Helper Function Inlining in Dynamic Binary Translation

Wenwen Wang
University of Georgia
Athens, GA, USA

Abstract

Dynamic binary translation (DBT) is the cornerstone of many important applications. Yet, it takes a tremendous effort to develop and maintain a real-world DBT system. To mitigate the engineering effort, *helper functions* are frequently employed during the development of a DBT system. Though helper functions greatly facilitate the DBT development, their adoption incurs substantial performance overhead due to the helper function calls. To solve this problem, this paper presents a novel approach to inline helper functions in DBT systems. The proposed inlining approach addresses several unique technical challenges. As a result, the performance overhead introduced by helper function calls can be reduced, and meanwhile, the benefits of helper functions for DBT development are not lost. We have implemented a prototype based on the proposed inlining approach using a popular DBT system, QEMU. Experimental results on the benchmark programs from the SPEC CPU 2017 benchmark suite show that an average of 1.2x performance speedup can be achieved. Moreover, the translation overhead introduced by inlining helper functions is negligible.

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

Keywords: Binary translation, Function inlining, Compiler optimization, QEMU

ACM Reference Format:
Wenwen Wang. 2021. Helper Function Inlining in Dynamic Binary Translation. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (CC '21), March 2–3, 2021, Virtual, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3446804.3446851

CC '21, March 2–3, 2021, Virtual, Republic of Korea

ACM ISBN 978-1-4503-8325-7/21/03…$15.00
https://doi.org/10.1145/3446804.3446851

1 Introduction

Dynamic binary translation (DBT) essentially translates machine instructions from a *guest* instruction set architecture (ISA) to a *host* ISA while preserving the semantics of the guest instructions. By executing the translated host instructions, DBT is able to emulate or enhance the functionality of the guest application on a host physical machine. As a key enabling technology, DBT has been extensively used in many important applications, such as whole program analysis [8], cross-architecture virtualization [7, 22], runtime optimization [11, 16], and mobile computation offloading [30]. Therefore, it is of paramount importance to continuously improve the efficiency of DBT systems.

However, in practice, it is not trivial to compose and maintain an efficient DBT system due to, for example, the substantial semantic differences between the guest and host ISAs. It is quite common for a DBT system to generate *dozens of* (or hundreds of in some cases) host instructions to emulate the semantic of a single somewhat complex guest instruction, which may just need to compute several condition codes [15]. Even worse, emerging CPU hardware features, e.g., single instruction multiple data (SIMD), hardware transactional memory (HTM), and trusted execution environment (TEE), frequently bring in diverse instruction set extensions to existing ISAs. This inevitably puts extra pressure on the development of DBT systems to keep pace with the fast-evolving CPU hardware.

Typically, the translation process in DBT is driven by prefabricated *translation rules*, each of which contains guest instructions and corresponding *semantically-equivalent* host instructions. In general, translation rules are created directly using the guest and host *assembly languages*. That is, DBT developers need to figure out what host assembly instructions are exactly required to emulate the semantics of the guest instruction(s) in each translation rule. Given the huge sizes of modern instruction sets, e.g., more than 3 000 instructions in the x86-64 ISA, it apparently poses a significant engineering effort to construct a *complete* set of translation rules. A recent learning-based approach points out an exciting new direction for DBT research [25]. However, it is still unclear whether this approach features the capability to learn translation rules for intricate guest instructions, e.g., floating-point and SIMD instructions.

As a matter of fact, to ease the development, debugging, and maintenance of DBT systems, DBT developers often rely

on *helper functions* to translate guest instructions with complex semantics and functionalities. More specifically, each of such guest instructions is emulated through a dedicated helper function, which takes as arguments the values specified by the source operands of the guest instruction and then conducts the computation according to the semantic of the guest instruction. The result of the computation, which is returned by the helper function, is finally used to update the destination operand of the guest instruction. This way, the translation process of the guest instruction is simplified to generate a host function call to invoke the helper function, regardless of how complicated the guest instruction is.

There are at least two obvious benefits from the adoption of helper functions in DBT systems. First, as mentioned above, the translation process, particularly for complicated guest instructions, is dramatically simplified. As a consequence, DBT developers can be radically emancipated from wrestling with the translation rules involving an extraordinary amount of host assembly instructions, which are quite unfriendly to novice DBT developers. Second, the debugging and maintenance of helper functions are much easier compared to complicated translation rules. This is because helper functions are usually written in high-level programming languages, e.g., C/C++, which are usually the same as the ones used to develop the DBT systems, rather than the obscure and hard-to-understand assembly languages. Given that the majority of DBT systems are sophisticated runtime systems, this benefit is very important for maintaining and extending the DBT systems.

Even though helper functions bring in the aforementioned benefits for DBT systems, they still face a fundamental issue. That is, invoking helper functions will incur substantial performance overhead and thus lead to a poor execution efficiency of the entire DBT system. There are two major sources of the performance overhead. First, additional host instructions need to be executed in order to invoke a helper function, including the instructions used to prepare the arguments of the helper function and the call instruction itself. Second, since the helper functions and the host binary code generated by the DBT system are located in different memory regions of the address space of the DBT system, invoking a helper function will unavoidably hurt the locality of the instruction cache. We will discuss more technical details about these two sources of performance overhead in Section 2. In a nutshell, at the same time of mitigating the engineering effort of DBT development, helper functions impose an unexpected and non-negligible performance penalty, which will hurdle the broad applications of DBT systems.

To address the performance issue introduced by helper functions, this paper proposes to *inline* the helper functions directly into the host binary code generated by the DBT system. This allows helper functions to stay in the same memory region as the generated host binary code and, therefore, they can be executed straightforwardly without the need of extra host instructions, e.g., the call instructions. In this way, the DBT system can not only benefit from the convenience and flexibility offered by helper functions, but also does not suffer from the poor performance efficiency.

Though function inlining is a pretty mature technique in traditional compilers, inlining helper functions in DBT systems renders a unique technical challenge. That is, the host binary code, which can be referred to as the "caller" function, is generated *on the fly* by the translator in the DBT system, while the helper functions, i.e., the callee functions, are compiled *statically* by the host native compiler along with the source code of the DBT system. In other words, the caller function and the callee functions to be inlined are compiled/generated separately by *different* compiler infrastructures. This is different from the classical scenario of inlining a function in a single source program, which can be achieved by the compiler that is used to compile the program.

To overcome this challenge, this paper presents a novel approach for helper function inlining in DBT systems. The proposed approach intelligently hides the low-level compilation details related to the host binary code and the helper functions, and therefore, the helper functions can be inlined transparently even across the boundary of compiler infrastructures. In particular, the proposed approach can automatically analyze the host assembly code compiled from the source code of the helper functions and transform the assembly code as necessary to enable the inlining capability. The assembly code is compiled to host binary code, which can be inlined into the code cache. We have implemented the proposed inlining approach in a prototype based on QEMU [3], which is a widely-used DBT system. Experimental results on all benchmarks from the SPEC CPU 2017 benchmark suite show that an average of 1.2x performance speedup can be achieved by the proposed inlining approach. This demonstrates the effectiveness of helper function inlining on enhancing the performance of DBT systems.

In summary, this paper makes the following contributions:

- We propose an effective approach to inline helper functions in DBT systems. The approach breaks the boundary of compiler facilities for helper function inlining. It enhances the performance efficiency of DBT, as well as preserving the benefits of helper functions.
- We implement a prototype based on the proposed inlining approach using a real-world DBT system, QEMU. The prototype is capable of inlining helper functions in QEMU. This demonstrates the effectiveness and practicability of the proposed inlining approach.
- We conduct comprehensive experiments on all benchmarks from the SPEC CPU 2017 benchmark suite. Experimental results show that the proposed inlining approach is able to achieve an average of 1.2x performance speedup compared to the original QEMU.

The remainder of this paper is organized as follows. Section 2 describes the background knowledge and the motivation to optimize helper function calls in DBT systems. Section 3 presents the technical details of the proposed helper function inlining approach. Section 4 shows the experimental results. Section 5 discusses related work, and Section 6 concludes this paper.

2 Background and Motivation

In this section, we describe the background knowledge about DBT and helper functions, as well as motivate this paper.

2.1 Dynamic Binary Translation (DBT)

To achieve cross-ISA binary translation, DBT firstly disassembles guest executable binary code into guest assembly instructions and then translates guest instructions into host instructions. The translation is directed by *translation rules*, which map guest instructions into semantically-equivalent host instructions. The granularity of the translation is a *basic block* (or *block* for short). Each block consists of a sequence of guest instructions, with only one entry and one exit. So a block may contain at most one branch instruction at the end of the block, and the execution of a block starts from the first instruction and ends at the last instruction. The translated host assembly code can be further assembled into binary code to be executed on a host physical machine.

To mitigate the performance overhead introduced by the translation process, a software *code cache* is typically employed in DBT to save the translated host binary code. This way, the host binary code can be reused later on without the need of re-translation, since a block is likely to be emulated multiple times in the same execution. A *hash table* is then established to map a block to its corresponding translated host binary code in the code cache. To look up the hash table, a hash key is typically created based on the address of the entry instruction of the block.

After the translation of a basic block is completed, the DBT system transfers the execution flow from the translator to the generated host binary code, so that the functionality of the block can be emulated by executing the generated host binary code. But, before executing the generated host binary code, the DBT system needs to save the execution context of the translator and restore the emulated guest machine state. This is because the generated host binary code needs to manipulate the emulated guest machine state, e.g., guest registers. This process is usually referred to as a *context switch* in DBT. Once the current block has been emulated, the DBT system will transfer the execution back to the translator from the code cache, through a context switch, to continue the translation and emulation of the next block.

Although each context switch may only comprise of several host instructions to save the current context and restore the target context, frequent context switches can still lead to significant performance overhead. To reduce the performance overhead caused by context switches, DBT systems rely on the *block chaining* technique to eliminate unnecessary context switches. Specifically, the translated host binary code of two successive blocks are chained together in code cache. This allows the emulation of the current block to be followed immediately by the emulation of the next block without going back to the translator, and thus removes redundant context switches.

2.2 Helper Functions

In general, the semantic of an assembly instruction is fairly easy to understand, even in a complex instruction set computer (CISC). Therefore, it is typically pretty straightforward to emulate the functionality of a guest assembly instruction using one or more host assembly instructions. For example, an x86-64 integer `add` instruction can be emulated simply by an AArch64 integer `add` instruction:

`add %rax, %rbx` ↦ `add x1, x1, x0`

Here, the guest `%rax` and `%rbx` registers are mapped to the host `x0` and `x1` registers, respectively.

However, apart from these simple instructions, there are many guest instructions that involve complicated operations, e.g., indirect branch, floating-point and SIMD instructions. These instructions may entail different features or semantics due to the inherent runtime nature of DBT systems and the diverse low-level implementation details on different hardware platforms. For example, we cannot naïvely translate a guest indirect branch instruction into a host indirect branch instruction because a guest branch target cannot directly serve as a valid host branch target. Instead, we need to look up the hash table using the guest branch target as the key to find out the address of the corresponding translated host binary code, which is the actual host branch target. In another example, the floating-point numbers and operations implemented on some hardware platforms may not conform rigorously to the IEEE 754 standard [1]. Therefore, to faithfully emulate a guest floating-point instruction on a host machine, it is mandatory to deal with potential semantic differences. These issues complicate the construction of translation rules with no doubt. Under this circumstance, helper functions emerge to facilitate the development of DBT systems by simplifying the translation rules.

In essence, a helper function is dedicated to emulate the complex semantic of a guest instruction. The values in the source operands of the guest instruction are passed to the helper function through the function arguments. After the semantic-equivalent computation in the helper function, the computation result is supplied via the return value to update the destination operation of the guest instruction. Figure 1 shows an example of helper function in a real-world DBT system, QEMU [3]. This helper function performs a fused

```
1 float64 helper_vfp_muladdd(float64 a,
2                            float64 b,
3                            float64 c)
4 {
5   union_float64 ua, ub, uc, ur, up;
6   ua.s = xa;
7   ub.s = xb;
8   uc.s = xc;
9   ...
10  if (float64_is_zero(ua.s) ||
11      float64_is_zero(ub.s) {
12    ...
13    ur.h = up.h + uc.h
14    ...
15  } else {
16    ...
17  }
18  ...
19  return ur.s
20 }
```

(a) A floating-point helper function

Guest (AArch64)

```
...
fmadd d0, d0, d1, d2
...
```

Translation

Host (x86-64)

```
...
/* context switch */
mov   0xc40(%rbp), %rdi // d0
mov   0xd40(%rbp), %rsi // d1
mov   0xe40(%rbp), %rdx // d2
call  *0xfb(%rip)
mov   %rax, 0xc40(%rbp) // d0
/* context switch */
...
```

Compiled Helper Function (x86-64)

```
<helper_vfp_muladdd>:
...
mov       %rdi, (%rsp)
movq      0x240a55(%rip), %xmm3
movsd     (%rsp), %xmm0
movapd    %xmm0, %xmm1
andpd     %xmm3, %xmm1
ucomisd   %xmm1, %xmm1
jp        d1e0c
...
```

(b) Translating a guest floating-point instruction into a host function call to the helper function. An indirect function call is used here to accommodate the large address offset between the call instruction in the code cache and the compiled binary code of the helper function.

Figure 1. An example of helper function. The source code and binary code of the helper function are simplified for illustration.

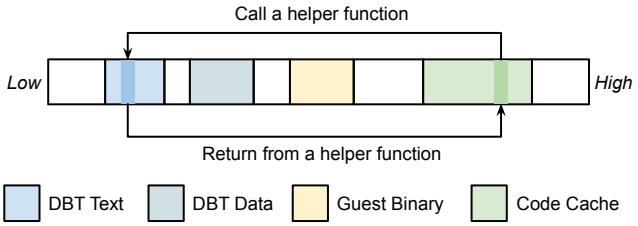

Figure 2. A sample virtual address space layout of a DBT system. The code cache and the compiled helper function binary (in DBT Text) are separated into two memory regions.

floating-point multiply-add operation on the three double-precision input values. Here, we omit the implementation details of the helper function, as it is out of the scope of this paper. The right side of Figure 1 shows the helper function-based translation for the guest AArch64 `fmadd` instruction. In the translated host x86-64 instructions, a call instruction is used to invoke the helper function. As we can see in this example, no matter how complex the semantic of the guest instruction is, the translation process can be simplified to generate necessary host instructions to invoke the target helper function. This substantially reduce the time and effort required to develop the translation rule at the assembly language level.

Motivation. From Figure 1, we can observe that, besides the call instruction, there are some other instructions in the generated host instruction sequence. These instructions are actually crucial to the success of the helper function invocation. For example, we need to pass the arguments to the helper function and receive the return value, through the host registers specified by the host application binary interface (ABI). Also, context switches are required to transfer the execution back and forth between the code cache and the compiled binary of the helper function. Given that these instructions are merely used to emulate a single guest instruction, they introduce non-negligible performance overhead on the execution efficiency of the entire DBT system.

Furthermore, the code cache and the compiled helper function binary code are located in different memory regions, as shown in Figure 2, because helper functions are typically compiled along with the source code of the DBT system and thus placed in the text section of the DBT system. That means frequent switches between them may put extra pressure on the instruction cache, leading to poor locality of the instruction cache. Therefore, it is urgent to enhance DBT performance by optimizing helper function calls.

3 Inlining Helper Functions in DBT

To reduce the performance overhead caused by invoking helper functions, we propose to inline helper functions into the translated host binary code. This way, we can eliminate redundant host instructions introduced by helper function calls. At the same time, the generated host binary code is more friendly to the instruction cache. Moreover, inlining helper functions would not destroy the benefits of helper functions to the development of DBT systems, as the inlining process is completely transparent to DBT developers.

Though the idea of helper function inlining is fairly intuitive, it is quite challenging to put it into practice. Different from the function inlining problem in traditional compilers, which has been studied thoroughly and deeply, helper function inlining in DBT faces a unique challenge, which mainly results from the inherent nature of DBT and helper functions. More specifically, in a DBT system, the helper function call instructions are generated *dynamically* by the translator, while the helper functions are compiled *statically* by a host native compiler, e.g., GCC, together with the source code of the DBT system. That means, the call instructions,

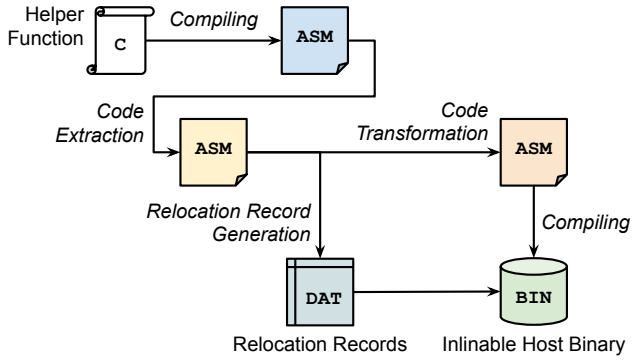

Figure 3. The high-level workflow of the proposed helper function inlining approach.

to be removed by the inlining, and the callee functions, to be inlined, are produced *separately* by different compiler facilities. Therefore, it is not clear how to leverage existing inlining techniques for helper function inlining in DBT, as they typically require the caller and callee functions to be compiled by the same compiler.

A straightforward solution for the above issue is to simply making a copy of the natively-compiled host binary code of a helper function into the code cache, when the translator generates a host call instruction to call the helper function. However, unfortunately, this does not work, because many host instructions compiled from the helper function by the native compiler cannot be executed directly in the code cache. For example, a memory access instruction using the program counter (PC) as the base address, i.e., the PC-relative addressing mode, may access an incorrect memory address due to the different PC value in the code cache. Besides, the execution environment provided in the code cache is typically restricted with the specific goal of emulating the guest code, and thus the native host environment is often hidden and inaccessible from the code cache. In other words, it is necessary to amend the compiled binary code of the helper function so that it can be inlined and executed in the code cache.

Instead, to overcome the aforementioned challenge, we propose a novel inlining approach, which seamlessly bridges the compilation gap between the dynamic translator and the static compiler. The approach automatically transforms the natively-compiled host code of a helper function into a form that is feasible to be smoothly inlined into the translated host binary code and executed safely in the code cache. The current building infrastructures and compilation tool chains of the DBT system are adopted to facilitate the whole inlining process. In addition, there is no need to annotate the source code of a helper function to enable the inlining optimization. Accordingly, DBT developers can freely modify existing helper functions and add new helper functions, and the proposed approach will inline the final version of the helper functions automatically and transparently.

Figure 3 shows the high-level workflow of the proposed inlining approach. First, we compile the source file of a helper function into the host assembly code using the existing compilation toolchain in the DBT system. Then, we extract the assembly instructions that correspond to this helper function. Since some of these instructions may not be executable in the code cache, we next transform them into instructions that can be executed in the code cache and generate relocation records if necessary, e.g., for instructions with the PC-relative addressing mode. Finally, we compile the assembly code into binary code, which can be combined together with the relocation records to produce a persistent file for helper function inlining. The actual inlining process is as simple as loading the file, copying the binary code into the code cache, and relocating the instructions. Next, we elaborate technical details of each step in this process.

3.1 Compiling Helper Functions

The first step of our inlining approach is to compile helper functions into host assembly code. We choose to start from the source code of the helper functions because this allows our approach to inherently support potential modifications to the helper functions in the future, e.g., fixing a software bug. To this end, we compile each source file that contains one or more helper functions using the default compiler and compilation options of the DBT system. We append the "`-S`" option, which is available in most modern compilers, e.g., GCC, ICC, and LLVM, to enforce the native host compiler to generate host assembly code for the compiled helper functions. After this step, we can obtain the host assembly code of each compiled helper function. However, this assembly code may not be executable in the code cache. So we proceed to the next step to transform the assembly code.

3.2 Transforming Assembly Code for Inlining

As mentioned before, natively-compiled host code is not inlinable due to the inconsistent execution environment provided by the code cache compared to the native host machine. To solve this issue, our approach automatically scans the host assembly code to identify potentially problematic instructions and transform them into instructions that can be correctly executed in the code cache.

In theory, when compiling a source program, a compiler may explore the whole instruction set and generate any instructions available in the instruction set when necessary. However, our experience shows that the assembly code generated for helper functions is primarily composed of several common types of host instructions. This is reasonable given that each helper function is developed for a specific and concrete purpose, i.e., emulating the semantic of the corresponding guest instruction. Therefore, this offers us an opportunity to classify the assembly instructions into multiple categories and handle each category individually.

More specifically, we find that there are three major types of host instructions compiled from a helper function in DBT.

(i) *Arithmetic and Logical Instructions.* These instructions perform basic arithmetic and logic operations, e.g., addition, bit-wise AND, comparison, and etc, on the values of the source operands and save the results to the destination operands or condition codes, e.g., the x86-64 RFLAGS register. Typically, these instructions use registers as operands, including general-purpose registers and SIMD registers.

(ii) *Memory Access Instructions.* An instruction in this category accesses a memory location specified by the memory operand of the instruction. On a reduced instruction set computer (RISC), the instruction may exhibit as either a `load` or a `store` instruction. But on a CISC, it is possible to fuse the memory access operation with an arithmetic operation, e.g., the x86-64 `add $0x1 (%rsp)` instruction adds the immediate value `0x1` to the stack location of `(%rsp)`.

(iii) *Branch Instructions.* These instructions are used to transfer the control flow of a program to the target address indicated by the operand. There are two types of branch instructions, direct and indirect. The key difference between them is that, the target address of a direct branch is known statically, while the target address of an indirect branch is generally not known until the branch is actually executed.

For the instructions in the first category, if they only have register operands, it is generally unnecessary to transform them as they can be executed properly in the code cache. But, in case they involve memory accesses, we will handle them in a way similar to the instructions in the second category. For the instructions in the third category, we distinguish function call and return instructions from regular branch instructions as we need to handle them in a special way.

Currently, indirect branches, including indirect jumps and indirect calls, are not supported yet, partially because we don't see any helper functions that have such instructions in reality. But on the other side, it is possible to extend our approach to support them by approximately identifying a set of potential targets of an indirect branch [5] and employing runtime checks to verify the targets. If a real target is not in the set, we can still fall back to a regular function call. More research work is still needed here to come up with an efficient solution for indirect branches.

Marking Memory Accesses. The key issue we need to resolve so that memory access instructions can be executed correctly in the code cache environment is to providing *correct* memory addresses that can be accessed from the code cache. As mentioned before, the memory access instructions generated by the native compiler may use the PC-relative addressing mode, which relies on a *fixed* immediate value encoded in an instruction to indicate the offset between the PC and the memory location to be accessed. Since the instruction will be moved to the code cache, the offset will be changed consequently and the address will be incorrect.

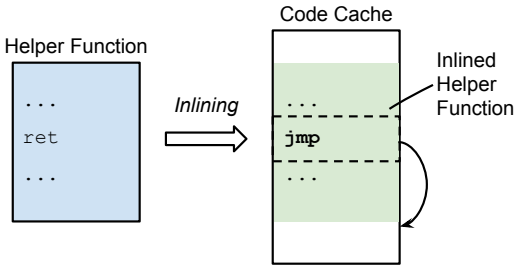

Figure 4. A return instruction in a helper functions is transformed to a direct branch instruction when the helper function is inlined into the code cache.

To solve this issue, we exhaustively scan the assembly code to collect all memory access instructions that may refer to an incorrect memory location when executed in the code cache. We then mark them as relocation required. More details about how to generate a relocation record will be discussed later on. When the helper function is inlined, the instructions will be relocated with correct memory addresses calculated according to corresponding values. This way, the memory operands will point to appropriate memory locations and the instructions can be executed properly in the code cache.

Handling Function Calls. Though function calls are pretty rare in helper functions, it is still possible for a helper function to invoke another function. To deal with this situation, one possible solution is to inline the callee function. However, this may not always be doable, because the callee function probably comes from an external library with only binary code available. It will involve complicated and likely inaccurate analyses if we inline a binary callee function. Therefore, to simplify our inlining approach, we firstly leverage host native compiler to inline as many callee functions as possible. This can eliminate the vast majority of calls to internal functions of the DBT system. For the remaining function calls to external libraries, we simply keep them without inlining.

Note that additional work is required here, as a function call in an inlined helper function will transfer the execution flow out from the code cache. Hence, we need to insert extra assembly instructions before and after each call instruction to fulfill the duty of context switches.

Updating Return Instructions. After a helper function is inlined, it will be executed directly in the code cache. As a result, it is unnecessary to execute a return instruction to exit from the helper function. In this sense, we need to modify return instructions in the helper function. To this end, we transform each return instruction into a direct branch instruction, which transfers the execution flow to the instruction that immediately follows the inlined helper function

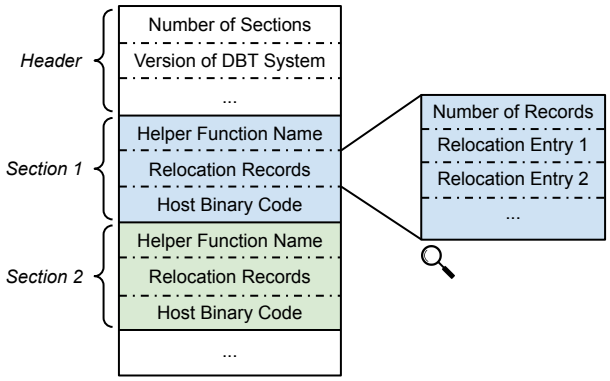

Figure 5. The internal structure of the `hfi` file used to save the metadata and binary code for helper function inlining.

instructions in the code cache, as depicted in Figure 4. This way, the following instructions will be executed automatically after the inlined helper function is completed.

3.3 Generating Relocation Records

Another key component of our inlining approach is to generate relocation records. This is necessary because, in most cases, the address information is not available until a helper function is actually inlined into the code cache at runtime. Hence, we generate a relocation record for each statically-unknown memory address. Note that although ELF (executable and linkable format) files generally contain relocation information, it does not contain the relocation information we need for helper function inlining. Take the PC-relative addressing mode as an example. ELF files do not need to generate relocation information for such instructions. This is because, when a relocatable ELF file is loaded to different addresses in the address space, the code and data in the ELF file are loaded together. That means, the offsets between the instructions and the accessed data do not change. However, for helper function inlining, only the instructions are copied, while the data is kept in the original location. Therefore, we need to generate additional relocation information to patch the "moved" instructions.

Every relocation record contains three items to provide the information required for the relocation. First, we need to know where the relocation will be applied. This is an address in the code cache where a relocated value will be placed into. We use the offset from the start of the helper function to the instruction to be relocated to indicate this address. To achieve this, we compile the assembly code of the helper function into binary code and then calculate the offset. Second, the type of the relocation record. Depending on the memory address to be relocated, different schemes are required to compute the correct value. For example, to relocate an instruction of the PC-relative addressing mode, we need to collect the new PC value when the instruction is copied into the code cache. Therefore, we use the relocation type `RELOC_PC_RELATIVE` to indicate this requirement. Finally, each relocation record has an optional item to supply additional information if necessary. This item is currently used to keep the original memory address before the relocation. But, it can be potentially extended to save other data required by different relocation types.

After the relocation records are generated for a helper function, we can combine them together with the compiled binary code of the helper function to produce an inlinable code for the helper function. Also, we merge the inlinable code of all helper functions into a persistent disk file to facilitate future inlining. We call such a file as an `hfi` (i.e., helper function inlining) file. Figure 5 shows the organization of the `hfi` file, which consists of one header and multiple sections. The header contains the metadata information, e.g., number of sections in this file, the version of the DBT system from which the file is generated, and etc. Each section corresponds to a helper function. It includes the name, the relocation records, and the compiled host binary code of the helper function. The `hfi` file will be shipped along with the DBT system for helper function inlining.

3.4 Inlining Helper Functions

Once the `hfi` file is generated, it can be used for helper function inlining. To this end, we load the `hfi` file to the memory at the start of the DBT system. To avoid potential consistency issues, we verify the DBT version number stored in the header of the `hfi` file to guarantee that it matches exactly with the version of the current DBT system. We then update the translation flow in the DBT system to query the `hfi` file each time when a helper function call needs to be generated. Specifically, if the query result shows that the helper function is included in the `hfi` file, it can be inlined to the code cache to enhance the performance. Otherwise, the original translation flow will be invoked to generate a helper function call as before. Note that we keep the original binary code of all helper functions in the compiled executable image of the DBT system, instead of removing them.

There are two steps to inline a helper function. First, we copy the host binary code of the helper function from the loaded `hfi` file to the code cache. Second, we relocate the copied binary code based on the relocation records specified in the section of this helper function. Here, it is worth pointing out that more call site-specific optimizations can be applied for polymorphic call sites. But, how to leverage such optimizations is out of the scope of this paper. We anticipate to exploit such opportunities in our future work. After these two steps, we can continue the original translation process.

4 Experimental Results

We have implemented a prototype of the proposed helper function inlining approach using QEMU [3] (version 5.1.0). The prototype is called QEMU-HFI, which takes AArch64 as

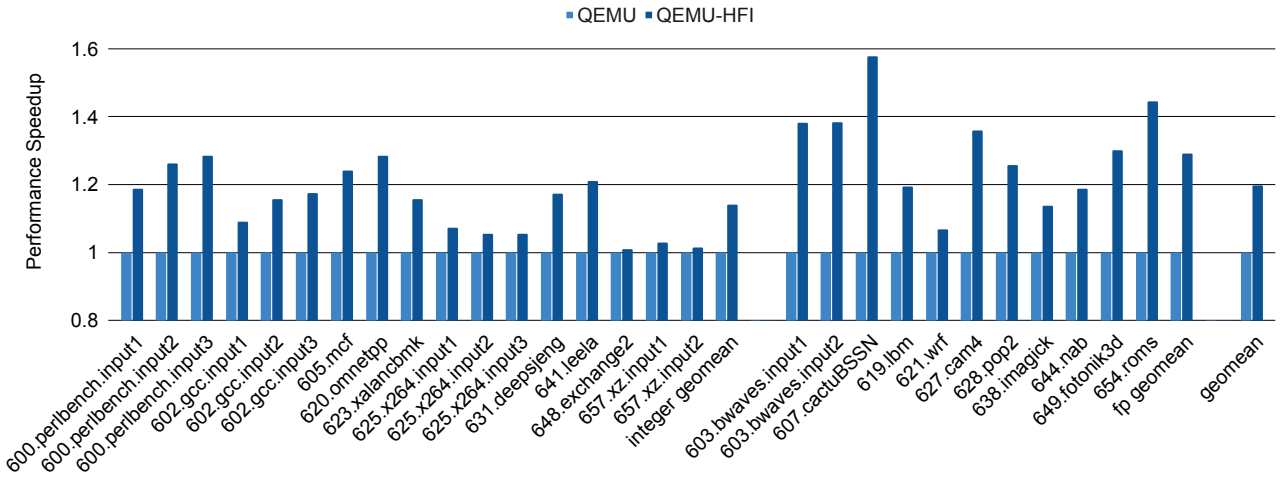

Figure 6. Performance speedup achieved by our helper function inlining approach for the SPEC CPU 2017 benchmark suite.

Table 1. Detailed configurations of our evaluation platform

	Configuration	
CPU	Intel Xeon E5-1620 v4 at 3.50GHz	
	# of Cores	4
	# of Threads	8
	Private L1 Data	32 KB
	Private L1 Instruction	32 KB
	Shared L2	256 KB
	Shared L3	10 MB
Memory	32 GB	
Operating System	Ubuntu 18.04.3 with Linux-4.15.0	

Table 2. Types of helper functions in QEMU.

Guest Instruction	Example
Indirect branch	helper_lookup_tb_ptr
Floating point	helper_vfp_addd
SIMD	helper_neon_padd_u8
Arithmetic and logical	helper_rbit64
Control	helper_exception_with_syndrome
Others	helper_get_cp_reg64

the guest ISA and x86-64 as the host ISA. QEMU employs the tiny code generator (TCG) to translate guest instructions. Specifically, a guest instruction is firstly translated into one or more TCG Ops, the intermediate representation of TCG, and then the TCG backend generates host binary code from TCG Ops. To implement QEMU-HFI, we intercept the translation from TCG Ops to host binary code, as each helper function call corresponds to a call operation in TCG Ops. By examining the first argument of the call operation we can figure out which helper function is called by this operation.

Next, we evaluate the performance of QEMU-HFI. To this end, we adopt the SPEC CPU 2017 benchmark suite [23], which is an industry standard benchmark suite and includes programs from various problem domains, such as artificial intelligence and discrete event simulation. Our evaluation covers all benchmarks in the suite. The detailed configurations of the experimental platform can be found in Table 1.

4.1 Performance Improvement

Figure 6 shows the performance improvement achieved by our helper function inlining approach. Here, the performance baseline is the original QEMU. As shown in the figure, our inlining approach can achieve performance improvement for *all* evaluated benchmarks. For some benchmarks, e.g., `607.cactuBSSN`, the performance speedup is as high as 1.58x. Overall, our inlining approach can achieve an average of 1.2x performance speedup. This demonstrates the capability of our inlining approach to reduce the performance overhead introduced by helper function calls in DBT systems.

Interestingly, Figure 6 also shows that our inlining approach achieves an average of 1.14x performance speedup for integer benchmarks, while an average of 1.29x performance speedup is reported for floating-point benchmarks. This is because floating-point benchmarks have a large amount of floating-point instructions, which are emulated using helper functions in QEMU. In contrast, most integer instructions can be emulated directly using native host instructions.

According to the emulated guest instructions, we can categorize helper functions in QEMU into different types, as shown in Table 2. We then further collect the distribution of the helper functions in different benchmarks. The result is presented in Figure 7. From this figure we can clearly see that indirect branches are the major source of helper functions in integer benchmarks (except `625.x264`, which has lots of SIMD instructions). Differently, floating-point instructions

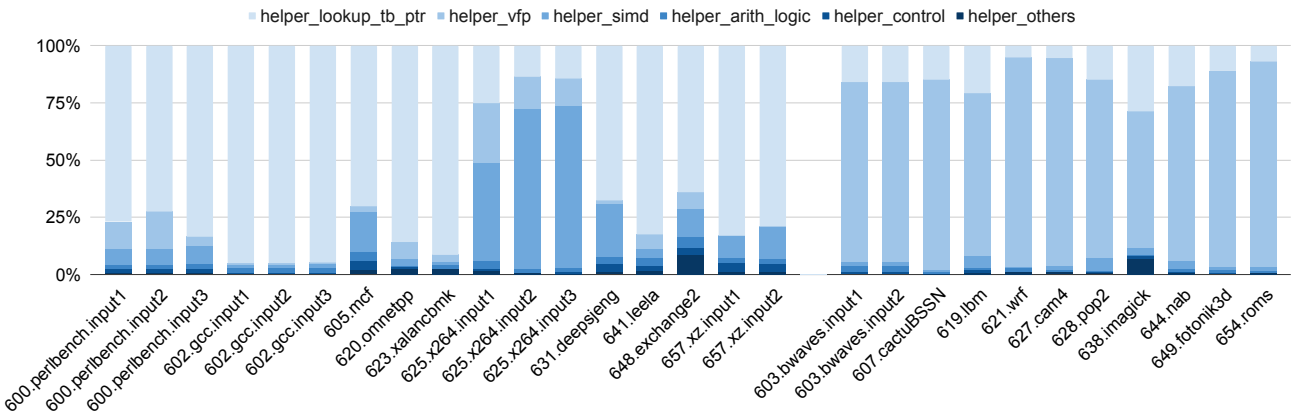

Figure 7. Distributions of helper functions in different programs of the SPEC CPU 2017 benchmark suite.

Table 3. Numbers of host instructions executed by QEMU and QEMU-HFI to emulate guest programs and percentages of host instructions reduced by QEMU-HFI.

	QEMU	QEMU-HFI	Reduced
600.perlbench.input1	9.87×10^{12}	8.04×10^{12}	18.51%
600.perlbench.input2	6.69×10^{12}	5.07×10^{12}	24.11%
600.perlbench.input3	5.99×10^{12}	4.61×10^{12}	23.09%
602.gcc.input1	8.64×10^{12}	7.89×10^{12}	8.70%
602.gcc.input2	4.24×10^{12}	3.43×10^{12}	19.29%
602.gcc.input3	4.11×10^{12}	3.31×10^{12}	19.47%
605.mcf	1.94×10^{13}	1.39×10^{13}	28.73%
620.omnetpp	1.17×10^{13}	7.90×10^{12}	32.50%
623.xalancbmk	1.23×10^{13}	1.09×10^{13}	11.52%
625.x264.input1	2.49×10^{12}	2.36×10^{12}	4.98%
625.x264.input2	9.64×10^{12}	9.29×10^{12}	3.68%
625.x264.input3	9.48×10^{12}	9.13×10^{12}	3.71%
631.deepsjeng	1.56×10^{13}	1.27×10^{13}	18.32%
641.leela	1.79×10^{13}	1.39×10^{13}	22.53%
648.exchange2	2.21×10^{13}	2.20×10^{13}	0.40%
657.xz.input1	2.88×10^{13}	2.79×10^{13}	3.26%
657.xz.input2	1.81×10^{13}	1.76×10^{13}	2.66%

account for a large portion of helper functions in floating-point benchmarks. This implies that to further enhance the execution efficiency of QEMU, it is necessary to investigate the translation process and emulation mechanism of indirect branches and floating-point instructions in integer and floating-point benchmarks, respectively.

Table 3 shows the numbers of host instructions dynamically executed by QEMU and QEMU-HFI to emulate each integer benchmark, and the percentage of host instructions reduced by QEMU-HFI. Here, we skip the results of floating-point benchmarks due to the space limitation. As depicted in the table, helper function inlining reduces the number of host instructions for all integer benchmarks, with a range from

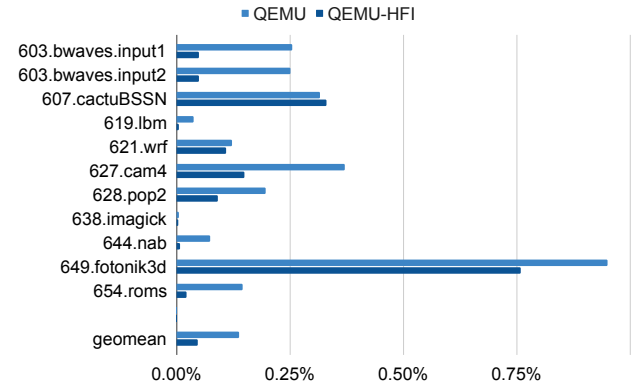

Figure 8. Miss ratios of instruction cache.

0.4% (i.e., `648.exchange2`) to 32.5% (i.e., `620.omnetpp`). This undoubtedly results from the fact that helper function inlining can remove host instructions generated by the translator related to invoking helper functions. Actually, the performance improvement attained by helper function inlining partly stem from the reduced host instructions.

We also use the hardware performance monitor unit (PMU) to study the impact of helper function inlining on the behavior of the instruction cache. Figure 8 shows the results of floating-point benchmarks. Because of the space limitation, we omit the results of integer benchmarks. As shown in the figure, helper function inlining reduces the instruction cache miss ratio for all floating-point benchmarks except `607.cactuBSSN`. It is unclear why this benchmark has a higher miss ration. One possible reason is that the inlined host binary code unexpectedly put additional pressure on the instruction cache. On average, we can observe that the instruction cache miss ratio of floating point benchmarks is reduced from 0.14% to 0.05%.

Table 4. Sizes of code caches in QEMU and QEMU-HFI with MB as the unit, and percentages increased by QEMU-HFI.

	QEMU	QEMU-HFI		QEMU	QEMU-HFI		QEMU	QEMU-HFI
600.perlbench.input1	12.13	12.25/0.97%	625.x264.input2	5.74	5.82/1.37%	619.lbm	1.15	1.18/2.38%
600.perlbench.input2	10.67	10.78/0.99%	625.x264.input3	5.62	5.69/1.30%	621.wrf	44.01	44.18/0.38%
600.perlbench.input3	11.25	11.35/0.94%	631.deepsjeng	2.45	2.47/0.84%	627.cam4	27.17	27.71/1.97%
602.gcc.input1	64.13	64.75/0.96%	641.leela	3.50	3.55/1.52%	628.pop2	18.30	18.54/1.31%
602.gcc.input2	65.73	66.36/0.96%	648.exchange2	3.10	3.13/0.93%	638.imagick	3.22	3.27/1.55%
602.gcc.input3	64.30	64.92/0.96%	657.xz.input1	2.13	2.15/1.16%	644.nab	2.46	2.50/1.60%
605.mcf	1.39	1.41/1.40%	657.xz.input2	2.11	2.13/1.05%	649.fotonik3d	4.50	4.60/2.26%
620.omnetpp	9.62	9.85/2.42%	603.bwaves.input1	2.47	2.54/2.79%	654.roms	7.89	8.05/2.03%
623.xalancbmk	13.40	13.75/2.60%	603.bwaves.input2	2.48	2.55/2.84%			
625.x264.input1	4.33	4.39/1.32%	607.cactuBSSN	15.68	15.93/1.61%	geomean		1.38%

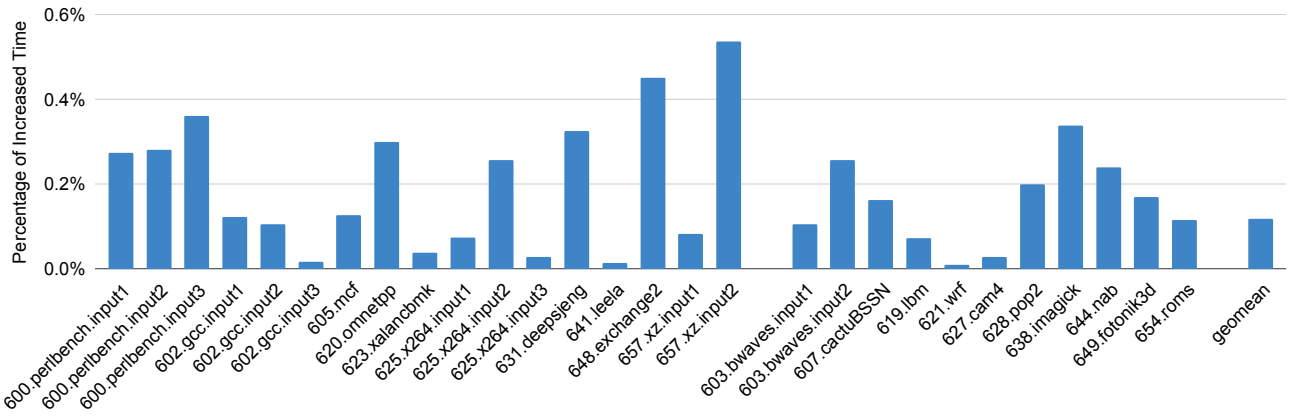

Figure 9. Translation overhead introduced by helper function inlining.

4.2 Code Cache Size

We next study the sizes of the code caches in QEMU and QEMU-HFI, as the increased size of the binary code is an important factor when traditional compilers determine whether it is worthwhile to inline a function. Table 4 shows the size of the code cache when emulating each guest application in QEMU and QEMU-HFI. As shown in the table, helper function inlining increases the size of the code cache as expected, because it expands the code cache to accommodate inlined helper function binaries. However, the percentage of the increased size is typically less than 3%, with an average of 1.38%. This means the code cache size increased by helper function inlining is generally acceptable. Indeed, our inlining approach is a *dynamic* approach, and thus inlines a helper function only if the helper function will be truly invoked.

4.3 Translation Overhead

Finally, we study the translation overhead caused by helper function inlining. Figure 9 shows the percentage of the execution time increased by copying the host binary code and relocating the copied binary code during the translation process. As shown in the figure, for all benchmarks, the increased time is less than 0.6%, and the average overhead is around 0.1%. This demonstrates that the translation overhead incurred by our inlining approach is negligible.

5 Related Work

Because of the special importance of DBT, plenty of work has been proposed to optimize the performance of DBT.

These optimizations mainly focus on two aspects of DBT systems. First, the performance overhead incurred by the translation process. To reduce the translation overhead, persistent code caching has been proposed to reuse the translated host binary code across different executions [4, 19, 28]. Second, the quality and the execution efficiency of the translated host binary code. For example, HERMES applies post-optimizations to generate high-quality host code [32]. Research work in [10] and [27] optimize DBT for dynamically-generated code. Similarly, research work in [26] presents a pattern translation approach for eflags across different architectures. A learning-based approach was recently proposed to automatically produce translation rules for DBT systems [13, 21, 25]. Given the pervasiveness of multi-core

CPUs, previous research work also investigates how to enhance DBT performance in multi-core and distributed environments [7, 29, 33]. Besides, a recent research work attempts to identify performance variance of a DBT system between different versions through performance regression testing techniques [31]. Broadly speaking, our helper function inlining approach falls into the second aspect, as it aims to enhance the quality of the translated host binary code. Different from previous research work, our inlining approach tackles the specific performance inefficiency issue caused by helper function calls in DBT systems. To the best of our knowledge, this issue was not studied by any existing DBT research work.

Stepanian et al. present an *incomplete* implementation to inline Java native function calls through JIT optimizations in JVM to achieve language interoperability [24]. Their approach keeps the intermediate representation (IR) when a native function is compiled by a native compiler. Since this IR cannot be recognized by the JVM, they create a *conversion engine* to dynamically transform it into the JVM IR for inlining. That is, they actually inline the IR of the native compiler, instead of the native binary code. This is obviously different from our helper function inlining approach. A significant limitation of the IR-based inlining approach is that the IR of the native compiler needs to be saved during the compilation process, and therefore, it closely ties the inlining approach to a specific native compiler. Besides, it takes a tremendous engineering effort to develop and maintain the conversion engine, which is infeasible for DBT systems. Perhaps, this is the reason why they only have an incomplete implementation. In contrast, our helper function inlining approach does not rely on the native compiler, as it works at the host assembly code level and inlines the binary code directly.

Piumarta el al. propose to generate more optimized code through selective inlining technique [17]. However, their approach mainly inlines basic blocks, while our work focuses on the entire helper functions. Also, they do not address the issue of generating relocation records to patch statically-compiled binary code. Moreover, it is not clear whether their approach needs to modify the original static compilation process to obtain the code pieces for inlining and how such code pieces are collected.

As a classical compiler optimization, function inlining can reduce the performance overhead caused by function calls [2, 5, 12]. It is widely available in almost all modern compiler systems [9, 14]. For example, GCC automatically turns on the function inlining optimization when "-O1" or a higher optimization level is specified. Though the idea of function inlining is simple and intuitive, there are still many heuristics to tune when applying the optimization in practice [6, 18, 20]. It is possible to leverage existing inlining tuning techniques to further enhance our helper function inlining approach for higher performance.

6 Conclusion

It is not an easy task to develop and maintain a DBT system in reality. Helper functions effectively mitigate the engineering effort in this process. But, the performance overhead introduced by helper functions is also substantial and needs to be resolved carefully. To this end, this paper presents a novel and practical approach to inline helper functions. This approach tackles the unique challenges imposed by the different compilation facilities used to generate the call instructions and to compile the helper functions. Specifically, the proposed approach transforms the compiled assembly code of a helper function and creates a relocation record whenever a memory address cannot be resolved statically. Experimental results on the SPEC CPU 2017 benchmark suite show that an average of 1.2x performance speedup can be achieved by the proposed inlining approach. Furthermore, the translation overhead incurred by inlining helper functions is negligible.

Acknowledgments

We are very grateful to the anonymous reviewers for their valuable feedback and comments. This work is supported in part by a faculty startup funding of the University of Georgia.

References

[1] 2019. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), 1–84. https://doi.org/10.1109/IEEESTD.2019.8766229

[2] Andrew Ayers, Richard Schooler, and Robert Gottlieb. 1997. Aggressive Inlining. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation* (Las Vegas, Nevada, USA) *(PLDI '97)*. Association for Computing Machinery, New York, NY, USA, 134–145. https://doi.org/10.1145/258915.258928

[3] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATC '05)*. USENIX, USA, 41–46.

[4] Derek Bruening and Vladimir Kiriansky. 2008. Process-Shared and Persistent Code Caches. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Association for Computing Machinery, New York, NY, USA, 61–70. https://doi.org/10.1145/1346256.1346265

[5] Brad Calder and Dirk Grunwald. 1994. Reducing Indirect Function Call Overhead in C++ Programs. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 397–408. https://doi.org/10.1145/174675.177973

[6] John Cavazos and Michael F. P. O'Boyle. 2005. Automatic Tuning of Inlining Heuristics. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*. IEEE Computer Society, USA, 14. https://doi.org/10.1109/SC.2005.14

[7] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. 2017. Cross-ISA Machine Emulation for Multicores. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO '17)*. IEEE Press, 210–220.

[8] Peter Feiner, Angela Demke Brown, and Ashvin Goel. 2012. Comprehensive Kernel Instrumentation via Dynamic Binary Translation. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/2150976.2150992

[9] GCC. 2020. Optimization Options. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[10] 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* (San Francisco, California) *(CGO '15)*. IEEE Computer Society, USA, 68–78.

[11] Shiliang Hu and James E. Smith. 2004. Using Dynamic Binary Translation to Fuse Dependent Instructions. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 213.

[12] Suresh Jagannathan and Andrew Wright. 1996. Flow-Directed Inlining. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(PLDI '96)*. Association for Computing Machinery, New York, NY, USA, 193–205. https://doi.org/10.1145/231379.231417

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

[14] LLVM. 2020. Inlining. https://clang.llvm.org/docs/analyzer/developer-docs/IPA.html.

[15] 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* (Ischia, Italy) *(CF '11)*. Association for Computing Machinery, New York, NY, USA, Article 26, 10 pages. https://doi.org/10.1145/2016604.2016635

[16] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO '19)*. IEEE Press, 2–14.

[17] Ian Piumarta and Fabio Riccardi. 1998. Optimizing Direct Threaded Code by Selective Inlining. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 291–300. https://doi.org/10.1145/277650.277743

[18] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Würthinger. 2019. An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilers. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO '19)*. IEEE Press, 164–179.

[19] 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, USA, 74–88. https://doi.org/10.1109/CGO.2007.29

[20] Douglas Simon, John Cavazos, Christian Wimmer, and Sameer Kulkarni. 2013. Automatic Construction of Inlining Heuristics Using Machine Learning. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '13)*. IEEE Computer Society, USA, 1–12. https://doi.org/10.1109/CGO.2013.6495004

[21] Changheng Song, Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Weihua Zhang. 2019. Unleashing the Power of Learning: An Enhanced Learning-Based Approach for Dynamic Binary Translation. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 77–89.

[22] Tom Spink, Harry Wagstaff, and Björn Franke. 2019. A Retargetable System-Level DBT Hypervisor. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 505–520. https://www.usenix.org/conference/atc19/presentation/spink

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

[24] Levon Stepanian, Angela Demke Brown, Allan Kielstra, Gita Koblents, and Kevin Stoodley. 2005. Inlining Java Native Calls at Runtime. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (Chicago, IL, USA) *(VEE '05)*. Association for Computing Machinery, New York, NY, USA, 121–131. https://doi.org/10.1145/1064979.1064997

[25] Wenwen Wang, Stephen McCamant, Antonia Zhai, and Pen-Chung Yew. 2018. Enhancing Cross-ISA DBT Through Automatically Learned Translation Rules. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 84–97. https://doi.org/10.1145/3173162.3177160

[26] 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), 2336–2347. http://crad.ict.ac.cn/EN/10.7544/issn1000-1239.2014.20130018

[27] Wenwen Wang, Jiacheng Wu, Xiaoli Gong, Tao Li, and Pen-Chung Yew. 2018. Improving Dynamically-Generated Code Performance on Dynamic Binary Translators. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Williamsburg, VA, USA) *(VEE '18)*. Association for Computing Machinery, New York, NY, USA, 17–30. https://doi.org/10.1145/3186411.3186413

[28] 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* (Denver, CO, USA) *(USENIX ATC '16)*. USENIX Association, USA, 591–603.

[29] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. 2020. Efficient and Scalable Cross-ISA Virtualization of Hardware Transactional Memory. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization* (San Diego, CA, USA) *(CGO '20)*. Association for Computing Machinery, New York, NY, USA, 107–120. https://doi.org/10.1145/3368826.3377919

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

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

[32] 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* (San Francisco, California) *(CGO '15)*. IEEE Computer Society, USA, 246–256.

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