMU-AArch64, which are very similar to QEMU-x86-64.

From Figure 8, we can make two observations. First, the performance of QEMU-x86-64 changes frequently across different versions. For some benchmarks, such as `perlbench` and `leela`, there is a clear performance regression from the version 3.0.0 to the version 4.0.1. For some other benchmarks, e.g., `canneal` and `vips`, the performance regression starts earlier, i.e., from the version 2.12.1 to the version 3.0.0. This phenomenon again suggests that it is highly necessary to conduct performance regression testing for DBT systems. Moreover, a comprehensive test suite is required to expose the performance regressions as early as possible. Second, the testing results of the test programs generated by FADATest highly match with those of the original benchmark programs. In particular, the performance changes uncovered by testing the original benchmark programs can also be captured by testing the generated test programs. This shows the effectiveness of the test programs generated by FADATest for DBT performance regression testing.

Valgrind. Figure 9 shows the performance testing results of Valgrind using the original benchmark programs and the corresponding test programs generated by FADATest. Similarly, the testing times of the version 3.14.0 are used as the performance baseline. As shown in the figure, Valgrind also suffers from performance regressions, e.g., from the version 3.17.0 to the version 3.18.0 for `perlbench`. Furthermore, the testing results of the test programs generated by FADATest perfectly match with those of the original benchmark programs. This further demonstrates the effectiveness and the portability of FADATest for DBT performance regression testing across different DBT systems.

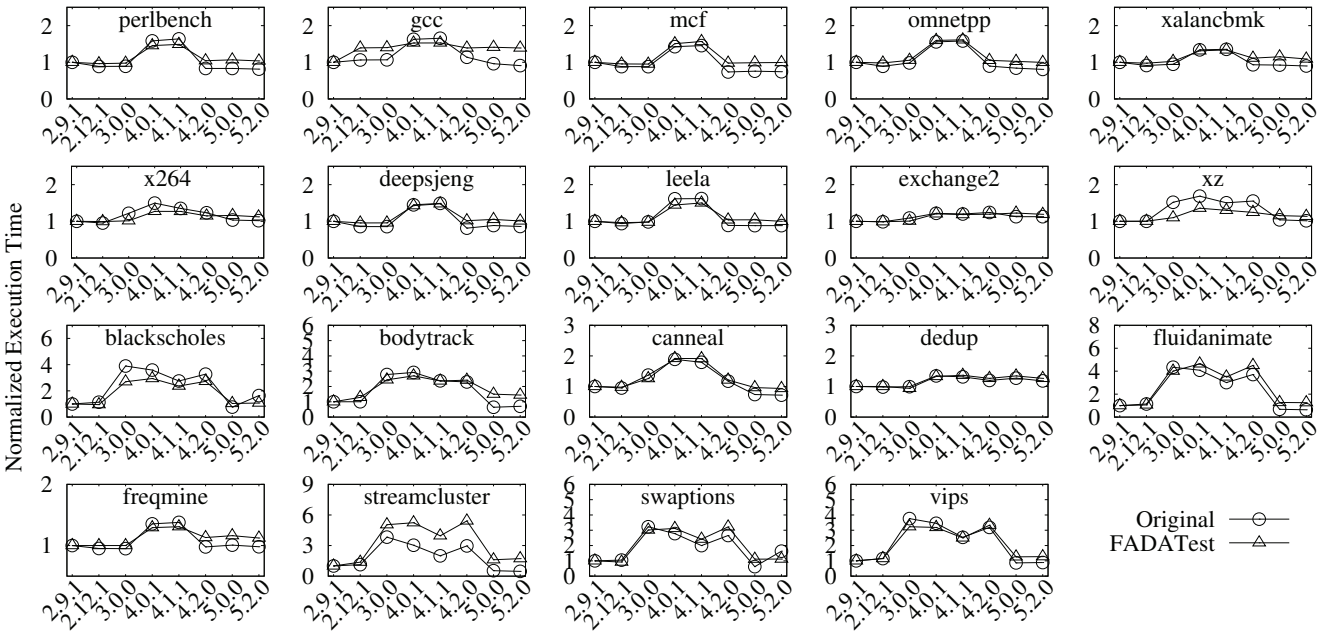

Figure 8: Testing effectiveness of the test programs generated by FADATest with different versions of QEMU-AArch64.

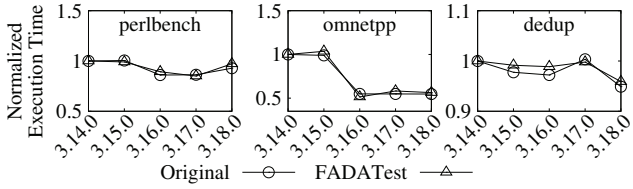

Figure 9: Testing effectiveness of the test programs generated by FADATest with different versions of Valgrind.

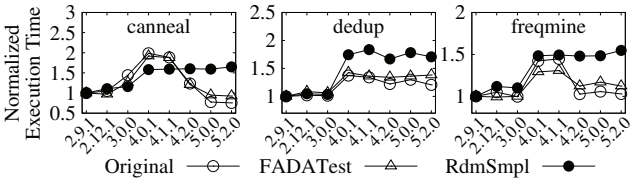

Figure 10: Test effectiveness of the test programs generated by FADATest and the random sampling mechanism (RdmSmpl) with different versions of QEMU.

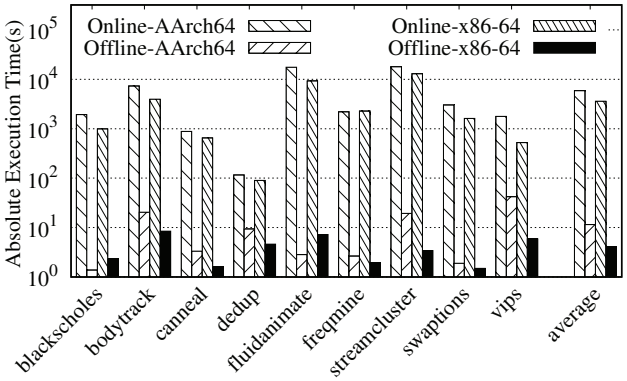

Figure 11: Absolute execution times (in seconds) of the two stages of the test program generation process in FADATest: online profiling and offline analysis and generation.

Comparing with Random Sampling. We next compare the testing results of the test programs generated by FADATest and those generated using the random sampling approach. We also include the testing results of original benchmarks as reference. Figure 10 shows the results of some benchmarks for QEMU. From the figure, we can clearly see that random sampling cannot achieve testing results that align with those of original benchmark programs. For example, random sampling fails to uncover the performance fluctuation after the version 4.0.1 for the benchmark `canneal`. The reason is that the test programs generated by random sampling cannot preserve the *program-specific* runtime characteristics of original benchmark programs. In contrast, FADATest generates test programs based on the profiled block hotness and thread activity information, which enables it to achieve a testing effectiveness similar to original benchmark programs.

5.3 Test Program Generation Process Study

Performance Overhead. There are two major stages in FADATest to generate a test program from a benchmark program: online profiling and offline analysis and generation. Figure 11 shows the absolute

Table 2: Sizes of log files generated by FADATest (in bytes). "A64:" AArch64; "x64:" x86-64.

Benchmark	A64	x64	Benchmark	A64	x64
blackscholes	327K	510K	freqmine	437K	638K
bodytrack	1.9M	2.0M	streamcluster	1.9M	2.9M
canneal	578K	826K	swaptions	377K	592K
dedup	1.3M	1.8M	vips	2.8M	2.8M
fluidanimate	510K	759K	**average**	**1.5M**	**1.2M**

Table 3: Comparisons of the test program generation process between FADATest and random sampling (RdmSmpl).

		canneal	dedup	freqmine
Online Profiling	FADATest	882s	116s	2188s
	RdmSmpl	5157s	1453s	33465s
Offline Analysis	FADATest	1.6s	4.63s	1.95s
	RdmSmpl	69.79s	69.94s	173.41s
Log Size	FADATest	578KB	1.3MB	437KB
	RdmSmpl	262MB	78MB	1.8GB

times required by the two stages for each benchmark and the two ISAs. We omit the results of SPEC CPU 2017 benchmarks due to the space limitation. For online profiling, the time is closely correlated to the original execution time of each benchmark program. On average, the execution time of online profiling in FADATest, compared to the original benchmark execution time, is around 2.4×. We think this performance overhead is reasonable and also acceptable, as the test programs only need to be generated once. Compared to the online profiling stage, the offline analysis and generation stage takes much less time. As shown in the figure, for all benchmarks, the generation time is less than 30 seconds. This demonstrates the high efficiency of the analysis and generation process in FADATest.

Storage Cost. We also study the cost of storing the log files generated by FADATest during the online profiling stage. Table 2 shows the detailed sizes of the log files generated for each benchmark and the two ISAs. As shown in the table, for all benchmarks, the size is less than 3MB. On average, it only costs around 1.5MB for a benchmark and one ISA. We believe this storage cost is acceptable in practice, given that the capacity of a typical storage device is up to several gigabytes even on embedded devices.

Comparing with Random Sampling. We further compare the test program generation process of FADATest with the random sampling approach. Table 3 shows the results. As shown in the table, it takes a much longer time for random sampling to finish the generation process. This is because the sampled blocks need to be saved during the sampling process. Moreover, the sizes of the log files generated by random sampling are much larger. These results show the impracticability of the random sampling approach.

6 CASE STUDY

In this section, we show two representative case studies on QEMU to demonstrate the practicability and necessity of conducting performance regression testing through FADATest during the daily development of a DBT system.

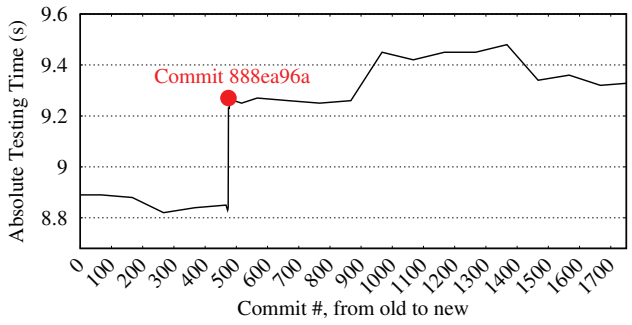

Figure 12: Performance regression by a single commit.

A performance regression caused by a performance bug. In this case study, we first observed a performance regression issue of the `perlbench` benchmark under the *reference* input between two commits: `eabb7b91` (04/21/2016) and `757e725b` (01/26/2016), which include 1764 other commits between them. That means, to figure out which commit(s) lead to the regression, it is necessary to conduct comprehensive performance testing on individual commits. Given the large number of the commits, it will be very time-consuming to conduct such testing using the original benchmark because of the long execution time of QEMU. Note that we didn't observe a performance difference if the *test* input of `perlbench` is used for the testing. Also, it may take more than 300 hours to test the commits with the *train* input. In contrast, with the FADATest generated test program, we complete the performance testing of the 1764 commits in around 13 hours. Figure 12 shows the testing result.

From the figure, we can clearly see that this performance regression issue is caused by a single commit, i.e., `888ea96a` (02/16/2016). Further investigation shows that this commit essentially removed the manually enforced `__always_inline__` attribute because the developer overoptimistically assumes that "*compilers have improved … and we are probably better off trusting the compiler rather than trying to force its hand.*" Unfortunately, however, the compiler is not as smart as human developers to make an optimal decision on whether a function should be inlined or not. On the other hand, inlining is an important compiler optimization that can enable many other optimizations, e.g., dead code elimination and constant propagation. That is why this commit leads to a performance regression.

A performance regression caused by extended functionalities. This case study centers on the performance regression of QEMU for the `perlbench` benchmark between the two release versions: 4.0.1 (10/17/2019) and 3.0.0 (08/14/2018), as shown in Figure 8. There are 5655 commits between these two versions. By conducting performance testing using the test program generated by FADATest, two commits that caused the regression are identified in around 42 hours: `f7b78602` (01/29/2019) and `c47eaf9f` (02/05/2019). The first commit added additional code to check whether a translation block is valid for different clusters. Since the code was added on the critical path of the translation process, it introduced a substantial performance slowdown. The second commit mainly added the support of ARM pointer authentication instructions, which were simply ignored before this commit. As a result, it would inevitably decrease the performance if the guest application has such instructions.

Summary. The above case studies clearly demonstrate the benefit of the high testing efficiency offered by FADATest on identifying problematic commits that cause performance regressions. At the same time, it shows the necessity of conducting performance regression testing during the daily development of a DBT system. Here, we would like to emphasize that the high testing efficiency of FADATest also provides a performance-centric view for developers on each commit. This is important even though the commit is to add a new feature to the DBT system and expects some performance slowdown, as the testing will tell developers the exact performance slowdown and allows them to immediately remedy the performance loss if necessary.

7 RELATED WORK

Performance Regression Testing. A considerable amount of research work has been conducted to enhance both effectiveness and efficiency of performance regression testing. Some research work leverages automated workload generation [4, 10] to produce proper input data sets for performance regression testing. Some research work creates selective strategies to attain better testing efficiency [19, 30, 31]. Bagherzadeh et al. analyze the performance results of system calls in historic versions of the Linux kernel [2]. WISE presents an automated test generation techniques to find performance bugs [7]. The research work in [18] proposes to utilize performance counters to detect performance regressions. To obtain a fast testing report, some research work studies test case prioritization approaches [20, 26]. PerfScope [14] improves the testing efficiency by conducting performance risk analysis for prioritization. To test the performance of multi-threaded programs, SpeedGun generates multiple threads from a single-threaded application [28]. PerfImpact sends the same input to two releases and automatically mines the corresponding execution traces to rank the impacts of code changes [24]. FOREPOST detects performance issues by adopting machine learning techniques to select input data to cover computationally intensive program paths [25].

Though performance regression testing has been explored comprehensively in common software systems, it is still very challenging to directly apply existing performance regression techniques to DBT systems. There are several reasons. First, different from common software systems, DBT systems take as input executable binaries, which make it quite difficult to automatically and flexibly generate representative inputs to conduct performance regression testing of DBT systems. Second, due to the low efficiency of DBT systems, it takes extremely long time to complete the testing of standard benchmark programs on DBT systems. Third, the diverse hardware platforms of a DBT system make it hard to use the same benchmark programs to test the performance of the DBT system on different hardware platforms, especially when the platforms have significantly different computing power. To overcome these challenges, FADATest generates test programs from real benchmark programs based on the execution characteristics of the benchmarks. The generated test programs can achieve efficient and adaptive performance regression testing of DBT systems.

Guest Program Generation for DBT Systems. PerfDBT [39] attempts to generate guest programs to test the performance of a DBT system. However, PerfDBT has several fundamental limitations when using the generated guest programs for DBT performance regression testing. First, it is designed specifically for x86-64 platforms, and thus cannot be applied to DBT systems running on a different hardware platform. Second, multi-threaded benchmark programs are not supported by PerfDBT. As a consequence, the multi-threading performance of the target DBT system cannot be tested. Third, PerfDBT assigns too many memory objects in the generated guest programs, which may introduce a negative impact on the performance testing results. In contrast, FADATest does not have these limitations. It is designed specifically for efficient and adaptive performance regression testing of DBT systems.

Code Generation for Hardware Simulators. To generate input programs for simulators, SimPoint samples instructions during the execution of a program [12]. The sampling policy is created based on the phase information of the original program. Though the selected instructions can be used to test architecture simulators, they are not suitable for DBT systems. This is because DBT systems are typically developed to run real-world applications, instead of small pieces of instructions. Moreover, SimPoint does not support multi-threaded programs. That means the instructions extracted by SimPoint cannot recover the thread behaviors of the original programs. Compared to the instructions extracted by SimPoint, the test programs generated by FADATest are more appropriate for testing the performance of DBT systems, as they can preserve the performance characteristics of original programs.

8 CONCLUSION

Although DBT is a key enabling technology, developing and maintaining a real-world DBT system is not easy. An important method to constantly maintain the performance efficiency of a DBT system is to frequently conduct performance regression testing. However, applying performance regression testing to DBT systems suffers from several practical challenges, such as the natural differences between DBT systems and regular software systems, the limited availability of test programs, and various hardware platforms targeted by a DBT system. This paper presents FADATest to address these challenges. FADATest employs several novel techniques to generate adaptable test programs, which can be used to achieve efficient and adaptive performance regression testing of DBT systems. We also implement a prototype of FADATest and use it to generate test programs from benchmark programs in SPEC CPU 2017 and PARSEC. Experimental results show that the test programs can uncover the same performance regression issues across different versions of QEMU and Valgrind as the original benchmark programs and the testing efficiency is improved significantly. We anticipate that FADATest will be integrated into the development cycles of existing DBT systems by automatically launching the performance regression testing before each code commit.

ACKNOWLEDGMENTS

We are very grateful to the reviewers for their valuable feedback and comments. This work was done while Jin Wu was visiting the University of Georgia. This work was supported in part by the M. G. Michael Award funded by the Franklin College of Arts and Sciences at the University of Georgia and a faculty startup funding of the University of Georgia.

REFERENCES

[1] Adam Henson. 2019. Automatic Website Performance Regression Testing. https://www.freecodecamp.org/news/automatic-website-performance-regression-testing-4e30e6bf5cd.

[2] Mojtaba Bagherzadeh, Nafiseh Kahani, Cor-Paul Bezemer, Ahmed E Hassan, Juergen Dingel, and James R Cordy. 2018. Analyzing a decade of Linux system calls. *Empirical Software Engineering* 23, 3 (2018), 1519–1551.

[3] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATC '05)*. USENIX Association, USA, 41.

[4] Abderrahmane Benbachir, Isnaldo Francisco De Melo, Michel Dagenais, and Bram Adams. 2017. Automated Performance Deviation Detection across Software Versions Releases. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 450–457.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. Association for Computing Machinery, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[6] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 463–469.

[7] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. 2009. WISE: Automated test generation for worst-case complexity. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 463–473.

[8] Jie Chen, Dongjin Yu, Haiyang Hu, Zhongjin Li, and Hua Hu. 2019. Analyzing Performance-Aware Code Changes in Software Development Process. In *Proceedings of the 27th International Conference on Program Comprehension* (Montreal, Quebec, Canada) *(ICPC '19)*. IEEE Press, 300–310. https://doi.org/10.1109/ICPC.2019.00049

[9] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2020. WEIZZ: Automatic Grey-Box Fuzzing for Structured Binary Formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3395363.3397372

[10] King Chun Foo, Zhen Ming Jiang, Bram Adams, Ahmed E Hassan, Ying Zou, and Parminder Flora. 2015. An industrial case study on the automated detection of performance regressions in heterogeneous environments. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 159–168.

[11] Gunnar Morling. 2020. Towards Continuous Performance Regression Testing. https://www.morling.dev/blog/towards-continuous-performance-regression-testing.

[12] Greg Hamerly, Erez Perelman, and Brad Calder. 2004. How to use simpoint to pick simulation points. *ACM SIGMETRICS Performance Evaluation Review* 31, 4 (2004), 25–30.

[13] Christoph Heger, Jens Happe, and Roozbeh Farahbod. 2013. Automated Root Cause Isolation of Performance Regressions during Software Development. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering* (Prague, Czech Republic) *(ICPE '13)*. Association for Computing Machinery, New York, NY, USA, 27–38. https://doi.org/10.1145/2479871.2479879

[14] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. 2014. Performance Regression Testing Target Prioritization via Performance Risk Analysis. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE '14)*. Association for Computing Machinery, New York, NY, USA, 60–71. https://doi.org/10.1145/2568225.2568232

[15] Jinhu Jiang, Rongchao Dong, Zhongjun Zhou, Changheng Song, Wenwen Wang, Pen-Chung Yew, and Weihua Zhang. 2020. More with Less – Deriving More Translation Rules with Less Training Data for DBTs Using Parameterization. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 415–426. https://doi.org/10.1109/MICRO50266.2020.00043

[16] Ivo Jimenez, Noah Watkins, Michael Sevilla, Jay Lofstead, and Carlos Maltzahn. 2018. Quiho: Automated Performance Regression Testing Using Inferred Resource Utilization Profiles. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering* (Berlin, Germany) *(ICPE '18)*. Association for Computing Machinery, New York, NY, USA, 273–284. https://doi.org/10.1145/3184407.3184422

[17] Joel Beales and Jeffrey Dunn. 2018. MobileLab: Highly accurate testing to prevent mobile performance regressions. https://engineering.fb.com/2018/10/19/android/mobilelab.

[18] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. 2005. Automated detection of performance regressions: The Mono experience. In *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 183–190.

[19] Rafaqut Kazmi, Dayang NA Jawawi, Radziah Mohamad, and Imran Ghani. 2017. Effective regression test case selection: A systematic literature review. *ACM Computing Surveys (CSUR)* 50, 2 (2017), 1–32.

[20] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang NA Jawawi, and Rooster Tumeng. 2018. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology* 93 (2018), 74–93.

[21] Stephen Kyle, Igor Böhm, Björn Franke, Hugh Leather, and Nigel Topham. 2012. Efficiently Parallelizing Instruction Set Simulation of Embedded Multi-Core Processors Using Region-Based Just-in-Time Dynamic Binary Translation. In *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems* (Beijing, China) *(LCTES '12)*. Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/2248418.2248422

[22] Philipp Leitner and Cor-Paul Bezemer. 2017. An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (L'Aquila, Italy) *(ICPE '17)*. Association for Computing Machinery, New York, NY, USA, 373–384. https://doi.org/10.1145/3030207.3030213

[23] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-State Based Binary Fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 627–637. https://doi.org/10.1145/3106237.3106295

[24] Qi Luo, Denys Poshyvanyk, and Mark Grechanik. 2016. Mining Performance Regression Inducing Code Changes in Evolving Software *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/2901739.2901765

[25] Qi Luo, Denys Poshyvanyk, Aswathy Nair, and Mark Grechanik. 2016. FOREPOST: A Tool for Detecting Performance Problems with Feedback-Driven Learning Software Testing. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 593–596. https://doi.org/10.1145/2889160.2889164

[26] Shaikh Mostafa, Xiaoyin Wang, and Tao Xie. 2017. PerfRanker: Prioritization of Performance Regression Tests for Collection-Intensive Software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/3092703.3092725

[27] Thanh H.D. Nguyen, Bram Adams, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2012. Automated Detection of Performance Regressions Using Statistical Process Control Techniques. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering* (Boston, Massachusetts, USA) *(ICPE '12)*. Association for Computing Machinery, New York, NY, USA, 299–310. https://doi.org/10.1145/2188286.2188344

[28] Michael Pradel, Markus Huggler, and Thomas R Gross. 2014. Performance regression testing of concurrent classes. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 13–25.

[29] QEMU wiki. 2021. Projects using the QEMU code. https://wiki.qemu.org/Links#Projects_using_the_QEMU_code.

[30] David Georg Reichelt and Stefan Kühne. 2018. How to Detect Performance Changes in Software History: Performance Analysis of Software System Versions. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering* (Berlin, Germany) *(ICPE '18)*. Association for Computing Machinery, New York, NY, USA, 183–188. https://doi.org/10.1145/3185768.3186404

[31] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing regression test selection techniques. *IEEE Transactions on software engineering* 22, 8 (1996), 529–551.

[32] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 475–486. https://doi.org/10.1145/2485922.2485963

[33] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security* (Hyderabad, India) *(ICISS '08)*. Springer-Verlag, Berlin, Heidelberg, 1–25. https://doi.org/10.1007/978-3-540-89862-7_1

[34] Standard Performance Evaluation Corporation. 2020. SPEC CPU 2017. https://www.spec.org/cpu2017.

[35] Amanda Swearngin, Myra B. Cohen, Bonnie E. John, and Rachel K. E. Bellamy. 2013. Human Performance Regression Testing. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 152–161.

[36] Wenwen Wang. 2021. Helper Function Inlining in Dynamic Binary Translation. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*. Association for Computing Machinery, New York, NY, USA, 107–118. https://doi.org/10.1145/3446804.3446851

[37] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. 2020. Efficient and Scalable Cross-ISA Virtualization of Hardware Transactional Memory. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation*

and Optimization. Association for Computing Machinery, New York, NY, USA, 107–120. https://doi.org/10.1145/3368826.3377919

[38] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, Stephen McCamant, Youfeng Wu, and Jayaram Bobba. 2017. Enabling Cross-ISA Offloading for COTS Binaries. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services* (Niagara Falls, New York, USA) *(MobiSys '17)*. Association for Computing Machinery, New York, NY, USA, 319–331. https://doi.org/10.1145/3081333.3081337

[39] Jin Wu, Jian Dong, Ruili Fang, Wenwen Wang, and Decheng Zuo. 2020. PerfDBT: Efficient Performance Regression Testing of Dynamic Binary Translation. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*. 389–392. https://doi.org/10.1109/ICCD50377.2020.00071

[40] Jin Wu, Jian Dong, Ruili Fang, Ziyi Zhao, Xiaoli Gong, Wenwen Wang, and Decheng Zuo. 2021. Effective Exploitation of SIMD Resources in Cross-ISA Virtualization. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Virtual, USA) *(VEE 2021)*. Association for Computing Machinery, New York, NY, USA, 84–97. https://doi.org/10.1145/3453933.3454016

[41] Ziyi Zhao, Zhang Jiang, Ying Chen, Xiaoli Gong, Wenwen Wang, and Pen-Chung Yew. 2021. Enhancing Atomic Instruction Emulation for Cross-ISA Dynamic Binary Translation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 351–362. https://doi.org/10.1109/CGO51591.2021.9370312

[42] Ziyi Zhao, Zhang Jiang, Ximing Liu, Xiaoli Gong, Wenwen Wang, and Pen-Chung Yew. 2020. DQEMU: A Scalable Emulator with Retargetable DBT on Distributed Platforms. In *49th International Conference on Parallel Processing - ICPP* (Edmonton, AB, Canada) *(ICPP '20)*. Association for Computing Machinery, New York, NY, USA, Article 7, 11 pages. https://doi.org/10.1145/3404397.3404403