A General Persistent Code Caching Framework for Dynamic Binary Translation (DBT)

Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant,
University of Minnesota, Twin Cities

https://www.usenix.org/conference/atc16/technical-sessions/presentation/wang

This paper is included in the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

Open access to the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16) is sponsored by USENIX.

A General Persistent Code Caching Framework for Dynamic Binary Translation (DBT)

Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant
University of Minnesota, Twin Cities
{wenwang, yew, zhai, mccamant}@cs.umn.edu

Abstract

Dynamic binary translation (DBT) translates binary code from one instruction set architecture (ISA) to another (same or different) ISA at runtime, which makes it very useful in many applications such as system virtualization, whole program analysis, system debugging, and system security. Many techniques have been proposed to improve the efficiency of DBT systems for long-running and loop-intensive applications. However, for applications with short running time or long-running but with few hot code regions such as JavaScript and C# applications in web services, such techniques have difficulty in amortizing the overhead incurred during binary translation.

To reduce the translation overhead for such applications, this paper presents a general persistent code caching framework, which allows the reuse of translated binary code across different executions for the same or different applications. Compared to existing approaches, the proposed approach can seamlessly handle even dynamically generated code, which is very popular in script applications today. A prototype of the proposed framework has been implemented in an existing retargetable DBT system. Experimental results on a list of applications, including C/C++ and JavaScript, demonstrate that it can achieve 76.4% performance improvement on average compared to the original DBT system without helper threads for dynamic binary translation, and 9% performance improvement on average over the same DBT system with helper threads when code reuse is combined with help threads.

1 Introduction

Dynamic binary translation or transformation (DBT) has been widely used in many applications such as system virtualization, whole program analysis, system debugging, and system security. Most notable examples include QEMU [6], DynamoRIO [4], Pin [17], Valgrind [18], and many others [9, 25, 5]. DBT systems can dynamically translate *guest* binary code in one instruction set architecture (ISA) to another *host* ISA (same as or different from guest ISA), and achieve the emulation of guest applications on the host machines or the enhanced functionalities of the guest binaries. It bypasses the need of an intermediate representation such as bytecode in language-level virtual machines, e.g. Java virtual machine (JVM) or Dalvik in Android.

Compare to native execution, DBT systems usually consist of two phases. In the first phase, guest binaries are emulated and profiled on the host system to detect the ”hotness” of code regions. Hot code regions are then translated and stored in a code cache. It allows the execution to enter the second phase in which the translated binaries in the code cache are executed without further code emulation or translation. Typically, for long-running and loop-intensive guest applications, 90% of the execution time could stay in the second phase [21]. It allows the overhead incurred in the first phase to be substantially amortized.

Many techniques have been proposed to improve the efficiency of the DBT systems and the performance of the translated host binaries [26, 27, 10]. Nevertheless, for many applications with short running time or long-running applications with few hot code regions, e.g., JavaScript and C# applications in web services [11] that require fast response time and high throughput, such techniques have difficulty in amortizing the overhead from the first phase.

Figure 1 shows the translation overhead (incurred in the first phase) of SPEC CINT2006 using HQEMU [13], a QEMU-based retargetable DBT that can dynamically translate guest binaries across several different major ISAs such as x86-32, x86-64 and ARM. To more accurately measure the translation overhead of short-running applications, the small *test* input is used to shorten their running time.

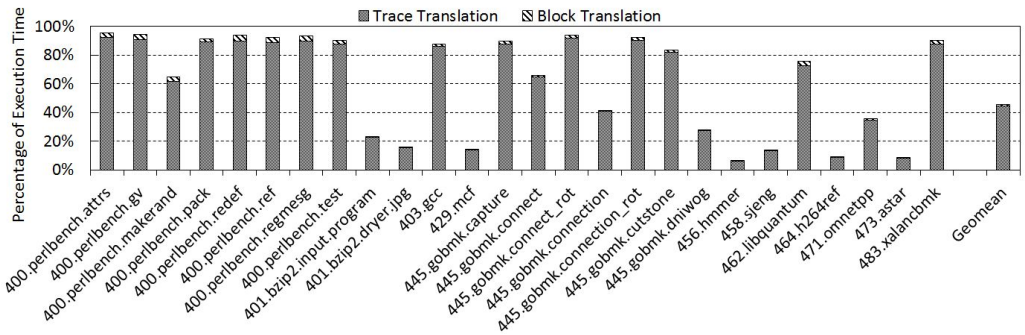

Figure 1: Translation overhead in HQEMU for SPEC CINT2006 with *test* input.

As shown in Figure 1, more than 40% of the execution time on average is spent in the first phase for dynamic translation (more details are discussed in Section 2). Therefore, reducing the translation overhead can significantly improve the performance of these short-running applications or those with few hot code regions. Additionally, lower translation overhead can help to reduce power consumption - a benefit critical to mobile devices with limited battery life. This is one of the reasons Google switches Android runtime from Dalvik to ART with native binaries [1].

One possible approach to reduce translation overhead is to use *static* binary translation (SBT), and perform binary translation offline. It can completely avoid such translation overhead at runtime. However, SBT has many open issues to deal with, such as code discovery for ISAs with variable-length instructions (e.g., Intel x86) and indirect branches with unknown branch targets [8, 24]. Also, it is difficult for SBT to leverage runtime information to optimize the performance of the translated code.

Besides SBT, helper threads have been proposed to shift the translation overheads to other threads [13]. Even though the translation overheads can be hidden substantially this way due to concurrency, it can reduce the system throughput, which could be sensitive in a cloud environment. To reduce re-translation overhead caused by code cache invalidation, annotation-based and inference-based schemes have been proposed to mitigate such overheads [11]. Even so, they still require re-translation the first time those guest binaries are re-encountered.

Compared to those approaches, persistent code caching is an effective alternative to reduce translation overhead [12, 7, 21, 22]. In this approach, the translated binaries are reused the next time the same guest application is executed (i.e., the generated binaries persist across different runs of the same application), or reused by other applications (i.e., the generated binaries such as those in shared libraries persist across different runs of different applications). The translation overhead can thus be reduced (or eliminated). Existing persistent code caching approaches [7, 21, 22] leverage guest instruction address or offset to detect persistent code hits when re-using persistent code. It limits their applicability to unmodified guest binary code at the same instruction address or offset across executions. However, different executions are very likely to have different guest binary code at the same instruction address or offset in practice, e.g., dynamically generated guest binary code.

In this paper, we focus on some main challenges in DBT systems using persistent code caching. The first is the need to deal with relocatable *guest* binaries. The second is the need to generate relocatable *host* binaries in order to persist across different runs for reuse. The third is the need to deal with the dynamically generated code by the guest applications, e.g., if the guest binary is dynamically generated by a just-in-time (JIT) engine, we need to deal with both the dynamically generated guest binaries by the JIT engine and the JIT engine itself [11], so both can persist across different runs.

Relocatable Guest Binaries. Typically, there are two kinds of relocatable guest binaries, (1) position-independent guest binaries, and (2) guest binaries that contain relocatable meta-data for load-time relocation. If a guest binary is position-independent, the offset from its starting address can be used to index and search for its persistent code [7]. However, a shared library may have different offsets when statically linked in different applications, hence, the offset alone is not sufficient to reuse persistent code across statically linked applications that share the same libraries. For the guest binaries that contain relocatable meta-data for load-time relocation, it is also not sufficient to index and search for its persistent code with only addresses of the guest binary code [21].

Relocatable Host Binaries. Position-dependent addresses exist in the translated host instructions need to be relocated. A typical example is the translation of a guest *call* instruction into two host instructions, *push* `next_guest_PC` and *jmp*. The position-dependent

branch target `next_guest_PC` needs to be made position independent (i.e., relocatable) in order to be effectively reused across different execution runs.

Another example is the exit stub in a translated code trace for returning to the DBT runtime system when the next guest code block is untranslated. The returned address of the next guest block needs to be made relocatable as this address could be different in different execution runs due to relocatable guest binaries.

In addition, a position-dependent host address could be embedded in a translated host instruction to jump back from the translated host code to the translator. A possible solution to this challenge is to ask the DBT to generate position-independent code and to avoid using host instructions with embedded position-dependent addresses. However, this kind of implementation is not always efficient [7]. Especially for applications with long running time or mostly covered with hot code regions that do not require persistent code caching, position-independent code could introduce unnecessary runtime overhead.

Dynamically Generated Guest Binaries. JIT engines have been widely adopted in many languages such as Java, JavaScript, and C#. Persistent code caching for dynamically generated code by such JIT engines is very challenging because their addresses are very likely to change across runs due to address space layer randomization (ASLR) used by operating systems for security reasons. Furthermore, despite the availability of bytecode for such applications, their dynamically generated binaries are often linked with other application-specific libraries making them difficult to port across different ISAs with only bytecode. One study shows that an average of more than 50% of such codes are in native binaries [15]. Many of the applications written in languages such as JavaScript also either have very short running time or have very few hot code regions [20]. Persistent code caching is an effective way to reduce re-translation overhead and to improve performance when ported across different ISAs.

This paper aims at addressing the above challenges in persistent code caching for DBT systems across different ISAs. We generate and maintain relocation records in the persistent code to get around the issues of position-dependent code mentioned above. Using such relocation records, host instructions that contain position-dependent addresses can be relocated before it is used. We also keep the original guest binary code in the generated persistent code. Only the persistent code that have the matched guest binary is reused. In this way, persistent code for dynamically generated code can be reused seamlessly even if the address of the generated code is changed in a later execution run.

A prototype of such a persistent code caching framework has been implemented in an existing retargetable DBT system. Experimental results on a set of benchmarks that include C/C++ and JavaScript applications demonstrate the effectiveness and efficiency of our approach. Without trying to optimize our prototype, 76.4% and 9% performance improvements on average are observed compared to the original system without and with helper threads for dynamic binary translation, respectively. Experiments on using persistent code across different execution runs for the same application with different inputs show that the size of the persistent code traces affect the benefits of such persistent code. Additionally, experiments on the persistent code accumulation across different applications suggest that an effective persistent code management is required.

The contributions of this paper are as follows:

- A general persistent code caching framework using relocation records and guest binary code matching is proposed to allow position-dependent host binaries to be reused across different execution runs. This approach is also applicable to dynamically generated guest binaries from JIT compilers. To the best of our knowledge, this is the first attempt to allow dynamically generated guest code to persist across different execution runs.
- A prototype for such a persistent code caching framework has been implemented in an existing DBT system, HQEMU [13], which is a QEMU-based retargetable DBT system.
- A number of experiments are conducted to demonstrate the effectiveness and efficiency of this approach. 76.4% and 9% performance improvements are observed with the help of persistent code caching compared to a retargetable DBT system (i.e., HQEMU) without and with the helper threads for dynamic binary translation.

The rest of the paper is organized as follows. Section 2 describes the proposed persistent code caching approach. Section 3 discusses some important implementation issues. Section 4 presents our experimental results. Section 5 discusses other related work. And Section 6 concludes our paper.

2 Persistent Code Caching - Our Approach

In general, DBT systems translate guest binary into host binary at the granularity of a *basic block* (or *block* for short), which is a sequence of guest instructions with only one entry and one exit. The generated host code

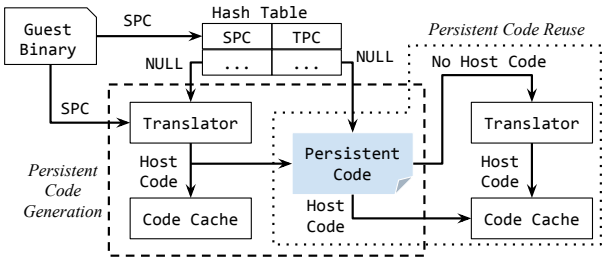

Figure 2: Persistent code caching for DBT systems.

is stored in an executable memory region, called *code cache*. The measured translation overhead in SPEC CINT2006 using *test* input is shown as *Block Translation* in Figure 1. After the translation of a guest block, the DBT system then transfers the execution to the generated host code in the code cache. During the execution in the code cache, a jump back to the translator/emulator is triggered when an untranslated block is encountered, and the translation is restarted.

In a DBT system, a *hash table* is used to manage the mapping from the *guest* program counter (PC) of the source binary, i.e., SPC, to the corresponding *host* PC of the translated code, i.e., TPC. Each time a new block is translated, the hash table is updated with the pair of SPC and TPC. Indirect branches and return instructions can look up the hash table to find out the TPC of the target block. A *jump stub* is used to chain translated code of two blocks connected via direct branch instruction to avoid the jump back to the translator, which is extremely time-consuming. The jump stub is initialized to jump back to the translator at first. After target block is translated, it is patched to jump to TPC of target block.

A hot path in the guest binary forms a *trace*, which contains several blocks with one entry and multiple exits. The number of blocks in a trace is the *size* of the trace. After a trace is formed, the blocks in this trace are re-translated and more aggressive optimizations can be applied to achieve a better performance. The translation/optimization overhead in this process for the SPEC CINT2006 benchmarks is shown as *Trace Translation* in Figure 1.

In our persistent code caching approach, translated host code for both basic blocks and traces are kept and reused across executions, i.e., they persist across executions. Thus, both block translation and trace translation overheads can be reduced. Figure 2 shows the framework and the work flow of our persistent code caching approach. It consists of two phases, *persistent code generation* and *persistent code reuse*.

In our approach, persistent code is organized in *entries*. As shown in Figure 2, after the translator finishes the translation of a basic block or trace, the generated host binary code and the related information such as guest binary code, relocation information and internal data structures are copied to a host memory region and form a new *persistent code entry*. The host memory region is called *persistent code memory* and is allocated at the start of the DBT system. At the end of the program execution, e.g., a guest `exit_group` Linux system call is emulated, all entries in the persistent code memory are flushed to the disk and stored in a *persistent code file*, which is used across different executions.

To reuse the persistent code generated in previous executions, the persistent code file is loaded into the memory at the start of the DBT system and installed into a two-level hash table, called *PHash* (persistent code hash table). The details of PHash is described in Section 2.2.

Before a guest block or trace is translated, PHash is looked up to check if there is a matching persistent code entry already. Here, *matching* means they have the same guest binary code. If a matched persistent code entry is found, the translator is bypassed and the translated host binary in this entry is copied to the code cache directly. Before the execution of the copied host binary, the required relocation and recovery are performed to ensure its correct execution. In this way, the host binary generated in one execution can be correctly reused in another execution, and the translation overhead is reduced or eliminated.

In addition, our persistent code caching approach supports *persistent code accumulation* across different executions. To accumulate persistent code, those basic blocks and traces that cannot find their matching entries in PHash will form new persistent code entries in persistent code memory. At the end of the current execution, these new persistent code entries are merged with existing persistent code entries to produce a new persistent code file.

Our persistent code caching approach can be integrated into most of the existing DBT systems. Only small changes to the translation work flow is required to generate, reuse, and accumulate persistent code. In the following sections, we describe in more details about our approach.

2.1 Persistent Code Generation

We form a new persistent code entry each time a basic block or a trace is translated. Each formed persistent code entry is comprised of three parts: *meta-data for the host binary code* (MDHBC), *guest binary code* (GBC), and *host binary code* (HBC). The organization of these three parts is shown in Figure 3.

MDHBC has two kinds of information: internal data structure and relocation information. The internal data structure is used to recover its corresponding

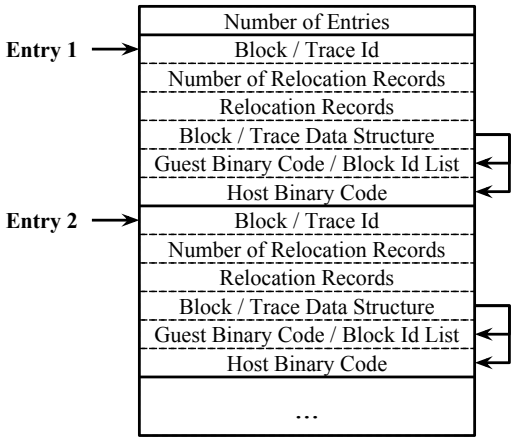

Figure 3: Detailed structure of persistent code file.

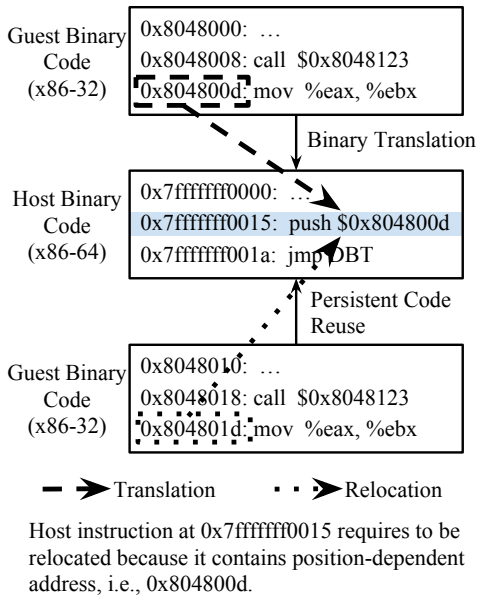

Figure 4: Host instruction relocation.

data structure to ensure the correct execution of the reused host binary code during the execution when the persistent code is used. The relocation information is used to convert the position-dependent addresses embedded in the host instructions. GBC is used to find and verify the matching persistent code entry. And HBC is the main part of persistent code reused across executions.

Meta Data for Host Binary Code (MDHBC). To generate MDHBC, we make a copy of the internal data structure of a block/trace when its translation is completed. Here, the internal data structure can vary between different DBT systems. In our prototype system, it is the data structure used by the DBT system (i.e., HQEMU) to represent each translated block/trace. A unique id is assigned to each copied data structure, called *block/trace id*. Typically, one of the most accessed items in this data structure is the offset of jump stubs in translated host binary code, which is used by the DBT system to chain two translated blocks/traces.

Another important part of MDHBC is the relocation information. For each block/trace, the relocation information is organized into records. Each relocation record is created for a host instruction whose operands contain a position-dependent address. Note, if a host instruction contains a guest position-dependent address and this address is also contained in the translated guest instruction, we do not create relocation record for such a host instruction because our guest binary code matching mechanism can filter out blocks/traces that include this guest instruction correctly if this position-dependent address is changed across executions. We use the example shown in Figure 4 to explain the information required for relocating host instructions.

In this example, the guest instruction *call* at guest address `0x8048008` is translated into two host instructions *push* `0x804800d` and *jmp* at host addresses `0x7fffffff0015` and `0x7fffffff001a`, respectively. If the translated code is reused across executions, the host instruction *push* `0x804800d` is required to be relocated because it contains guest position-dependent address `0x804800d`, which is the return address of the *call* instruction, or the address of the guest instruction following the call instruction. To relocate such a host instruction, two kinds of information are required. (1) *The location of the position-dependent address in code cache*, which is usually the start address of the relocated host instruction plus the size of its opcode. In this example, the location of `0x804800d` in code cache is `0x7fffffff0015` plus the size of *push* opcode, which is `1`. (2) *The correct address that should be put into this location*. In this example, the host code is reused by the basic block at address `0x8048010` with a *call* instruction at `0x8048018`. Thus, the correct address is `0x804801d`, which is the return address of this *call* instruction.

Each relocation record saves two items to keep the two required pieces of information. The first is the *offset* from the starting address of the translated host code for this block/trace to the location of the position-dependent address. In the above example, the offset is `0x16`. After the host code is copied from a persistent code entry to the code cache for reuse, this offset plus the starting address of the copied host code is the address for relocation. The second is a *source id* that identifies the source of the correct address. We use an enumerate type for the source id, called `IDType`. In the above example, the source id is `GUEST_NEXT_PC`, which means the correct address is

the address of the guest instruction following the last instruction in this block.

To facilitate the generation of relocation records, a unified API is provided for DBT developers:

```
extern int gen_reloc_record (int    offset,
                             IDType source,
                             BLOCK  *block,
                             TRACE  *trace);
```

This function generates a relocation record for `block` or `trace` in its MDHBC. Specifically, it should be called when a position-dependent address is inserted into a host instruction during binary translation.

Guest Binary Code. A copy of the guest binary code of a block is placed in the persistent code memory after its translation. The address information of this copy is added to the data structure of this block in MDHBC for future access. To limit the size of the generated persistent code, instead of copying the guest binary code of blocks in a trace, we only keep a list of block ids in the copied data structure of this trace in MDHBC.

Host Binary Code. After the translation of a block/trace, the translated host code is copied from code cache to the persistent code memory. Also, the address information of this copy needs to be saved into the data structure of the block/trace in MDHBC.

At the end of the execution, all formed entries in persistent code memory are flushed to disk to produce a persistent code file. The number of persistent code entries is stored at the beginning of the file, followed by all entries, as illustrated in Figure 3.

2.2 Persistent Code Reuse

After the generation of persistent code, it can be reused across executions. This section describes how to reuse persistent code.

At the beginning of an execution, the persistent code file is loaded from disk into memory. Each entry in the file is processed and installed into a two-level hash table, i.e., *PHash*. Figure 5 shows the structure of PHash (part of the second level hash tables is omitted due to space limitation). Relocation and data structure recovery will be done later after a matching persistent code entry is found for a block/trace.

We use L1-PHash and L2-PHash to represent the first and the second level hash tables in PHash. The hash keys of L1-PHash and L2-PHash are generated based on the guest binary code in each persistent code entry. If an entry corresponds to a trace, the guest binary of its first basic block is used to generate the hash key. Note, the property of the translated host binary code cannot be used as

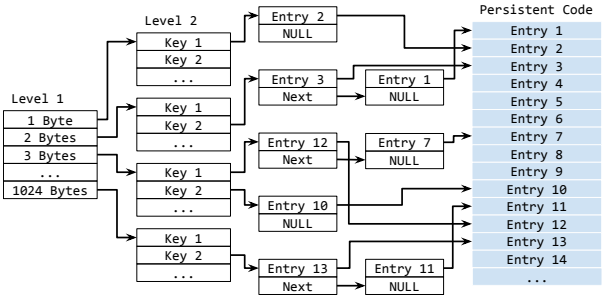

Figure 5: Two-level hash table used to look up persistent code for reuse.

the hash keys, because there is no translated host binary code but the original guest binary code of a block/trace when PHash is looked up to find out the matched entry for this block/trace.

Specifically, the hash key of L1-PHash is generated using the size of the guest binary code in bytes. The reason is that guest binary with very large basic blocks is very rare in practice, which can help to limit the size of L1-PHash. In our experiments, we found 1024 is enough for all guest applications evaluated. At the second level, several bytes in the guest binary are chosen to construct a byte combination. With this byte combination as input, a simple hash function is used to generate an integer in the range from 1 to 256, which is used as the hash key of L2-PHash. Compared to a single-level hash table, this two-level hash table can achieve comparable performance with a smaller overall table size for persistent code with a very large number of entries.

To limit the size of PHash, L2-PHash is allocated only when at least one persistent code entry hit it. In other words, L1-PHash is allocated first and initialized with `NULL`. When an entry hits a `NULL` in L1-PHash, a new L2-PHash is allocated and its address is used to update the corresponding `NULL` entry in L1-PHash. After filling all persistent code entries, PHash can be used in this execution without further update.

To find a matching persistent code entry for a basic block, its guest binary is used to look up PHash. The boundary of this basic block, i.e., the start and the end addresses of this basic block, should be identified first, The start address is usually available in existing DBT systems. However, the end address is unavailable until the basic block is being decoded, and a branch instruction is encountered or the maximum number of guest instructions for a basic block is reached [6, 13]. After this, the guest binary code of this basic block can be used to look up PHash. Algorithm 1 describes the details of such a lookup for a basic block.

First, the size of the guest binary code of this block is used to look up L1-PHash. Second, an hash key is

Algorithm 1: Persistent Code Reuse

Input: A basic block *bb*, with the start address and size of the guest binary code, *addr* and *size*
Output: Persistent code entry matching with *bb*, otherwise NULL

```
1  h ← PHash [size];
2  if h = NULL then
3      return NULL;
4  key ← hash_func (addr, size);
5  p ← h [key];
6  while p ≠ NULL do
7      if guest_binary_same (p, addr, size) then
8          do_host_code_copy (p, bb);
9          do_host_code_relocation (p, bb);
10         do_data_structure_recovery (p, bb);
11         return p;
12     p ← p → next;
13 return NULL;
```

calculated based on the guest binary code of this block using a simple hash function to look up L2-PHash. Last, the list pointed by the hit entry in L2-PHash is scanned. The guest binary code in each persistent code entry on this list is compared with the guest binary code of the input block until a matching entry is found. If there is no such matching entry, NULL is returned and the translator is re-invoked to perform the translation. The look-up procedure for a trace is similar to a basic block. The only difference is that the guest binary code of all basic blocks in the trace is compared to find out a matching entry.

After a matching persistent code entry is found, the host code in this entry is copied to code cache. Then the relocation records in this entry is applied to the copied host code. Note, the copied code is still unchained at this point. However, after we recover the internal data structure, block/trace chaining will be done automatically by the translator during the ensuing execution.

Global optimizations applied by DBT systems might complicate the above look-up process. For example, some global optimizations for the current basic block might be based on its predecessor and successor blocks. A typical example is the conditional code optimization for Intel x86-like ISAs that have side effects on condition codes in some instructions [19]. It can eliminate unnecessary conditional code generation in current block based on the definition and usage of conditional codes in its predecessor and successor blocks. In such cases, if the guest binary code of the predecessor or successor blocks is changed, the translated host binary code of current block cannot be reused.

To deal with this issue, the guest binary of all blocks that may affect the translation of the current block should be compared to ensure the consistency of its translated host code. This can be realized by two steps. First, in persistent code generation phase, blocks that affect the translation decisions of current block are saved to MD-HBC when a global optimization is applied. Second, during the above lookup process, the guest binary of *all* saved basic blocks, not just the current block, are compared to determine a matching persistent code entry for the current block.

2.3 Persistent Code Accumulation

Even with persistent code, a basic block or a trace may still need to be translated because it may not have a matching persistent code entry found. In our approach, these newly translated host code is accumulated for future reuse. To accumulate persistent code from multiple execution runs, we form new persistent code entries for newly translated blocks and traces in each execution. At the end of the execution, the newly formed persistent code entries are merged with existing entries to generate a new persistent code file. Section 4 discusses the effectiveness of persistent code accumulation across same and different guest applications.

3 Implementation

A prototype of our persistent code caching approach has been implemented in an existing retargetable DBT system, HQEMU [13], which is a retargetable DBT system based on QEMU [6]. HQMEU uses original translator in QEMU, i.e., Tiny Code Generator (TCG), to translate basic blocks. Moreover, LLVM JIT [14] is used to generate more efficient code for traces. To detect traces, HQEMU inserts a *profile stub* and a *predict stub* into host binary generated by TCG for each basic block. The profile stub is used to count the number of dynamic executions of this basic block. Once the counter reaches a preset threshold, the profile stub is disabled and the *predict stub* is enabled to form the trace.

After a trace is formed, HQEMU converts TCG intermediate representations (IR) of the basic blocks in this trace to LLVM IR. Then, an LLVM JIT engine takes the LLVM IR as input and generates host binary code on the fly. During this process, several LLVM optimization passes are applied to improve the efficiency of the generated host code. Additional translation/optimization overhead is also introduced due to the time-consuming LLVM optimizations applied. To mitigate such translation/optimization overhead, HQEMU spawns helper threads to perform translations/optimizations for traces. However, this solution can reduce the system throughput, which could be sensitive in a cloud environment.

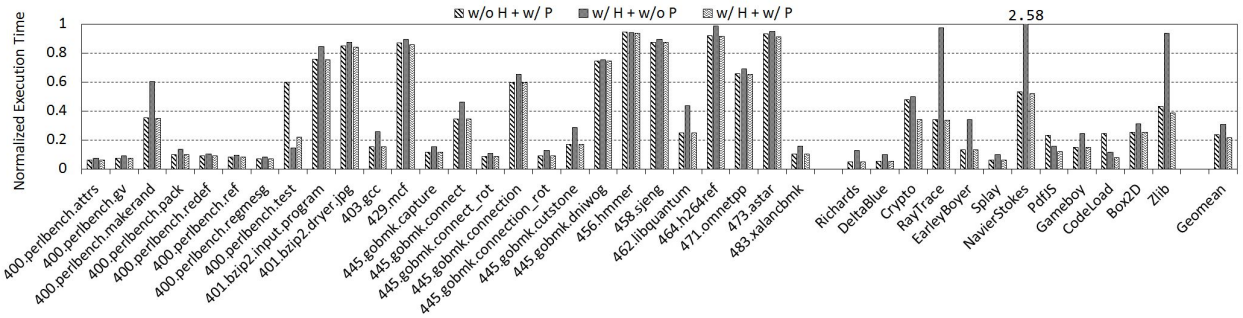

Figure 6: Performance improvement achieved by our persistent code caching approach. The base line is the execution without helper thread and persistent code, i.e., `w/o H + w/o P`.

The first issue that needs to be addressed in our implementation is *helper function calls*. For retargetability purpose, HQEMU (also in QEMU) leverages helper functions in the translator to emulate some complicated guest instructions, e.g., floating point and SSE instructions. These helper functions are called directly from code cache via position-dependent addresses. However, due to ASLR used by operating systems for security reasons, these function addresses are very likely to change across executions. Therefore, these function calls needs to be relocated for persistent code caching purpose. In our implementation, we use the unified API presented in Section 2.1 to generate relocation entries for helper function calls generated by TCG. For helper function calls generated by LLVM JIT, to simplify the implementation, we leverage redundant debugging meta data to acquire the relocation information from LLVM JIT.

Another special issue is about *global memory* and *constant memory pool* allocate by LLVM JIT during translating/optimizing traces. Global memory is a memory region used to save global variables that can be accessed by different traces. Constant memory pool is a local memory region allocated by LLVM JIT only for some specific traces, which is followed by the translated host code for these traces. For global memory, we save the initial value of each global variable into MDHBC once it is allocated by LLVM JIT. These values are used to recover the global memory in future executions. For constant memory pool, we adjust the starting address of translated host code for each trace to start from the beginning of the memory pool, if it exists. Thus, the constant memory can be saved and recovered along with the host code of traces.

Here, we focus on x86-32 to x86-64 cross-ISA translation. In the implementation, the profile stub and the predict stub are enabled and disabled as in original HQEMU to detect and form traces, whether or not the persistent code is available. Besides, each jump stub in host binary code copied from persistent code entries is set to jump back to the translator at first and patched later by the translator via block/trace chaining automatically. To guarantee the atomicity of the patch operation, which is a write to a 32 bit memory location, the address of the location should be 4-byte aligned on the host platform. Fortunately, HQEMU allocates code cache for each block/trace from an aligned address, which solves this issue naturally.

4 Experimental Study

Two types of guest applications, SPEC CINT2006 and SpiderMonkey [3] are employed to evaluate the performance. For SPEC CINT2006, the complete suite is included using *test* input. SpiderMonkey is a JavaScript engine written in C/C++ from Mozilla, which has been deployed in various Mozilla productions, including Firefox. The performance of SpiderMonkey is evaluated on Google Octane [2], which is a benchmark suite used to measure a JavaScript engine's performance by running a suite of test applications. Our experiments cover 12 of 15 applications in Octane, due to the failure of 3 applications when run on original HQEMU.

The experiment platform is equipped with Intel(R) Xeon CPU 16 cores with Hyper-threading enabled, 64G bytes memory, and Ubuntu-14.04 with Linux-3.13. To reduce the influence of random factors, each application is run three times and their arithmetic average is taken.

4.1 Performance Improvement

Figure 6 shows the performance improvement achieved by using our persistent code caching approach. The baseline is the execution without helper thread and persistent code, i.e., `w/o H + w/o P`. In this experiment, different numbers of helper threads were evaluated, but only the performance results with one helper thread are presented here because their results are all very similar. On

	P(capture)	P(connect)	P(connect_rot)	P(connection)	P(connection_rot)	P(cutstone)	P(dniwog)
capture	100%(4)	90.73%(2.5)	83.04%(2.4)	91.16%(2.3)	76.03%(2.6)	88.22%(2.3)	89.23%(2.2)
connect	66.36%(2.4)	100%(3.5)	76.03%(2.3)	88.58%(2)	71.06%(2.3)	67.25%(2.2)	86.33%(2)
connect_rot	73.7%(2.4)	91.9%(2.3)	100%(4)	91.91%(2.2)	83.15%(2.4)	74.38%(2.3)	90.86%(2.2)
connection	43.48%(2.2)	57.48%(2)	49.51%(2.2)	100%(3.5)	47.82%(2.3)	50.81%(2)	84.02%(2.1)
connection_rot	70.18%(2.6)	89.27%(2.3)	86.56%(2.4)	92.17%(4.2)	100%(4.2)	70.61%(2.6)	91.12%(2.4)
cutstone	73.97%(2.3)	76.89%(2.2)	70.31%(2.3)	89.40%(2)	64.25%(2.6)	100%(3.6)	88.65%(2)
dniwog	28.83%(2)	37.53%(2)	32.94%(2.1)	56.46%(2)	31.83%(2.2)	33.98%(1.9)	100%(3.3)

Table 1: Persistent code hit ratios with different inputs (the numbers in parentheses are the average numbers of basic blocks in the hit traces of persistent code).

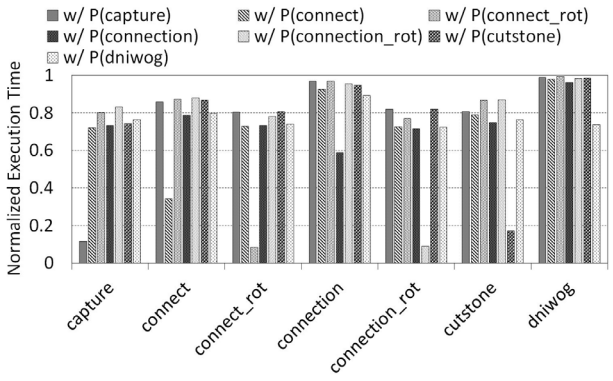

Figure 7: Effectiveness of persistent code for the same application with different inputs.

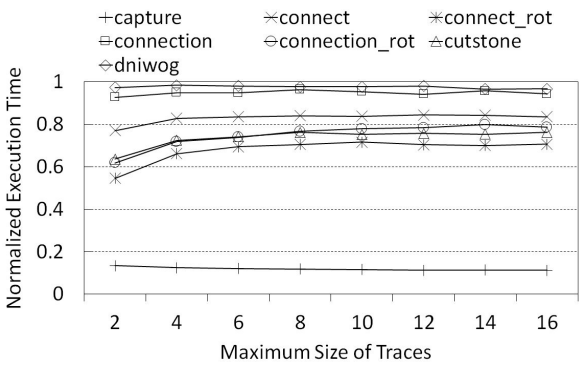

Figure 8: Performance sensitivity of persistent code caching on the maximum size of traces.

average, 76.4% and 9% performance improvements are achieved compared to the original DBT system without and with a helper thread enabled, respectively.

As shown in Figure 6, the performance is improved for all applications except `400.perlbench` with `test.pl` input. For this application, our approach achieves 40% performance improvement without helper threads, but introduces 5.5% overhead with helper threads. After further investigation, we have found that this input creates many child processes (around 72 of them) to perform many extremely short-running tests.

In the execution with both persistent code and helper threads, our implementation looks up persistent code before sending a trace translation request to helper threads. The performance overhead introduced by this look-up process might be relatively high for these extremely short-running tests because the generated persist code is very large (more details are presented in Section 4.3). However, in the execution with helper threads but without persistent code, a trace translation request is sent to helper threads once after the trace is formed, which introduce very little performance overhead. This is also the reason why it cannot achieve similar performance improvement in this application for `w/o H + w/ P` compared to `w/ H + w/o P`. Similar characteristics can be observed on several applications in Octane, e.g., `Crypto`, `PdfJS`, and `CodeLoad`, but they have different causes. More details are discussed in Section 4.2.

To study the efficiency of persistent code across different inputs for the same application, an experiment is performed on `445.gobmk`, which plays the game of Go, a game with simple rules but has extremely complicated strategies. There are seven inputs for this benchmark contained in the *test* input.

In this experiment, persistent code generated from one input is reused in a run using another input. The experimental result is illustrated in Figure 7, where `w/ P(X)` means persistent code generated from `X` input is used. Helper thread is disabled in this experiment. As shown in the figure, persistent code is still helpful to improve the performance across different inputs. This is because different inputs of the same application are very likely to share same blocks/traces. Table 1 shows the percentage of blocks/traces that can be found in persistent code, i.e., persistent code hit ratio. The numbers in the parenthesis are the average sizes of the hit traces in persistent code. This size is more than 3 for the same input. However, it is less than 3 for the different inputs. This means, the average size of traces shared across different inputs of the same application is usually not too large.

The default maximum size of a trace is 16 basic blocks in the original HQEMU. Another experiment is con-

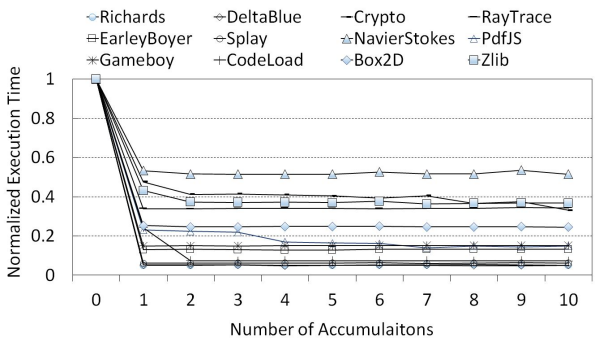

Figure 9: Effectiveness of persistent code accumulation from multiple runs of the *same* application.

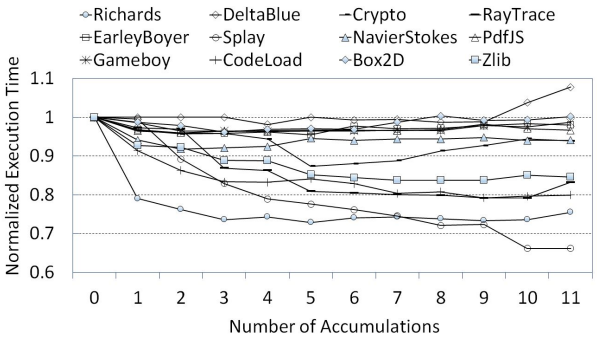

Figure 10: Effectiveness of persistent code accumulation from multiple runs of *different* applications.

ducted to study the performance sensitivity of persistent code on the size of traces across different inputs. In this experiment, the input `capture` is used for generating persistent code, and the maximum size of a trace is set between 2 and 16. The persistent code generated in each maximum trace size is applied to other inputs with the same maximum trace size. Due to the potential impact of the maximum size on the performance, various baselines of each input are used depending on the chosen maximum size of traces. As shown in Figure 8, the best performance is achieved when the maximum size of traces is 2. This shows why the average size of hit traces in the last experiment is less than 3. Hence, code persistence across different inputs prefers to use shorter traces (2 or 3 basic blocks).

4.2 Accumulation Effectiveness

The effectiveness of persistent code accumulation is evaluated on the same and different applications. Due to their different behavior, C/C++ and JavaScript applications in our benchmark suite were evaluated separately. Here, we only show the experimental results of JavaScript applications because C/C++ applications have similar results and our space is limited here.

Firstly, we evaluate the effectiveness of persistent code accumulation for the *same* application. In this experiment, the persistent code is accumulated from multiple runs for the same application. Each application is executed multiple times with previously generated persistent code accumulated and used in the later runs, if possible. Each time when a new persistent code is encountered, it is included for the later runs of the same application to evaluate the benefit of the accumulation.

Figure 9 shows the performance (normalized to the execution time without using persistent code) of each application using persistent code accumulated from one run up to ten runs. As shown in the figure, such persistent code accumulation is helpful to several applications, e.g., `Crypto`, `PdfJS`, and `CodeLoad`. However, the benefit of accumulation quickly diminishes after a small number of runs for most applications. The figure also shows that persistent code cannot achieve similar performance improvement for `w/o H + w/ P` compared to `w/ H + w/o P` because persistent code collected from one execution cannot fully cover the later executions.

Typically, there are two reasons for this phenomenon. (1) There is a chance to miss some traces due to the limited buffer size used for trace formation in HQEMU. These traces have opportunities to be formed and translated in the executions when persistent code is available. (2) The behavior of guest JIT engine is changed due to the performance improvement introduced by persistent code. In this situation, new guest binary is generated dynamically by the guest JIT engine that requires translation. Generally, the performance becomes stable after two or three accumulations for most applications shown in Figure 9 because no significant amount of new host code is generated beyond that point.

Next, we evaluate the effectiveness of persistent code accumulation from *different* applications. In this experiment, the persistent code is accumulated from the executions of all applications in a set of twelve applications except for one. The accumulated persistent code is then used by the excluded application. To evaluate the effectiveness of accumulation, the excluded application is run with the persistent code accumulated from one application up to eleven applications in the set. Without loss of generality, the order of the eleven applications chosen during the accumulation phase is random.

Figure 10 shows the results of this experiment. Initially, most applications can benefit from the accumulated code. However, as more persistent code is accumulated (from one to eleven), its performance benefit diminishes and even becomes harmful beyond some point for some applications. This indicates that persistent code accumulation from different applications is not always beneficial because large persistent code can introduce high

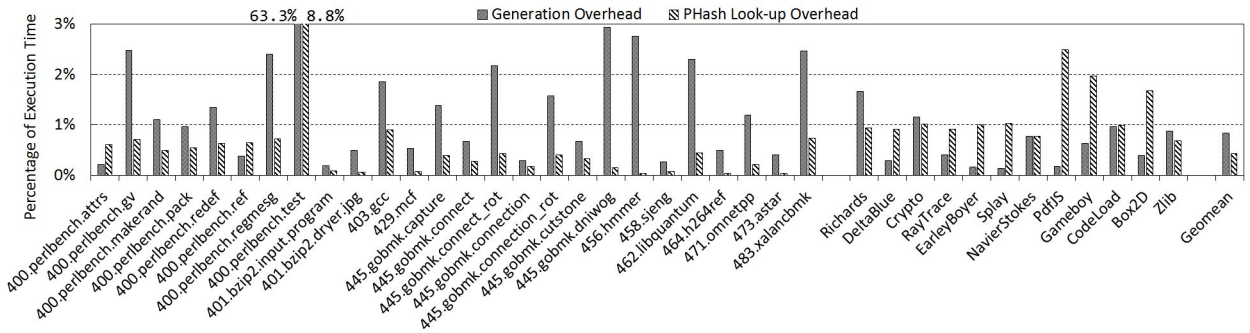

Figure 11: Performance overhead introduced by persistent code caching.

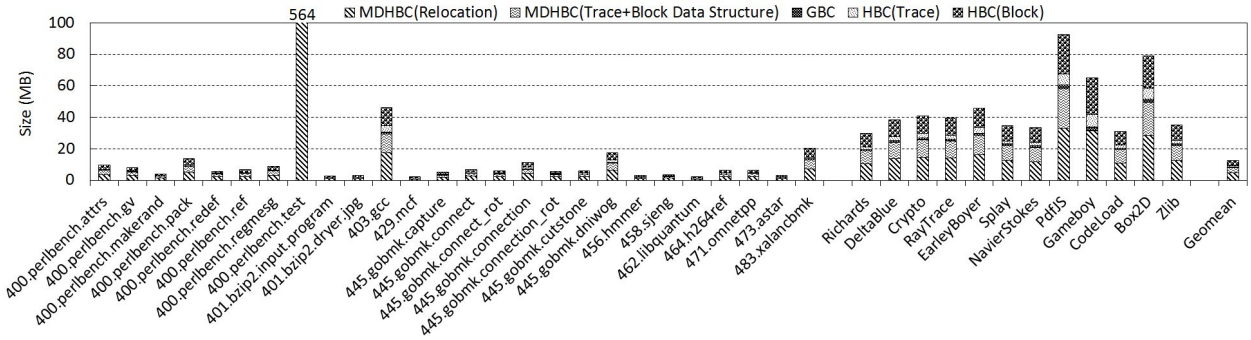

Figure 12: The sizes of different parts in persistent code.

overhead in looking up PHash. In our experiments, 5 is the turning point. Currently, there is no limitation on the size of accumulated persistent code. An effective persistent code management approach should limit the size to avoid this detrimental impact.

4.3 Performance Overhead and Code Size

Figure 11 shows the performance overhead introduced by persistent code generation and PHash lookup. The baseline is the original execution without helper thread and without persistent code. As shown in the figure, for all applications except `400.perlbench` with `test.pl` input, both kinds of overhead are less than 3%. As mentioned before, `400.perlbench` with `test.pl` has multiple child processes. Due to the persistent code accumulation from each child process, a huge generation overhead is introduced. After analyzing the execution, we found 95% of the performance overhead is caused by disk file operations. Another overhead is from PHash lookup because of the large accumulated persistent code. On average, less than 1% performance overhead is introduced by our persistent code caching approach.

Figure 12 shows the size of each part of the persistent code. For most applications, the total size is less than 40MB, except `400.perlbench` with `test.pl` input, which generates a very large persistent code due to persistent code accumulation. As shown in the figure, MDHBC is the largest part in persistent code, with 61% on average. In our current implementation, all information in the internal data structures of blocks/traces is kept for future recovery. However, some of information in these data structures is never used. There still is a good potential to further reduce the size of persistent code, and is part of our future work. The second largest part in persistent code is HBC(Block), which occupies 28% of the memory space, compared to 6.8% for HBC (Trace). This is because the host binary of traces is translated and optimized by LLVM JIT, while host binary for basic blocks is translated by TCG and is not optimized.

5 Related Work

The potential of reusing translated code across executions have been studied in previous work [12, 21, 22, 7]. Using DynamoRIO [4], which is a dynamic binary optimization system, the work [12] shows that many of the most heavily executed code traces in SPEC CPU2000 are similarly optimized during successive executions. This indicates the significant potential for leveraging the inter-execution persistence of translated/optimized code.

A mechanism of persistent code cache has been im-

plemented in [21, 22] for Pin [17], which is a dynamic binary instrumentation system. It takes traces and corresponding internal C++ data structures used by Pin as persistent code, which are generated when the original code cache in Pin becomes full or at the exit of the execution. To reuse persistent code across applications, shared libraries have to be loaded to the same memory address. Another persistent code caching scheme is proposed for process shared code cache [7]. It organizes guest applications and libraries into separate modules, and shares the code cache of modules among processes. To support relocatable guest applications, an offset of an instruction address from the start of the module is used to look up the persistent code of this module.

Compared to those schemes, our approach has three significant advantages. First, the guest binary code rather than the guest instruction address or offset is used to look up persistent code. It enables our approach to support dynamically generated code. Second, there is no restriction on the change of guest application and the libraries it depends on. If the guest application (or the libraries) is modified since last time when the persistent code was generated, e.g., upgraded to a newer version, our persistent code can still benefit from the unchanged part in guest binary code without any modification. Lastly, persistent code generated from a static-linked application, which contains application code and library code, is still helpful for other applications (static-linked or dynamic-linked) that use the same library code. However, it is difficult to realize this in existing mechanisms.

A similar scheme of guest binary matching, called guest binary verification, is also explored to reuse translated code in a single execution [16]. Different from the approach discussed in this paper, there is no relocation issue in a single execution. Also, it uses guest instruction address to discover persistent code, which is different from our approach using guest binary code.

There are also several schemes to optimize DBT systems [11, 26, 27, 10, 23]. Basically, these optimizations can cooperate with our persistent code caching approach to improve the performance of DBT systems.

6 Conclusion

This paper presents a general and practical persistent code caching framework to work within existing DBT systems. The proposed approach saves translated host code as persistent code and reuses it across executions to amortize or mitigate the translation overhead. Different from existing persistent code caching schemes, our proposed approach uses guest binary code to look up and verify persistent code. One significant benefit from this approach is to support persistent code caching for dynamically generated code, which is very popular in script languages. To find out a matching persistent code entry, a two-level hash table is designed to organize persistent code entries and speed up the look-up process. A prototype of this approach has been implemented in an existing retargetable DBT system. Experimental results on a set of benchmarks, including C/C++ and JavaScript, show that this approach can achieve 76.4% performance improvement on average compared to the original DBT system without helper threads to offload the overhead of dynamic binary translation, and 9% performance improvement on average over the same DBT system with helper threads when translated code reuse is combined with the help threads.

7 Acknowledgments

We are very grateful to Andy Tucker and the anonymous reviewers for their valuable feedback and comments. This work is supported in part by the National Science Foundation under the grant number CNS-1514444.

References

[1] ART and Dalvik. `https://source.android.com/devices/tech/dalvik/index.html`.

[2] Google Octane. `https://developers.google.com/octane`.

[3] Mozilla SpiderMonkey. `https://developer.mozilla.org/en-%US/docs/Mozilla/Projects/SpiderMonkey`.

[4] BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2000), PLDI '00, ACM, pp. 1–12.

[5] BARAZ, L., DEVOR, T., ETZION, O., GOLDENBERG, S., SKALETSKY, A., WANG, Y., AND ZEMACH, Y. IA-32 Execution Layer: A Two-phase Dynamic Translator Designed to Support IA-32 Applications on Itanium®-based Systems. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2003), MICRO 36, IEEE Computer Society, pp. 191–201.

[6] BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATC '05, USENIX Association, pp. 41–46.

[7] BRUENING, D., AND KIRIANSKY, V. Process-shared and Persistent Code Caches. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2008), VEE '08, ACM, pp. 61–70.

[8] CHEN, J.-Y., SHEN, B.-Y., OU, Q.-H., YANG, W., AND HSU, W.-C. Effective Code Discovery for

ARM/Thumb Mixed ISA Binaries in a Static Binary Translator. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (Piscataway, NJ, USA, 2013), CASES '13, IEEE Press, pp. 19:1–19:10.

[9] CHERNOFF, A., HERDEG, M., HOOKWAY, R., REEVE, C., RUBIN, N., TYE, T., YADAVALLI, S. B., AND YATES, J. FX!32: A Profile-Directed Binary Translator. *IEEE Micro 18*, 2 (Mar. 1998), 56–64.

[10] FEINER, P., BROWN, A. D., AND GOEL, A. Comprehensive Kernel Instrumentation via Dynamic Binary Translation. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 135–146.

[11] HAWKINS, B., DEMSKY, B., BRUENING, D., AND ZHAO, Q. Optimizing Binary Translation of Dynamically Generated Code. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2015), CGO '15, IEEE Computer Society, pp. 68–78.

[12] HAZELWOOD, K., AND SMITH, M. D. Characterizing Inter-Execution and Inter-Application Optimization Persistence. In *Workshop on Exploring the Trace Space for Dynamic Optimization Techniques* (2003), pp. 51–58.

[13] HONG, D.-Y., HSU, C.-C., YEW, P.-C., WU, J.-J., HSU, W.-C., LIU, P., WANG, C.-M., AND CHUNG, Y.-C. HQEMU: A Multi-threaded and Retargetable Dynamic Binary Translator on Multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (New York, NY, USA, 2012), CGO '12, ACM, pp. 104–113. *See* `http://itanium.iis.sinica.edu.tw/hqemu/`.

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

[15] LEE, G., PARK, H., HEO, S., CHANG, K.-A., LEE, H., AND KIM, H. Architecture-aware Automatic Computation Offload for Native Applications. In *Proceedings of the 48th International Symposium on Microarchitecture* (New York, NY, USA, 2015), MICRO-48, ACM, pp. 521–532.

[16] LI, J., ZHANG, P., AND ETZION, O. Module-aware Translation for Real-life Desktop Applications. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (New York, NY, USA, 2005), VEE '05, ACM, pp. 89–99.

[17] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 190–200.

[18] NETHERCOTE, N., AND SEWARD, J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 89–100.

[19] OTTONI, G., HARTIN, T., WEAVER, C., BRANDT, J., KUTTANNA, B., AND WANG, H. Harmonia: A transparent, efficient, and harmonious dynamic binary translator targeting the intel® architecture. In *Proceedings of the 8th ACM International Conference on Computing Frontiers* (New York, NY, USA, 2011), CF '11, ACM, pp. 26:1–26:10.

[20] RATANAWORABHAN, P., LIVSHITS, B., AND ZORN, B. G. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development* (Berkeley, CA, USA, 2010), WebApps'10, USENIX Association, pp. 3–14.

[21] REDDI, V. J., CONNORS, D., COHN, R., AND SMITH, M. D. Persistent Code Caching: Exploiting Code Reuse Across Executions and Applications. In *Proceedings of the International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2007), CGO '07, IEEE Computer Society, pp. 74–88.

[22] REDDI, V. J., CONNORS, D., AND COHN, R. S. Persistence in Dynamic Code Transformation Systems. *SIGARCH Comput. Archit. News 33*, 5 (Dec. 2005), 69–74.

[23] ROY, A., HAND, S., AND HARRIS, T. Hybrid binary rewriting for memory access instrumentation. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2011), VEE '11, ACM, pp. 227–238.

[24] SMITH, J., AND NAIR, R. *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, 2005.

[25] SRIDHAR, S., SHAPIRO, J. S., NORTHUP, E., AND BUNGALE, P. P. HDTrans: An Open Source, Low-level Dynamic Instrumentation System. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments* (New York, NY, USA, 2006), VEE '06, ACM, pp. 175–185.

[26] WANG, C., WU, Y., BORIN, E., HU, S., LIU, W., SAGER, D., NGAI, T.-F., AND FANG, J. Dynamic Parallelization of Single-threaded Binary Programs Using Speculative Slicing. In *Proceedings of the 23rd International Conference on Supercomputing* (New York, NY, USA, 2009), ICS '09, ACM, pp. 158–168.

[27] WANG, W., WU, C., BAI, T., WANG, Z., YUAN, X., AND CUI, H. A Pattern Translation Method for Flags in Binary Translation. *Journal of Computer Research and Development 51*, 10 (2014), 2336–2347.