Improving Dynamically-Generated Code Performance on Dynamic Binary Translators

Wenwen Wang
University of Minnesota, Twin Cities

Jiacheng Wu
College of Software, Nankai University

Xiaoli Gong*
College of Computer and Control Engineering, Nankai University
State Key Laboratory of Computer Architecture, ICT, CAS

Tao Li
College of Computer and Control Engineering, Nankai University

Pen-Chung Yew
University of Minnesota, Twin Cities

Abstract

The recent transition in the software industry toward dynamically generated code poses a new challenge to existing dynamic binary translation (DBT) systems. A significant re-translation overhead could be introduced due to the maintenance of the consistency between the dynamically-generated guest code and the corresponding translated host code. To address this issue, this paper presents a novel approach to optimize DBT systems for guest applications with dynamically-generated code. The proposed approach can maximize the reuse of previously translated host code to mitigate the re-translation overhead. A prototype based on such an approach has been implemented on an existing DBT system HQEMU. Experimental results on a set of JavaScript applications show that it can achieve a 1.24X performance speedup on average compared to the original HQEMU.

CCS Concepts • **Software and its engineering → Virtual machines**; **Just-in-time compilers**; **Dynamic compilers**; *Runtime environments*; Retargetable compilers; Scripting languages; Software reverse engineering;

Keywords DBT; JIT; Binary Code Matching

ACM Reference Format:
Wenwen Wang, Jiacheng Wu, Xiaoli Gong, Tao Li, and Pen-Chung Yew. 2018. Improving Dynamically-Generated Code Performance on Dynamic Binary Translators. In *VEE '18: 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, March 25, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3186411.3186413

*To whom correspondence should be addressed.

ACM ISBN 978-1-4503-5579-7/18/03...$15.00
https://doi.org/10.1145/3186411.3186413

1 Introduction

Dynamic binary translation (DBT) has been extensively used in many important applications, which include whole-program analysis and application debugging, e.g., Intel Pin [23], DynamoRIO [4], and Valgrind [25], software development, e.g., Android Emulator [33] and QEMU [3], software vulnerability detection and defense, e.g., BAP [5] and BitBlaze [32], and mobile computation offloading [38]. In general, a DBT system dynamically translates/modifies binary code in its original (or *guest*) instruction set architecture (ISA) into binary code in its target (or *host*) ISA, which could be *same as* (e.g., in binary instrumentation) or *different from* the original guest ISA. By executing the translated/modified host binary code, a DBT system can support the emulation of the guest application, enhance its functionalities, or optimize the performance of the guest application on a host machine.

Recently, the widespread deployment of *scripting languages* such as JavaScript, PHP, and C# [1, 2, 6, 11, 26, 29], which are typically used in software rapid prototyping tools, has presented a special challenge to existing DBT systems. Unlike most others, applications written in scripting languages often have short execution time, and are either interpreted or executed through *dynamically-generated code* (DGC) by just-in-time (JIT) compilers on language-level virtual machines such as JavaScript engines [8, 24].

It is interesting to note that both the guest JIT engine and the DBT system employ similar code caching mechanism to amortize the translation overhead and to improve the overall performance. To support such applications on a DBT system, we need to maintain *consistency* between the guest DGC produced by the guest JIT engine and its corresponding dynamically-translated host code by the DBT. This is because such guest DGC could be modified or re-optimized by the guest JIT engine at runtime. However, maintaining such consistency could incur substantial overhead for DBT systems if not handled properly.

Typically, DBT systems have to invalidate the translated host binary each time the corresponding guest binary is modified by the guest JIT. In general, most RISC architectures require an explicit request to flush the instruction cache if the code has been modified through the data cache [18]. However, CISC architectures such as Intel x86 maintain the coherence between the instruction and data caches through hardware, and hence, no explicit request is required when the guest binary is updated or changed by its JIT. This makes it quite challenging for a DBT system to determine whether a write in the guest application is to update the guest binary, or simply a regular write to its data. Instead of instrumenting the guest binary to monitor every write instruction and catch those that update the guest binary, most existing DBT systems rely on setting page access permission to *read-only* for those guest binaries. When any of those pages is changed (or *written*), an access violation (through *page fault*) is triggered and the consistency can be maintained by invalidating the corresponding host binary.

Although the above mechanism allows existing DBT systems to correctly emulate guest applications with DGC, it could suffer very poor performance. In particular, each time a page fault is triggered, *all* host binary corresponding to this page has to be invalidated because we don't know which part of the page is modified. However, considering the size of a regular memory page, e.g., 4KB on ARM and Intel x86, it is often unnecessary to invalidate all host binaries in that page because only a small code snippet is modified. By invalidating the whole page, we need to go through the re-emulation and re-translation for most of the unmodified code, which can incur a lot of overhead.

Based on this observation, InferenceDR [12] selectively instruments guest *write* instructions to obtain the *precise* memory locations of the modified guest binary code and invalidate the corresponding host binary code only. Although such an approach can reduce the re-translation overhead, it still needs to find the *exact* host binary that corresponds to the identified guest binary. It often requires establishing complicated mappings between them, which can be quite challenging especially for guest ISAs with variable-length instructions such as Intel x86.

With further investigations on the guest applications with DGC, we find that not only the size of each update by the guest JIT engine is very small, but also most of the updates are used for the generation of new guest binaries, instead of modifying the existing ones. That means, only a very tiny portion of the generated guest binaries is modified or updated later. Specifically, we find more than 95% of the generated *guest* binaries are not modified during their entire execution in our experiments. This indicates that most of the *host* binaries in an invalidated page can actually be *reused* without the need of re-translation.

Inspired by the above observation, this paper presents a novel approach to optimize DBT systems for applications with DGC. The approach preserves and reuses most of the generated host binary until the corresponding guest binary is modified or changed. It can thus avoid most of the unnecessary re-translation required in previous approaches. Compared to [12], the performance overhead incurred by this approach is negligible because only a simple analysis is needed. A prototype based on such an approach has been implemented on an existing DBT system HQEMU [15], which is an improved QEMU that supports binary traces and uses the LLVM compiler as a backend for code optimization. Experimental results on a set of JavaScript applications show that the proposed approach can achieve a 1.24X performance speedup on average compared to the original HQEMU. Moreover, the performance and memory overhead introduced by the proposed approach is negligible.

In summary, this paper has the following contributions:

- A novel approach is proposed to optimize DBT systems for guest applications with dynamically generated code. It preserves most of the DBT code for reuse, and re-translates only when the corresponding guest code is actually modified.
- A prototype based on such an approach has been implemented on an existing DBT system HQEMU to demonstrate the feasibility and the effectiveness of the proposed approach. Several critical implementation issues are also addressed.
- A number of comprehensive experiments on a set of JavaScript applications are conducted. They show the proposed approach can achieve a 1.24X performance speedup on average with negligible performance and memory overhead.

The rest of this paper is organized as follows. Section 2 presents some background of DBT techniques. Section 3 describes the technical details of the proposed approach. Section 4 presents an implementation based on an existing DBT system, i.e., HQEMU. Section 5 shows the experimental results. Section 6 discusses some related work, and Section 7 concludes the paper.

2 Background of DBT

In general, a DBT system translates guest binary code into host binary code at the granularity of a guest *basic block* (or *block* for short), which is a sequence of *guest* instructions with only one entry and one exit [31]. Note that each *guest block* could be translated into multiple *host blocks*. The translated host blocks are saved into a software-managed *code cache* for further execution.

A *hash table* is employed to establish the mapping between each guest block and its translated host block(s). Each time a guest block needs to be emulated, the hash table is looked up to check whether this block has been translated or not. If yes,

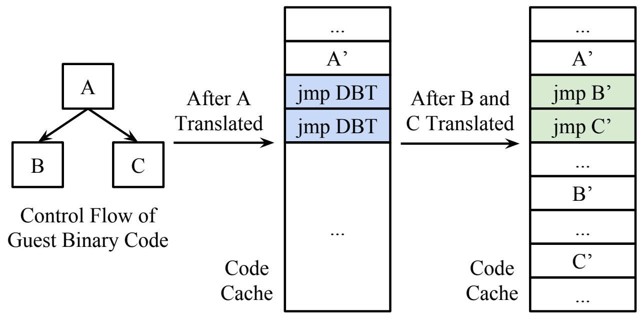


Figure 1. Block chaining of translated host binary code.

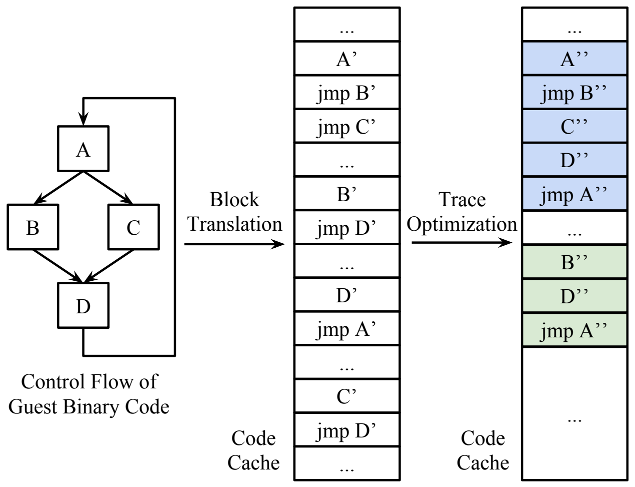


Figure 2. Trace formation to improve performance.

the execution will be redirected to the translated host binary in the code cache. Otherwise, the translator is invoked to translate the guest block. After the translation is finished, the hash table is updated with the newly translated host binary.

Typically, for long-running and loop-intensive guest applications, executing the translated host binary accounts for more than 90% of the execution time [28]. This allows the translation overhead to be substantially amortized. However, for guest applications with few hot code regions or short execution time, such as applications written in scripting languages, the translation overhead could be a significant part of the total execution time [37]. It is important for a DBT system to curb the translation overhead for good performance.

Block Chaining. To switch control between the binary translator and the code cache, a user-level *context switch* is typically required to save and restore the execution contexts (i.e., register file) because each may use the host registers differently. To reduce the overheads caused by repeated context switches, a DBT optimization called *block chaining* is often used to *chain* the translated blocks in the same execution path together [30, 34]. Specifically, a *jump stub* is generated at the exit of a translated block and is initialized to jump back to the binary translator. After its target block is translated, the stub is patched (or changed) to jump to the translated target block instead. Figure 1 illustrates the process of block chaining. As shown in the figure, the two jump stubs generated for block A' are initialized to jump back to the translator, i.e., "jmp DBT". After the guest blocks B and C are translated, the jump stubs are patched to jump to B' and C' instead. In this way, the execution can stay within the code cache and the number of context switches can be reduced.

Trace Formation. Trace formation is a useful optimization for DBT systems. Most existing DBT systems, including DynamoRIO and Intel Pin, use trace formation to enable more optimizations to improve performance. In general, a *trace* is a *hot* execution path in the *guest* binary, which contains several blocks with one entry and multiple exits. The number of guest blocks in a trace is called the *size* of the trace. Typically, two algorithms, i.e., the *next executing tail* (NET) algorithm [7] and the *cyclic-path-based repetition* algorithm [13], are used to form hot traces. The formation process often starts with a hot guest block, which can be identified via dynamic profiling, and terminates when a cyclic execution path (i.e., a loop) is detected either by a backward jump or a repeat of the same guest address. After a guest trace is formed, the blocks in the trace are gathered together and more aggressive optimizations can be applied by the DBT system to translate them into host binary code for a better performance. Obviously, more overhead could incur due to the additional optimizations applied.

Figure 2 shows an example of trace formation. Here, each guest block is translated individually in the block translation phase. During the trace formation, two traces, i.e., ACD and BD, are formed. Note that the guest block D is duplicated in both traces, an optimization called *tail duplication* [14]. This can avoid additional branches to block D and allow more optimization opportunities. After the traces are formed, basic blocks in each trace are re-translated and more optimizations can be applied on traces. A noticeable improvement is that redundant branches between blocks in the same trace can be eliminated, e.g., the branches between A and C, and between B and D.

Maintaining Consistency. To ensure the correctness of the emulation, DBT systems must maintain *consistency* between the guest binary and its translated host binary. As mentioned earlier, for guest architectures that require an explicit request to flush the instruction cache, e.g., MIPS and ARM, it is easier because a DBT system only needs to monitor such flush requests, and flush the host binaries accordingly.

However, for guest architectures that do not need such explicit flush requests, e.g., Intel x86, it is more challenging because it is hard to distinguish between updating a data block vs. updating a binary block. Most existing DBT systems leverage page protection mechanism. A guest code page is set to *read-only* after all of its code blocks are translated into

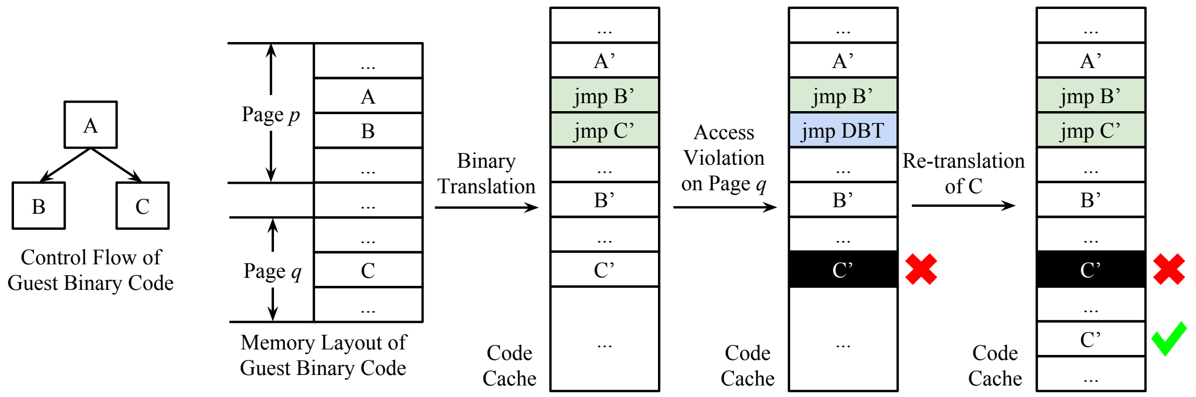

A
B
C
Control Flow of Guest Binary Code
Page p
Page q
...
A
B
C
Memory Layout of Guest Binary Code
Binary Translation
A'
jmp B'
jmp C'
B'
C'
Code Cache
Access Violation on Page q
jmp DBT
Re-translation of C

Figure 3. Consistency maintenance between guest binary code and corresponding host binary code in existing DBT systems.

host binaries. Each time any of its blocks is modified, e.g., by a JIT engine, a *page fault* is triggered and reported to the DBT system. The access permission of the page is reverted to *writable* for the needed update, and changed back to *read-only* after it is done. Given the difficulty of knowing which location in this page actually triggered the page fault using such a mechanism, existing schemes simply invalidate all of the host binaries that correspond to *all* of the guest blocks in this page to guarantee the desired consistency.

Figure 3 illustrates an example of the consistency maintenance process described above. Here, the guest blocks A and B are located in the same guest code page *p*, while the guest block C is in another page *q*. Initially, the translated host blocks A', B', and C' form a chain after the block-chaining process. Also, the pages *p* and *q* are set to *read-only* to detect writes to these pages. If a page fault is triggered on the page *q* with the block C in it, the host block C' will be invalidated to maintain the consistency. It is important to note that all of the traces that contain C as a member block also need to be invalidated because C no longer exists. In addition, all of the chained blocks that contain C' also need to be invalidated. This is because, as shown in the figure, all of the jump stubs targeting C' in the block chains that contain C' will now need to be broken and jump back to DBT.

Figure 3 also shows that C may need to be re-translated, and the new C' can form a chain again with A' and B' in the following execution. If C has indeed been modified, this re-translation is necessary to ensure the correctness of the execution. However, if C is not modified, such re-translation is *redundant* and wasted, especially when multiple traces that contain the block C need to be re-formed and re-translated.

3 Maximizing Reuse of Host Binary Code

To mitigate the significant performance overhead introduced by the re-translation of the dynamically-generated *guest* binary code, this paper tries to reuse the translated *host* binary as much as possible. The approach is inspired by the observation that it is very rare in practice that the originally generated *guest* binary is modified or updated by the guest JIT engine after it is generated.

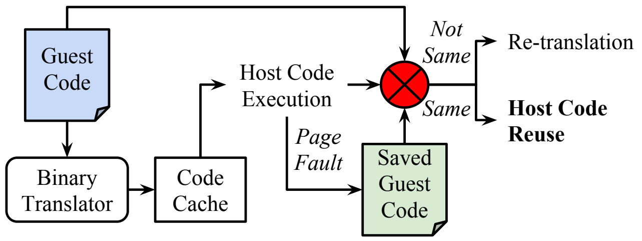


Figure 4. Overview of our host code reuse approach.

Figure 4 shows the overview of the proposed approach. As shown in the figure, each time a page fault is triggered on a guest code page, the blocks in this page are saved for future reference. When a block needs to be re-translated, the previously saved *guest* block (if exists) is compared to see if there is any difference between them. If no difference is found, i.e., "*Same*" shown in the figure, the previously translated *host* binaries could be reused to avoid re-translation. Otherwise, the block will be re-translated to maintain the consistency and the saved guest code is flushed.

3.1 Saving Guest Binary Code

The original guest binary is saved at the granularity of a basic block, which is also the basic translation unit for most of the DBTs. A buffer, called *guest binary code buffer* (GBCB), is allocated to hold the saved guest blocks. The address and the size of a saved guest block are kept in a data structure for future references. To reduce the time to save the guest blocks, SIMD instructions available on the host ISA can be leveraged to speed up the copying and comparing process [19]. Moreover, since only the guest blocks in a faulted page are saved, not the whole guest binary, this can dramatically reduce the required size of GBCB. As shown in Section 5, the memory overhead incurred for maintaining GBCB is negligible for most of the evaluated JavaScript applications.

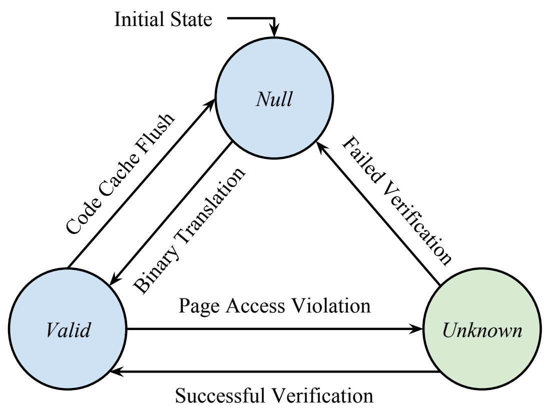


Figure 5. Transitions between different states of host binary code. The events on the lines show the transition conditions.

Algorithm 1: Mark Host Binary Code as *Unknown*

```
Input: The guest code page p, on which a page access
       violation is triggered.
Output: None.
1 foreach basic block block in p do
2     Save_Guest_Code (block);
3     Mark_Host_Code_Unknown (block);
4     Break_Host_Code_Chaining (block);
5     t ← Trace_Contain_Block (block);
6     if t != NULL then
7         foreach trace trace in t do
8             Mark_Host_Code_Unknown (trace);
9             Break_Host_Code_Chaining (trace);
```

It is worth noting that there is no need to save guest binary code for traces because they are composed of guest blocks. To reuse host traces, we only need to record the block information for each trace and the trace information for each block. More details about reusing host binary code of traces are in the next subsection.

3.2 Reusing Host Binary Code

To reuse the previously translated *host* binary, we need to enhance the consistency maintenance mechanism in existing DBT systems. Specifically, each time a page access violation is triggered, we need to keep the translated host binary that corresponds to the guest blocks in this faulted page, instead of flushing it from the code cache. However, a verification step is required before the host binary can be reused, in case that the original guest binary is modified.

In most of the existing DBT systems, the translated host binary for each guest block/trace typically has two states: *null* and *valid*. The *null* state indicates that this block/trace has yet to be translated, and the binary translator needs to be invoked to generate the host binary code. On the other hand, the *valid* state means that the host binary code of this guest block/trace has been generated and can be executed to emulate the guest block/trace. To accommodate the host binary code that can be reused after a successful verification, we introduce a new state: *unknown*, which indicates that its status is not known yet and needs to be verified before it can be reused or flushed.

Figure 5 shows the transitions among these three states. The event on each line denotes the condition required for the state transition. As shown in the figure, the initial state of the host binary for a guest block/trace is *null*. After the block/trace is translated, the state transitions to *valid*. If a page fault is detected, the state then transitions to *unknown*. To reuse the host binary in the unknown state, a verification is needed to guarantee the consistency. If the verification is successful, the state transitions back to valid. Otherwise, the state changes to null and the translator will be invoked to re-translate the guest block/trace.

Algorithm 1 illustrates the process in our approach when a page access violation is detected on a guest code page p. For each block in p, we firstly save the guest binary code of this block (Line 2), and then mark the host binary code of the block as *unknown* (Line 3). In addition, we need to break the block chaining to this block (Line 4). That is, the jump stubs of all other blocks that jump to the host binary of this block are redirected back to DBT. In this way, we can guarantee that the verification step is invoked before the host binary of this block is reused. Similar operations are performed on each trace that contains this block (Line 5 ~ 9).

To verify whether the host binary of a block/trace in the *unknown* state can be reused, we compare the guest binary of this block/trace saved in GBCB with its current guest binary to see if there is any difference between them. SIMD instructions available on the host can be used to speedup the comparison process. In most cases, the host binary can be reused if the original guest binary is not modified. However, there could be exceptions due to global optimizations, which are discussed in the next subsection. In addition, we may need to update some jump stubs in the host binary before the reuse because they may jump to the host binary that is in the *unknown* state.

Algorithm 2 shows the detailed reuse process of previously translated host binary, which is invoked before each block is translated. The algorithm firstly checks if there exists a trace that was formed with the block as the entry block (Line 1). If yes, it tries to reuse the host binary of this trace (Line 2 ~ 24). Otherwise, it just tries to reuse the host binary of the block (Line 25 ~ 41). If the state of the host binary of the trace is *valid*, no further verification is required (Line 3 ~ 4). Otherwise, if the state is *unknown*, we compare the guest binary code of each block in this trace if it is not compared before (Line 5 ~ 17). If a modification is detected on the

Algorithm 2: Reuse Host Code

```
Input: The block to be translated.
Output: The address of valid host binary of the block or
        the trace that starts from the block, or NULL.

1  trace ← Trace_Start_Block (block);
2  if trace != NULL then
3      if Host_Code_State (trace) == Valid then
4          return Host_Code_Address (trace);
5      else if Host_Code_State (trace) == Unknown then
6          flag ← True;
7          foreach basic block bb in trace do
8              if Host_Code_State (bb) != Unknown then
9                  continue;
10             if Guest_Code_Same (bb) == True then
11                 Mark_Host_Code_Valid (bb);
12                 Update_Host_Code_Chaining (bb);
13             else
14                 Mark_Host_Code_None (bb);
15                 Flush_Code_Cache (bb);
16                 flag ← False;
17                 break;
18         if flag == False then
19             Mark_Host_Code_None (trace);
20             Flush_Code_Cache (trace);
21         else
22             Mark_Host_Code_Valid (trace);
23             Update_Host_Code_Chaining (trace);
24             return Host_Code_Address (trace);
25 if Host_Code_State (block) == None then
26     return NULL;
27 if Host_Code_State (block) == Valid then
28     return Host_Code_Address (block);
29 if Guest_Code_Same (block) == True then
30     Mark_Host_Code_Valid (block);
31     Update_Host_Code_Chaining (block);
32     return Host_Code_Address (block);
33 else
34     Mark_Host_Code_None (block);
35     Flush_Code_Cache (block);
36     t ← Trace_Contain_Block (block);
37     if t != NULL then
38         foreach each trace trace in t do
39             Mark_Host_Code_None (trace);
40             Flush_Code_Cache (trace);
41     return NULL;
```

guest binary of a block, the host binary of this trace cannot be reused and needs to be flushed from the code cache (Line 18 ∼ 20). Otherwise, we can reuse the host binary code after its jump stubs are updated (Line 22 ∼ 24). A similar verification process is performed for the host binary code of the block before it can be reused. In this way, the host binary of the blocks/traces can be maximally reused in the same execution to mitigate the heavy re-translation overhead for guest applications with DGC.

3.3 Handling Global Optimizations

Global optimizations are often employed by existing DBT systems to improve the performance of the generated host binary. Some global optimizations could complicate the host binary reuse process described above because these global optimizations may need the information from the predecessor and/or the successor of a block/trace. A typical example is the conditional code optimization for x86-like ISAs that have side effects on the condition codes in some instructions [27, 36]. In such cases, if the guest binary of a predecessor or a successor block is changed, the previously translated and globally-optimized host binary of the current block may not be reusable even if it is not changed. The side effects of the changed predecessor or successor block may render the current block not reusable.

To handle the potential inconsistency introduced by global optimizations, we extend our approach to compare the guest binary of all blocks/traces that may affect the translation of the current block/trace. Specifically, each time when a global optimization is applied, the guest binary of the blocks/traces that may affect the translation/optimization decisions of the current block/trace is saved to GBCB. Furthermore, the verification process is extended to cover all of those blocks/traces. That is, the host binary of the current block/trace can be reused only if the corresponding guest binary of all the blocks/traces is not changed. In this way, we can work properly with global optimizations in existing DBT systems.

4 Implementation

A prototype based on the proposed approach has been implemented using HQEMU [15], which is a retargetable DBT system based on QEMU [3]. HQEMU keeps the original translator in QEMU, i.e., Tiny Code Generator (TCG), to translate basic blocks. In addition, an LLVM JIT engine [20] is employed to generate more efficient host binary code for hot traces. HQEMU detects and forms traces through an instrumentation-based scheme. Specifically, a *profile stub* and a *predict stub* are inserted into the beginning of the host code generated by TCG for each basic block. The profile stub is used to count the number of dynamic executions of the block. Once the counter reaches a preset threshold, the profile stub is disabled and the predict stub is enabled to form the trace for this block.

A relaxed version of the NET algorithm [7] is leveraged by HQEMU to form hot traces. The original NET algorithm terminates the trace formation at every backward branch, which is considered as an indicator of a cyclic execution path. HQEMU relaxes such a backward-branch constraint and stops the trace formation when an earlier guest block is emulated again. After a trace is formed, HQEMU translates TCG OPs, which are the intermediate representations (IRs) of pseudo-instructions used by TCG, of each basic block into LLVM IRs. The LLVM JIT engine is then invoked to compile the LLVM IRs into host binary. During this process, several LLVM optimization passes are applied to improve the performance of the generated host binary. Additional overhead could be introduced if more LLVM optimization passes are applied.

In our current implementation, GBCB is allocated on the heap with the initial size 128 MB. To simplify the implementation, we did not manage nor reuse the memory space in GBCB. It is because the space used by guest binaries and GBCB for most of the applications is very small. It is not worth the time overhead to manage the flushed space. A pointer is maintained to mark the beginning of the available space for GBCB. It is also used to check the potential buffer overflow of GBCB, which is very unlikely as mentioned earlier. Each time when the saved guest binary code of a basic block needs to be flushed, we simply reset the address and the size of the saved guest binary code in the block data structure to NULL and 0, respectively.

The page access violations on guest code pages are detected via the host signal SIGSEGV. HQEMU has its own signal handler for SIGSEGV. Each time when a SIGSEGV is triggered on a guest page, HQEMU checks if the page was writable in its original access permission. If yes, that means the page is set to read-only by HQEMU and the SIGSEGV is caused by the modification to the guest code page. We then save guest binary code of blocks in this page and mark the host code of corresponding blocks and traces as unknown.

5 Experimental Results

We use the Google Octane V2.0 [10] benchmark suite to evaluate the proposed approach. Google Octane has been extensively used to measure the performance of JavaScript engines. It contains 15 benchmarks representative of common use cases of JavaScript applications. Unfortunately, HQEMU has errors when emulating the following 5 programs: Mandreel (assertion failed error), RayTrace (Scene rendered incorrectly), RegExp (Wrong checksum), Splay (execution stalled), and TypeScript (random assertion failed). Thus, our experiments cover 10 of the 15 benchmarks. The JavaScript engine used in our experiments is Chrome V8 (or V8 for short) [8], which is written in C++ and has been deployed in many applications, including Google Chrome [9] and Node.js [17].

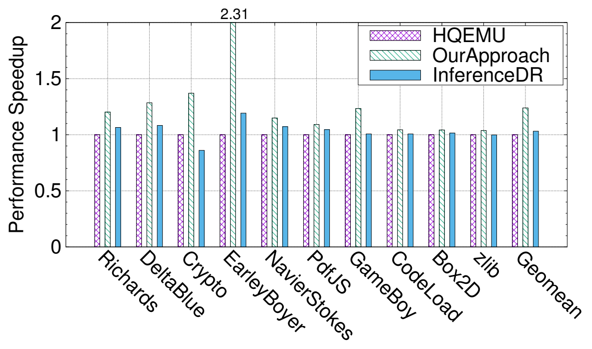


Figure 6. Performance speedup achieved by our proposed approach and InferenceDR [12] with HQEMU as the baseline.

We also compare our approach with InferenceDR [12], a state-of-the-art scheme that is most relevant to our approach. Due to the unavailability of its source code, we implemented its scheme according to the details provided in [12]. In InferenceDR, each time a memory location in a guest code page is modified, it tries to locate its corresponding host binary. To do this, it evenly divides a guest code page into several regions and implements a hashed mapping scheme between the guest binary in each divided region and its corresponding translated host binary.

The experiments are conducted on a machine with a 16-core 2.6 GHz Intel Xeon E5-2650 v2 processor and with hyper-threading enabled. It has a 64GB main memory, and the operating system is Ubuntu 16.04 with Linux-4.10. During the measurements, the machine is set up exclusively to run the evaluated benchmarks. In addition, each benchmark is run five times, and their arithmetic means are used to reduce the potential influence of random errors.

5.1 Performance Results

Figure 6 shows the overall performance results. We compare the performance using our proposed approach with that of original HQEMU and the InferenceDR. As each of them takes a significantly different amount of time, we normalize its execution time against that on HQEMU. On average, our proposed approach can achieve a 1.24X performance speedup compared to HQEMU, whereas InferenceDR only achieves a 1.03X speedup. This demonstrates that our proposed approach can improve the performance of a DBT such as HQEMU for applications with DGC. Moreover, it can achieve higher performance improvement than InferenceDR, because no complicated analysis is required by our approach.

Octane V2.0 has a specific setup to run its benchmarks on the Chrome V8 JavaScript Engine. When they interact with a host DBT such as HQEMU, more details are needed to understand how different aspects of the program execution and runtime setup impact the overall performance and its measurements. Next, we first give some details about the

Octane V2.0 benchmark suite. After that, we present the performance results of different configurations of Octane V2.0. More details on relevant measurements that can better understand the performance differences are presented in Section 5.2. The time and the memory overhead of our proposed scheme are presented in Section 5.3.

We group the benchmarks in Octane V2.0 into two classes: one with a shorter execution time (SE) and the other with a relatively longer execution time (LE). This is important for such benchmarks because the dynamically-generated code at runtime for these benchmarks can be re-optimized or form more efficient traces by the V8 engine depending on how long a benchmark is run during its execution. Longer execution time allows the V8 engine to apply more optimizations with better performance for the benchmark, while a shorter execution time will not be able to afford the re-optimization overhead. We use the ratio between the measured time spent in executing the benchmark vs. the time spent in the V8 engine to classify them into the SE or LE group. Knowing which group a benchmark is in gives a better perspective of its measured performance. We also use slightly different experimental setups for each group of benchmarks to have a better understanding of how they interact with a DBT system. An abbreviated name is provided for each benchmark in parentheses for easier reference.

- In SE Group: Richards (*Ric*), DeltaBlue (*Del*), Crypto (*Cry*), EarleyBoyer (*Ear*), NavierStokes (*Nav*)
- In LE Group: PdfJs (*Pdf*), Gameboy(*Gam*), CodeLoad (*Cod*), Box2D (*Box*), zlib (*zli*)

Next, we describe all execution phases required when an Octane benchmark is run under Octane's *test framework* in its *native environment*, i.e., on a guest machine, as opposed to run on a DBT such as HQEMU. After that, we discuss how these execution phases interact with a DBT system and how we need to adjust our experimental setups to take these execution phases into consideration when it is run on a DBT system such as HQEMU.

In a *native* environment, the V8 engine will be loaded and initiated first by the operating system, followed by loading the Octane test framework (also written in JavaScript) by the V8 engine. After such an initial setup is completed, each Octane benchmark will then be loaded and run sequentially on the engine under the control of the test framework. The *test framework* is simply a script that sequentially runs the entire set of the Octane benchmarks, collects execution time and performance data, and produces a "score" for each benchmark program, as well as a score for the entire benchmark suite. During the execution of each Octane benchmark, it will be emulated and translated by the V8 engine, and translated into binary code that is stored in its code cache.

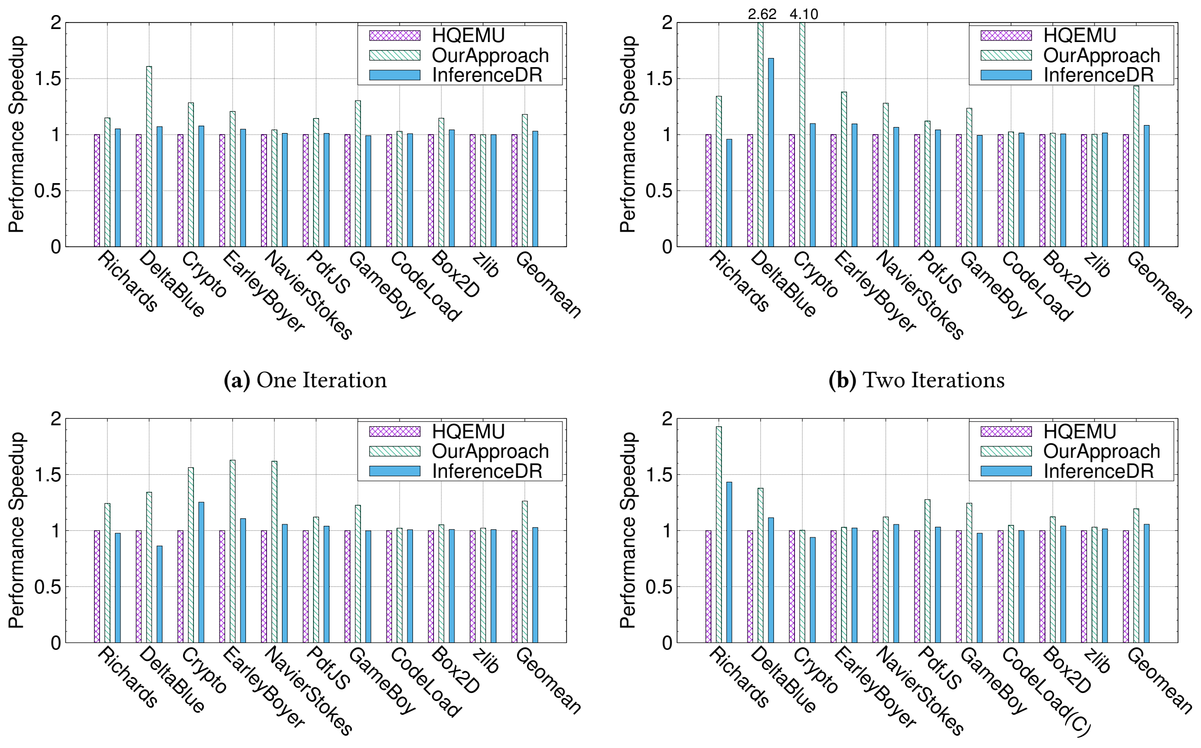
Instead of being in a native environment, when it is emulated on a DBT such as HQEMU, all execution phases remain the same, except the V8 engine is loaded by HQEMU (instead of the native operating system) initially, and the test framework is bootstrapped from there until all of the execution phases described above are completed. During the emulation on HQEMU, each execution phase will interact with the various mechanisms on HQEMU, which include HQEMU's own translation, emulation, as well as its mechanism to maintain the consistency between the guest code dynamically generated by the V8 engine and the translated host code in its code cache as described in the earlier sections. Our proposed approach tries to reduce or mitigate some of the translation overheads in HQEMU for a better overall performance.

As the loading of the V8 engine and the Octane test framework are only done in the initialization phase, it is reasonable to exclude them from the performance measurements for each Octane benchmark. Hence, in the following experiments, all of the performance measurements for each benchmark only start at the time when the benchmark is being loaded by the Octane test framework until its completion.

The Octane test framework can be set up in a variety of ways. For example, after the test framework is setup, we can run *all* of the benchmarks through and collect performance data for each benchmark. We can also run a test with one *single* Octane benchmark but run multiple iterations of the same benchmark, and take the average run time as its performance result. The performance data collected using these two different setups can be quite different, in particular, for benchmarks in the SE group (i.e., with short running time). Running an SE benchmark with multiple iterations in one test will provide enough time for the JIT optimization and trace formation mechanism to kick in for a better performance, as opposed to run it only once. This may not have that much impact for benchmarks in the LE group as each iteration of their execution can take long enough time to have such performance benefit.

It is important to note that the impact of run time on whether the JIT optimization and trace formation can be triggered or not might be magnified by the fact that we have *two* JIT engines as we run the Octane benchmarks on HQEMU, i.e., one in the guest V8 Engine and one in host HQEMU. Both are sensitive to run time when they apply their optimization and trace formation. In order to study such an effect, we run each Octane benchmark once, twice, four times and fifty times in each test run. It is worth noting that in the default configuration of the test framework, a fixed execution time is used to test each benchmark, and thus different benchmarks may run different iterations. That means, shorter benchmarks can run more iterations, while longer benchmarks run fewer iterations.

Figure 7 shows the experimental results for different iterations. Two interesting phenomena can be observed from the figure. First, for each evaluated iteration, the performance speedup achieved by our approach can always outperform those achieved by InferenceDR. This demonstrates again the



(a) One Iteration

(b) Two Iterations

(c) Four Iterations

(d) Fifty Iterations

Figure 7. Performance speedup for each benchmark using its time on HQEMU as the baseline with specific fixed iteration(s).

effectiveness of our approach. Second, the highest performance speedup for different benchmarks can be achieved at different iterations by our approach. For instance, some benchmarks, e.g., DeltaBlue and Crypto, achieve the highest speedup at two iterations, while some benchmarks, e.g., EarleyBoyer and NavierStokes, achieve the highest speedup at four iterations. The reason behind this phenomenon could be that the behavior of the JavaScript benchmarks can affect the code generation in the V8 engine, and further affect the trace formation and translation in HQEMU.

5.2 Performance Analysis

To better understand the measured performance data for the Octane benchmarks, we profile each benchmark during its emulation on HQEMU, and collect some details about its runtime behavior and its interaction with HQEMU, especially for those relevant to our proposed approach. In the following experiments in this subsection, we exclude the test framework in the Octane benchmark suite, because it is also written in JavaScript and could introduce potential influences on the trace formation from DGC in HQEMU. Also, the benchmark Gameboy is excluded because it cannot run on the V8 engine without the test framework.

Trace Translation Overhead. Table 1 presents the trace translation overhead introduced by the original HQEMU and our approach. We show the 5 benchmarks in the SE group in the first 5 rows, and the 4 benchmarks in the LE group in the next 4 rows. Here the GameBoy benchmark is missed due to the reason mentioned before. From the second column through the fourth column, we show the trace generation time on HQEMU in milliseconds, and from the fifth through the seventh column are the data for our approach. The column under the title “Total” shows the total emulation time of each benchmark in milliseconds. The column under the title “Trace” shows the total trace formation time for each benchmark. The column under “Ratio” shows the percentage of the time spent in the trace translation, i.e., it is the “Trace” column divided by the “Total” column.

It is obvious that for benchmarks in the LE group, i.e., long execution-time, trace translation time takes up a smaller percentage of the total emulation time. However, the percentage is much higher in the SE group. As our proposed approach tries to preserve and reuse the translated host binary code of blocks and traces, the percentage of time spent in trace translation is significantly smaller compared to its HQEMU counter part as shown in the table.

Reuse of Generated Traces from DGC. As trace generation could take a significant amount of time in HQEMU for DGC code, the amount of trace reuse enabled by our approach indicates the re-translation overhead that can be mitigated. The experimental results are shown in Table 2. The first column of the table shows all of the benchmarks in

	HQEMU			Our Approach		
	Total	Trace	Ratio	Total	Trace	Ratio
Ric	912	165	18%	817	98	12%
Del	3142	906	28%	2516	758	30%
Cry	2112	877	41%	1611	504	31%
Ear	10866	1751	16%	8448	922	9.7%
Nav	3319	689	20%	2793	196	7.0%
Pdf	110394	13294	12%	91156	6073	6.7%
Cod	22372	460	2.1%	22368	262	1.2%
Box	109812	20326	18%	93632	2111	2.3%
zli	112330	6106	5.4%	112317	4275	3.8%

Table 1. The percentage of time spent in trace translation.

	HQEMU	Our Approach	Modified	Reused	Ratio (%)
Ric	85	57	1	116	99.15
Del	318	156	1	411	99.76
Cry	282	127	0	186	100
Ear	547	214	12	785	98.49
Nav	205	76	2	220	99.1
Pdf	4600	2594	25	4572	99.46
Cod	161	95	0	361	100
Box	7574	4924	62	10302	99.4
zli	1835	1424	28	944	97.12

Table 2. Statistics of trace reuse enabled by our approach.

	Total	PF	BLK	%PF	%BLK
Ric	912	10.430	5.890	1.14%	0.65%
Del	3142	42.787	21.214	1.36%	0.68%
Cry	2112	24.170	26.581	1.14%	1.26%
Ear	10866	186.435	37.880	1.72%	0.35%
Nav	3319	43.259	19.895	1.30%	0.60%
Pdf	110394	1424.717	616.079	1.29%	0.56%
Cod	22372	436.078	84.110	1.95%	0.38%
Box	109812	1886.982	984.077	1.72%	0.90%
zli	112330	1269.707	401.581	1.13%	0.36%

Table 3. The percentage of time spent in the SIGSEGV handler and the block translation.

their abbreviated names. The second column shows the total number of traces generated in HQEMU for each benchmark. It is important to note that, as the traces can be invalidated and re-generated due to page protection mechanism mentioned earlier even if they are not modified, the count may include the re-generated traces that are the same as before they are invalidated, i.e., it includes redundantly-generated traces due to the invalidation scheme used in HQEMU. The third column is the total number of traces generated using our proposed approach, which tries to preserve and reuse as many unmodified traces as possible. As shown in the table, the number of traces generated (or re-generated) is significantly lower than that of HQEMU's.

The fourth and the fifth columns of the table are for our approach only. In the fourth column, we show the total number of traces invalidated due to the modification of the guest binary code by the V8 engine. That is, these traces need to be re-formed and re-translated. The fifth column shows the total number of traces our approach identifies as unmodified and reusable after a page protection violation is triggered. It is worth noting that a trace could be reused multiple times because its host binary code may be marked as *unknown* multiple times. The last column shows the trace reuse ratio of our approach, i.e., the fifth column divided by the sum of the fourth column and the fifth column. It is quite clear that the reuse ratio of our proposed approach is extremely high.

Other Performance Overhead in DBT. Table 3 shows the experimental results of he performance overhead incurred by capturing the modifications of the dynamically-generated code and the block-only translation.

The first column of the table again shows all of the benchmarks in their abbreviated names. The second column shows the total emulation time of each benchmark in milliseconds. The third column shows the time spent in the SIGSEGV handler, which is used to capture the page protection violations as described in Section 4. As a comparison to its relative significance in terms of the overall overhead during the emulation, we include the total translation time for the all of the basic blocks translated during the emulation for each benchmark in the fourth column, i.e., not including the trace translation overhead. The fifth column shows the total page-fault overhead in percentage, i.e., it is the ratio between "PF" and "Total". Similarly, the last column shows the overhead of the basic block translation in percentage.

It is quite clear from the data in Table 3 that the page-fault overhead is quite small for all of the benchmarks. It is an indication that the modifications to the code cache from the V8 engine caused by its dynamically-generated code are quite rare. However, the page-wide invalidation of the translated host code when it happens could result in significant re-translation overhead as described before.

5.3 Performance and Memory Overhead

Performance Overhead. In order to measure the overall performance overhead incurred by our proposed approach, we use the following strategy. The main idea is to use our proposed approach for each benchmark, but use the worst-case scenario that makes all of the attempts to preserve and reuse the translated blocks and traces fail, i.e., we still have to re-translate all of the blocks and traces as in the original HQEMU without using our proposed scheme. This will incur *all* of the overhead in our proposed scheme, but without

	Ric	Del	Cry	Ear	Nav	Pdf	Gam	Cod	Box	zli	Geomean
HQEMU	34.62	692.66	562.95	887.37	58.46	47030.50	65775.25	4276.33	15447.00	13378.60	
WC	37.69	699.59	609.39	930.42	58.89	48559.25	68814.00	4542.99	17915.10	14631.30	
Ratio	8.87%	1.00%	8.25%	4.85%	0.73%	3.25%	4.62%	6.24%	15.98%	9.36%	4.53%

Table 4. Performance overhead for each benchmark in its default configuration. The time is in millisecond (ms).

	Ric	Del	Cry	Ear	Nav	Pdf	Gam	Cod	Box	zli
Memory (KB)	155.09	372.04	428.36	475.94	104.44	1926.20	2695.95	1938.33	2733.66	713.30

Table 5. Memory overhead used in saving guest binary for each Octane benchmark in its default configuration.

any performance benefit. We calculate the additional time used in this worst-case scenario compared with the time in original HQEMU. The emulation time with these two scenarios for each benchmark is shown in Table 4, in which the row marked as "HQEMU" is the emulation time of the original HQEMU for each benchmark. The row marked as "WC" is the emulation time assuming the worst-case scenario of our approach as described above. From the table, we found the average overhead is less than 5%, which is quite small.

Memory Overhead. We then study the total memory usage in our approach. In our implementation, the initial size of GBCB is 128MB, as mentioned in Section 4. To minimize the memory management overhead, we do not reclaim and reuse the buffer space. We use a pointer to mark the starting position of the free space in GBCB. Hence, by calculating the offset of the pointer to the start of the buffer, we can obtain the buffer usage. The results are shown in Table 5. From the data we can see that all of the benchmarks use less than 3MB of buffer space, which indicates that the memory overhead introduced by the proposed approach is very low.

Impact of Different Iterations. To understand the impact of repeated emulation of the same benchmark on the overall performance and memory overhead, we collect such data by running each benchmark in a test with 1, 2, 4, and 50 iterations. The strategies and the mechanisms used are the same as those described in Section 5.1. The data collected are presented in Table 6 for the overall performance overheads, and in Table 7 for the overall memory overheads.

For overall performance overhead, it can show us when more translated blocks and traces are generated and maintained through the repeated emulations of the same benchmark whether any additional performance overhead can be incurred by using our approach. For memory usage, this can give us some indication of whether more efficient memory reclaim and reuse schemes need to be employed when multiple emulation runs are the usage scenario of an application.

As shown in Table 6, the overall performance overhead remain low for all of the benchmarks in all of the repeated runs. Also, the general trend for the overhead is to go down because such overhead can generally be amortized across different emulation runs. However, the trends could be erratic for some benchmarks. From the data in Table 7, we can see that the memory overhead is still very small and is quite stable, independent of the number of times it is emulated for most of the benchmarks even after emulating 50 iterations, except two benchmarks: Cod (i.e., CodeLoad) and Pdf (i.e., PdfJs), whose memory usages seem to have increased after 50 iterations, but not at a linear rate. This may indicate that for some benchmarks a more efficient memory reclaim and reuse mechanism may be required if multiple emulation runs are their normal usage scenario.

6 Related Work

To optimize DBT systems for guest applications with dynamically generated code, previous work proposes two mechanisms [12]. The first mechanism asks developers to annotate source code to declare dynamic code regions. This information can be embedded into guest binary code via the compilation process. In this way, the DBT system can precisely monitor the modifications to the dynamically-generated guest code and invalidate the corresponding translated host binary code. Although this mechanism can effectively reduce the re-translation overhead, it is not practical because it requires the re-compilation of guest applications. To overcome this limitation, the second mechanism, called InferenceDR, is proposed to selectively instrument guest write instructions to identify the modifications to guest binary code. Each time a memory location in a guest code page is modified, it tries to find out if there exists any translated host binary code that corresponds to this memory location. If yes, the host binary code is invalidated and flushed. In this way, the consistency between the guest code and the host code can be preserved.

Compared to InferenceDR, our approach has at least two advantages. First, the performance overhead introduced by our approach to reuse the host binary code is very small. It is because we only need to save and compare the guest binary code and no additional analysis is required. In contrast, a considerable performance overhead can be incurred by the analysis needed in InferenceDR. It is because the analysis is required for every write instruction to the guest code

		Ric	Del	Cry	Ear	Nav	Pdf	Gam	Cod	Box	zli	Geomean
1	HQEMU	832	2687	4823	3540	1904	99180	159033	15616	106077	105029	
	WC	882	2925	5253	3701	2001	108800	168968	15919	123659	107189	
	Ratio	6.01%	8.86%	8.92%	4.55%	5.09%	9.70%	6.25%	1.94%	16.57%	2.06%	5.81%
2	HQEMU	23.5	390	6999	1394.5	228	71855	110560.5	13290.5	63191.5	55027.5	
	WC	25.5	401	7315	1560.5	232	75846.5	117074.5	14326.5	67163	61038.5	
	Ratio	8.51%	2.82%	4.51%	11.90%	1.75%	5.55%	5.89%	7.80%	6.28%	10.92%	5.75%
4	HQEMU	41	80.5	842.75	830.5	80.5	46687.5	65049.5	9029.75	33532.25	29381.5	
	WC	42.5	84.25	892.75	947.5	85.25	48161.75	67633.5	9415.25	35192.75	33168.5	
	Ratio	3.66%	4.66%	5.93%	14.09%	5.90%	3.16%	3.97%	4.28%	4.95%	12.89%	5.57%
50	HQEMU	12.4	17.66	279.74	173.46	14.18	11445.24	6705.08	3215.58	3411.88	4341.64	
	WC	12.9	18.62	287.24	193.9	14.84	12354.58	7108	3484.16	3841.34	4393.12	
	Ratio	4.03%	5.44%	2.68%	11.78%	4.65%	7.95%	6.01%	8.35%	12.59%	1.19%	5.35%

Table 6. Performance overhead introduced by our approach under different iterations.

	Ric	Del	Cry	Ear	Nav	Pdf	Gam	Cod	Box	zli
1	126.85	275.85	273.73	321.49	90.80	1105.32	1577.60	187.77	2092.16	737.39
2	132.19	291.56	299.84	360.79	94.77	1438.96	2464.90	293.23	2452.09	706.92
4	132.87	293.02	318.37	397.55	97.81	1937.23	2655.88	519.27	2486.38	815.08
50	159.34	353.14	464.66	476.35	103.27	8950.80	3242.37	5771.54	2951.73	824.22

Table 7. Memory overhead used in saving guest binary code with different iterations (size: KB).

page. This overhead can significantly offset the performance benefit obtained by reducing the redundant re-translation. Second, our approach can support global optimizations such as the condition-code optimization in existing DBT systems. However, InferenceDR cannot work well with global optimizations because it does not record the block dependences needed in global optimizations.

A similar scheme using guest code comparison, called *guest binary verification*, is employed to reuse translated host code for re-loadable guest modules [21]. It is very different from the objective of this paper, which is targeting guest applications with dynamically-generated code. The major challenge of our approach is that the dynamically-generated guest code could be modified by the guest JIT engine at runtime during the execution, whereas there is no such runtime requirement in guest binary verification.

There are also several schemes to optimize DBT systems in previous research work [16, 22, 35, 37, 39]. In general, our approach can leverage those optimizations to further improve the emulation performance.

7 Conclusion

This paper presents a novel approach to reuse the host binary code generated for guest applications with dynamically-generated code, such as those developed in scripting languages, on a dynamic binary translator such as HQEMU. Each time when a modification to a guest code page is detected, the proposed approach saves the guest binary code of each basic block in this page. The saved guest binary code is used to verify whether the previously translated host binary code can be re-used before a block needs to be re-translated to mitigate the re-translation overhead. A prototype based on such an approach has been implemented using an existing DBT system HQEMU. Experimental results on a set of JavaScript benchmarks from Google Octane demonstrate that the proposed approach can achieve an average of 1.24X performance speedup compared to the original HQEMU.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1514444. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

This work is also partially supported by the Natural Science Foundation of Tianjin, China under Grant No. 16JCYBJC15200 and 17JCQNJC00300, the Major Science and Technology Program of Big Data and Cloud Computing of Tianjin, China under Grant No. 15ZXDSGX00020, the National Key Research and Development Program of China under Grant No. 2016YFC0400709, the National Natural Science Foundation of China (61702286), Open Projects of State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences (CARCH201604), and Special Funding for Excellent Enterprise Technology Correspondent of Tianjin, China (17JCTPJC49500).

References

[1] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. 2014. The Hiphop Virtual Machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 777–790. https://doi.org/10.1145/2660193.2660199

[2] Wonsun Ahn, Jiho Choi, Thomas Shull, María J. Garzarán, and Josep Torrellas. 2014. Improving JavaScript Performance by Deconstructing the Type System. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 496–507. https://doi.org/10.1145/2594291.2594332

[3] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATC '05)*. USENIX Association, Berkeley, CA, USA, 41–46. http://dl.acm.org/citation.cfm?id=1247360.1247401

[4] Derek L. Bruening. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* Ph.D. Dissertation. Cambridge, MA, USA. AAI0807735.

[5] 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 (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 463–469. http://dl.acm.org/citation.cfm?id=2032305.2032342

[6] Monika Dhok, Murali Krishna Ramanathan, and Nishant Sinha. 2016. Type-aware Concolic Testing of JavaScript Programs. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 168–179. https://doi.org/10.1145/2884781.2884859

[7] Evelyn Duesterwald and Vasanth Bala. 2000. Software Profiling for Hot Path Prediction: Less is More. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. ACM, New York, NY, USA, 202–211. https://doi.org/10.1145/378993.379241

[8] Google. Accessed: November 27, 2017. Chrome V8 JavaScript Engine. (Accessed: November 27, 2017). http://developers.google.com/v8

[9] Google. Accessed: November 27, 2017. Chrome Web Browser. (Accessed: November 27, 2017). http://www.google.com/chrome

[10] Google. Accessed: November 27, 2017. Octane: The JavaScript Benchmark Suite for the modern web. (Accessed: November 27, 2017). http://developers.google.com/octane

[11] Dibakar Gope, David J. Schlais, and Mikko H. Lipasti. 2017. Architectural Support for Server-Side PHP Processing. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 507–520. https://doi.org/10.1145/3079856.3080234

[12] Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao. 2015. Optimizing Binary Translation of Dynamically Generated Code. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, Washington, DC, USA, 68–78. http://dl.acm.org/citation.cfm?id=2738600.2738610

[13] Hiroshige Hayashizaki, Peng Wu, Hiroshi Inoue, Mauricio J. Serrano, and Toshio Nakatani. 2011. Improving the Performance of Trace-based Systems by False Loop Filtering. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 405–418. https://doi.org/10.1145/1950365.1950412

[14] David Hiniker, Kim Hazelwood, and Michael D. Smith. 2005. Improving Region Selection in Dynamic Optimization Systems. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 38)*. IEEE Computer Society, Washington, DC, USA, 141–154. https://doi.org/10.1109/MICRO.2005.22

[15] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. 2012. HQEMU: A Multi-threaded and Retargetable Dynamic Binary Translator on Multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA, 104–113. https://doi.org/10.1145/2259016.2259030

[16] Chun-Chen Hsu, Pangfeng Liu, Jan-Jan Wu, Pen-Chung Yew, Ding-Yong Hong, Wei-Chung Hsu, and Chien-Min Wang. 2013. Improving Dynamic Binary Optimization Through Early-exit Guided Code Region Formation. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '13)*. ACM, New York, NY, USA, 23–32. https://doi.org/10.1145/2451512.2451519

[17] Joyent Inc. Accessed: November 27, 2017. Node.js. (Accessed: November 27, 2017). http://nodejs.org

[18] David Keppel. 1991. A Portable Interface for On-the-fly Instruction Space Modification. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*. ACM, New York, NY, USA, 86–95. https://doi.org/10.1145/106972.106983

[19] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting Superword Level Parallelism with Multimedia Instruction Sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 145–156. https://doi.org/10.1145/349299.349320

[20] Chris Lattner. 2002. *LLVM: An Infrastructure for Multi-Stage Optimization.* Master's thesis. Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL. *See* `http://llvm.org`.

[21] Jianhui Li, Peng Zhang, and Orna Etzion. 2005. Module-aware Translation for Real-life Desktop Applications. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments (VEE '05)*. ACM, New York, NY, USA, 89–99. https://doi.org/10.1145/1064979.1064993

[22] Yu-Ping Liu, Ding-Yong Hong, Jan-Jan Wu, Sheng-Yu Fu, and Wei-Chung Hsu. 2017. Exploiting Asymmetric SIMD Register Configurations in ARM-to-x86 Dynamic Binary Translation. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT '17)*. IEEE Computer Society, Washington, DC, USA, 343–355. https://doi.org/10.1109/PACT.2017.15

[23] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[24] Mozilla. Accessed: November 27, 2017. SpiderMonkey JavaScript Engine. (Accessed: November 27, 2017). http://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

[25] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[26] Semih Okur, David L. Hartveld, Danny Dig, and Arie van Deursen. 2014. A Study and Toolkit for Asynchronous Programming in C#. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 1117–1127. https://doi.org/10.1145/2568225.2568309

[27] Guilherme Ottoni, Thomas Hartin, Christopher Weaver, Jason Brandt, Belliappa Kuttanna, and Hong Wang. 2011. Harmonia: A Transparent, Efficient, and Harmonious Dynamic Binary Translator Targeting the Intel® Architecture. In *Proceedings of the 8th ACM International Conference on Computing Frontiers (CF '11)*. ACM, New York, NY, USA, Article 26, 10 pages. https://doi.org/10.1145/2016604.2016635

[28] Vijay Janapa Reddi, Dan Connors, Robert Cohn, and Michael D. Smith. 2007. Persistent Code Caching: Exploiting Code Reuse Across Executions and Applications. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*. IEEE Computer Society, Washington, DC, USA, 74–88. https://doi.org/10.1109/CGO.2007.29

[29] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. 2012. Automated Repair of HTML Generation Errors in PHP Applications Using String Constraint Solving. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 277–287. http://dl.acm.org/citation.cfm?id=2337223.2337257

[30] Kevin Scott and Jack Davidson. 2001. *Strata: A Software Dynamic Translation Infrastructure.* Technical Report. Charlottesville, VA, USA.

[31] Jim Smith and Ravi Nair. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[32] 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 (ICISS '08)*. Springer-Verlag, Berlin, Heidelberg, 1–25. https://doi.org/10.1007/978-3-540-89862-7_1

[33] Android Studio. Accessed: November 27, 2017. Run Apps on the Android Emulator. (Accessed: November 27, 2017). http://developer.android.com/studio/run/emulator.html

[34] Cheng Wang, Shiliang Hu, Ho-seop Kim, Sreekumar R. Nair, Mauricio Breternitz, Zhiwei Ying, and Youfeng Wu. 2007. StarDBT: An Efficient Multi-platform Dynamic Binary Translation System. In *Proceedings of the 12th Asia-Pacific Conference on Advances in Computer Systems Architecture (ACSAC'07)*. Springer-Verlag, Berlin, Heidelberg, 4–15. http://dl.acm.org/citation.cfm?id=2392163.2392166

[35] Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. 2018. Enhancing Cross-ISA DBT Through Automatically Learned Translation Rules. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA. https://doi.org/10.1145/3173162.3177160

[36] Wenwen Wang, Chenggang Wu, Tongxin Bai, Zhenjiang Wang, Xiang Yuan, and Huimin Cui. 2014. A Pattern Translation Method for Flags in Binary Translation. *Journal of Computer Research and Development* 51, 10 (2014). http://crad.ict.ac.cn/EN/10.7544/issn1000-1239.2014.20130018

[37] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. 2016. A General Persistent Code Caching Framework for Dynamic Binary Translation (DBT). In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16)*. USENIX Association, Berkeley, CA, USA, 591–603. http://dl.acm.org/citation.cfm?id=3026959.3027013

[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 (MobiSys '17)*. ACM, New York, NY, USA, 319–331. https://doi.org/10.1145/3081333.3081337

[39] Xiaochun Zhang, Qi Guo, Yunji Chen, Tianshi Chen, and Weiwu Hu. 2015. HERMES: A Fast cross-ISA Binary Translator with Post-optimization. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, Washington, DC, USA, 246–256. http://dl.acm.org/citation.cfm?id=2738600.2738631