Efficient and Retargetable Dynamic Binary Translation

Ding-Yong Hong

April 2013

Computer Science
National Tsing Hua University

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Abstract

Dynamic binary translation (DBT) is a core technology to many important applications such as system virtualization, dynamic binary instrumentation and security. However, there are several factors that often impede its performance: (1) emulation overhead; (2) translation overhead; (3) translated code quality. The issues also include its *retargetability* that supports guest applications from different instruction-set architectures (ISAs) to host machines also with different ISAs—an important feature to system virtualization.

An investigation of the JIT compiler design decisions reveals that the lightweight, template-based code emitter is inadequate for generating the optimal host instructions; the heavyweight aggressive optimizer causes too much translation overhead to the running applications. Emulation overheads from region transitions, helper function invocation and thread synchronization, also cause the impediments to building an efficient DBT system.

Addressing the dual issue of good translated code quality and low translation, we take advantage of the ubiquitous multicore platforms, and use a *multithreaded* approach to implement DBT. By running the translator and the dynamic binary optimizer on different cores with different threads, it can off-load the overhead incurred by DBT on the target applications; thus, afford DBT of more sophisticated optimization techniques as well as its retargetability. Using QEMU and LLVM as our building blocks, we demonstrated in a multi-threaded DBT prototype, called *HQEMU* (*Hybrid-QEMU*), that this framework can be beneficial to both short-running and long-running applications.

A study of the translation granularity reveals that considerable overhead is incurred from code region transitions. Two region formation approaches, HPM-based and software-based trace merging, are designed to improve existing trace selection algorithms. The novel HPM-based trace merging technique can detect and merge separated traces based on the information provided by the on-chip hardware HPM. The software-based region formation combines the potential separate traces early in the program execution, and is helpful for emulating short-running applications. Both approaches can result in the elimination of region transition overhead and improve the overall code performance significantly.

We also address the *performance scalability* issue of multi-threaded applications across ISAs. We identify two major impediments to performance scalability in QEMU: (1) coarse-grained locks used to protect shared data structures, and (2) inefficient emulation of atomic instructions across ISA's. And two techniques are proposed to mitigate those problems: using IBTC to avoid frequent accesses to locks, and lightweight memory transactions to emulate atomic instructions across ISAs.

Finally, a distributed DBT of the client/server model is proposed for embedded systems: a thin translator on each thin client and an aggressive optimizer on the server to service the optimization requests from thin clients. It successfully off-loads the optimization overhead of thin clients to the server. Moreover, the proposed asynchronous translation model can tolerate network disruption, and hide the optimization overhead and network latency.

Contents

List of Figures

List of Tables

Acknowledgments

This thesis would not be possible without the help of many people. First of all, I would like to thank my advisor, Professor Yeh-Ching Chung, for his support and guidance throughout my years of research. His encouragement and understanding is truly appreciated.

I would like to express my sincere thanks to Dr. Jan-Jan Wu, who is also my co-advisor at Academia Sinica, for the reviews and many invaluable advices of this thesis. It has been my pleasure to work with her and learn from her the enthusiasm for research. I also want to thank Professor Pen-Chung Yew and Wei-Chung Hsu. Their unique insights inspired many key ideas of this thesis, and they were always patient and pointed me in the right direction when I got stuck in my research. I am also grateful to Professor Pangfeng Liu and Dr. Chien-Min Wang for giving me helpful comments and suggestions to improve this research.

Many thanks to my colleagues at Academia Sinica, Chun-Chen Hsu and Chao-Rui Chang. I appreciate the amount of time they provided for the countless discussions. And also thanks to Jiazheng Zhou, Xuan-Yi Lin and other SSLAB members for that good time studying and doing research at NTHU.

Finally, I am heartily thankful to my family and wife, Rita Chen, for their continuing support, and always encouraging me to be more positive and optimistic on the road to complete my Ph.D. degree.

This dissertation includes the papers that have been previously accepted/submitted for publication as follows:

Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Yeh-Ching Chung, Pangfeng Liu and Chien-Min Wang. "HQEMU: A Multi-Threaded and Retargetable Dynamic Binary Translator on Multicores," in *IEEE/ACM International Symposium on Code Generation and Optimization*, San Jose, California, March 2012. (Chapter 3)

Ding-Yong Hong, Jan-Jan Wu, Pen-Chung Yew, Wei-Chung Hsu, Chun-Chen Hsu, Pangfeng Liu, Chien-Min Wang and Yeh-Ching Chung. "Efficient and Retargetable Dynamic Binary Translation on Multicores," accepted to appear in *IEEE Transactions on Parallel and Distributed Systems*, 2013. (Chapter 5)

Ding-Yong Hong, Chun-Chen Hsu, Chao-Rui Chang, Jan-Jan Wu, Pen-Chung Yew, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang and Yeh-Ching Chung. "A Dynamic Binary Translation System in a Client/Server Environment," *submitted*, 2013. (Chapter 6)

Chapter 1

Introduction

Virtualization has been gathering a lot of attention due to the advance of multi-core architecture and the trend of cloud computing. Virtual machine technology, the core of virtualization, is rapidly emerging as an integral component of future computer systems—it provides a layer of abstraction between the *guest* platform and the underlying *host* hardware or operating system—an important feature that enables many interesting applications, such as workload consolidation, resource provisioning, software delivery, security, availability, debugging, and simulation [82].

In recent years, the virtual machine designers mostly focus on a simple scenario—the computing environment is homogeneous, where guest and host platforms are compatible with each other. Under such scenario, developing the virtualization tools is easier, as well as the software deployment across the computing environment.

While computer designers continue to improve the performance of computing systems, reducing the power consumption is also critical across all kinds of computing. This trend has driven the evolution of computing systems, from mobile devices to supercomputers, to become heterogeneous computers. For example, more and more systems nowadays use both multi-core CPUs and many-thread GPUs or GPGPUs for higher performance and power efficiency; the concept of deploying servers of different computing capabilities (e.g. using both x86 and ARM servers) in a data center is also proposed. From a hardware perspective, the range of diverse computing resources can be on a chip, within a computer, or on a distributed network. And from a software perspective, an application can also has heterogeneous programming model, such as a CUDA program.

With such heterogeneity, including CPU architecture, accelerator types, memory models and I/O devices, the implementation of virtualization technology and its applications becomes more complicated. For instance, it becomes difficult to support virtual machine migration and workload consolidation among machines whose processors are not compatible (i.e. x86 and ARM), or to deploy a CUDA program on machines with no or incompatible GPUs. Hence, the virtual machine techniques need be re-designed to be aware of such heterogeneous system architecture.

Among these heterogeneity factors, the major impediment to virtualization is from the diversity of instruction-set architecture (ISA). One solution to enable virtualization across ISAs is through *dynamic binary translation*. Dynamic binary translators (DBTs) that emulate a *guest*

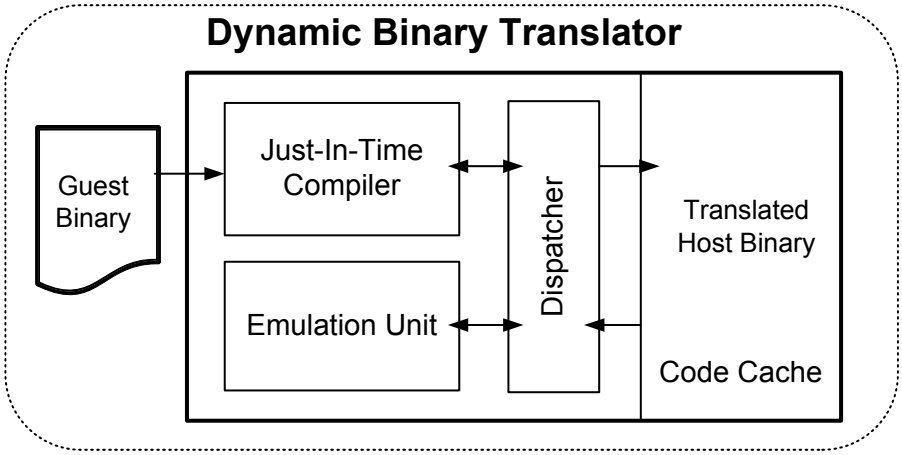


Figure 1.1: A general framework of a dynamic binary translation system.

binary executable code in one ISA on a *host* machine with a same or *different* ISA are gaining importance. It is because dynamic binary translation is the core technology of system virtualization, an often required system support in the new era of cloud and mobile computing. One advantage of dynamic binary translation is that it operates directly on program binaries with no need to recompile or access the source code of the guest application, providing the greatest flexibility to execute and manipulate legacy and proprietary code, which is important to virtualize unmodified guests. Dynamic binary translation is also widely used for many purposes, such as transparent performance optimization [7, 64, 79, 83], security monitoring [20, 56, 78], runtime profiling [66, 74], and application migration [9, 21, 26].

With the increasingly heterogeneous computing systems, the effective design of the DBT system is an important issue—not only for the overall performance but for its integration with the heterogeneous environment.

1.1 General Framework of a DBT System

A typical DBT system generally has four components – a just-in-time (JIT) compiler, an emulation unit, a dispatcher and a code cache. Figure 1.1 illustrates the interaction of these four components in a DBT system. When the translation starts, the *JIT compiler* fetches a segment of the guest binary code, performs optimization on the code, and translates it to binary code of the host ISA. The translated host code is cached in a software-based *code cache* to enable reuse and to amortize the overhead of code translation. The *emulation unit* provides special handlers for exceptions and interrupts, e.g. emulating I/O devices in full-system virtualization or handling system calls in process-mode emulation. The *dispatcher* coordinates the translation and execution of binary code. It determines whether to resume execution in the code cache, to perform exception handling, or to kick-start the JIT compiler if an untranslated guest code is encountered.

Based on the ISA of the guest binary code and the host platform, DBTs can be classified into three main categories – same-ISA DBT, cross-ISA DBT, and retargetable DBT. The guest ISA is identical to the host ISA in a same-ISA DBT, but are different in a cross-ISA DBT. These two DBT types can only conduct for a special binding of the guest and host ISA, and are *not* retargetable. On the contrary, a retargetable DBT can retarget among several different ISAs.

1.2 Limitation of Same-ISA and Cross-ISA DBTs

Same-ISA and cross-ISA DBTs, which are collectively referred to as the *dedicated* DBTs, are designed to run with specific guest and host ISA. Usually, the dedicated DBT systems assume the host architecture has the *same* or *richer* register set than the guest architecture, and maintain *special register mapping* between the guest architectural states and the host architectural states. Since the guest and host ISA are fixed, the translator can be carefully-architected to achieve good translation quality without incurring much overhead.

For example, it is a common technique for same-ISA DBTs to conduct the code translation simply by copying the guest binary code to the code cache. As a result, the guest registers are mapped to the *same* host registers and the translated code quality is almost the same as that of the original guest binary. The same-ISA DBT systems based on x86 architecture include Intel Pin [66], Intel StarDBT [86], and HP/MIT DynamoRIO [16]. Such same-ISA constraints can also be found in hardware-based virtualization tools, such as Xen [10] and KVM [59], although they are not DBTs.

As for the cross-ISA DBT systems, Intel IA-32 EL [9] and DEC FX!32 [21, 48] enable IA-32 applications to be executed on Itanium machines and Alpha machines, respectively. The host Itanium has 128 general purpose registers and Alpha has 32, which are much *richer* than the 8 general purpose registers in IA-32. Hence, the guest registers are allowed to be mapped to a dedicated set of the host registers without causing much register pressure. Since the guest and host architecture are also fixed, their translators can be tuned to use the best host instructions while translating the guest instructions.

With the constraint of fixed guest and host ISA, dedicated DBT systems can be well-tuned and make the emulation performance of the guest application close to or even better than the native speed[1]. However, such constraint also causes the lack of translation flexibility. It is hard for a dedicated DBT to perform adaptive SIMDization to any vector size or run 64-bit binary code on 32-bit machines. More importantly, it has too much limitation for dedicated DBTs to be integrated with the increasingly heterogeneous environment. Therefore, using dedicated DBT systems is restricted.

[1]Native speed is the performance of running the guest program, which is compiled to the ISA of the host architecture, on the host machine natively.

1.3 Challenges in Designing a Retargetable DBT System

Due to the limitation of dedicated DBT systems, it motivates the use of the *retargetable* DBT framework. The concept is to have a *single* DBT framework to take on application binaries from *several different ISAs* and retarget them to host machines with *different ISAs*. Using a common intermediate representation (IR) in the translator is an effective approach to achieve retargetability. There are several factors that often impede the performance of a DBT system: (1) emulation overhead before translation; (2) translation and optimization overhead, and (3) translated code quality. The performance scalability is also an issue when multi-threaded applications are emulated. Moreover, *retargetablity* of a DBT system is also an important requirement. The difficulty is on the development of a retargetable framework by which dynamic binary translators for different ISAs can be built with ease. This requirement imposes additional constraints on the structure of a DBT and, thus, additional overheads.

The translated code quality is particularly important because it directly impacts the performance of the emulated application. An effective translator design should emit high-quality host codes, but exert low overhead on the running applications while making code translation. The requirements of low translation overhead and high-quality codes are often in conflict with each other, especially for a retargetable DBT. The problem is that the IR usually provides a small set of operation codes—much smaller than in most existing ISAs. A guest instruction is very likely to be translated to multiple operation codes, especially for specialized or complex instructions; the semantics of the original instruction may also be lost after translation. The challenge is on the translator to select the host instructions representing the intermediate operation codes. The translator could apply many aggressive optimizations, such as peephole optimization and sophisticated pattern matching for instruction selection, to choose the optimal host instructions, however, incur high translation overhead, or have low optimization overhead but sacrifice the code quality.

To find a good balance, the designer needs to select optimization schemes that are highly cost effective. This approach, however, limits many optimization opportunities because it is very difficult to find sufficient optimization schemes that meet both low overhead and high translation quality. Hence, designing a retargetable DBT with both high translated code quality and low translation overhead is challenging.

Unlike dedicated DBTs, it is not easy for a retargetable DBT to have a fixed register mapping between all supported guest and host processor architecture. Instead, the guest processor is virtualized by a software structure in the memory. The emulation is completed via operating the virtual processor on behalf of the execution of guest instructions. During the emulation, there can be frequent loads and stores of the virtual processor registers—a problem of the emulation overhead. More details of such emulation overhead are discussed in Chapter 2.3.

The overhead of a DBT becomes more critical to the overall performance when it comes to translating multi-threaded applications. Severe contentions on shared data structures in a DBT can incur substantial synchronization overhead when multiple application threads are being

translated simultaneously. Moreover, the emulation of atomic instructions, which are required to implement synchronization primitives in multithreaded guest applications, becomes more challenging not only for its correctness but for its efficiency as well. The overhead of synchronization could result from poor implementation of atomic regions when emulating guest atomic instructions. As a large number of threads needs to be translated and optimized at runtime, any inefficiency in handling globally shared data structures and the emulation of atomic instructions in a DBT could be amplified many times over. Preliminary results (shown in Chapter 2) using the PARSEC benchmark suite show that such large penalties could nullify the benefit of parallel execution in a dynamic binary translation environment, and could even render it to become slower than running it sequentially.

Considering the embedded systems, the performance of the translated binaries on embedded devices can significantly affect the energy consumption because it is directly linked to the number of instructions executed and the overall execution time of the translated code. Even though the capability of today's embedded devices will continue to grow, the concern over translation efficiency and energy consumption will put more constraints on a DBT for embedded devices, in particular, for *thin clients* than that for servers—it poses another challenge to design an efficient and retargetable DBT for embedded systems.

1.4 Research Overview

The goal of this dissertation is to build an efficient and retargetable DBT framework that can achieve *high-quality code translation*, *low overhead* and *good scalability*. Until now, few DBTs are designed for retargetability [12, 85, 90]. Among these DBTs, QEMU [12] is a state-of-the-art and retargetable DBT system that enables both full-system virtualization and process-level emulation. It can translate binary applications from several guest architecture, such as x86, PowerPC, ARM and SPARC, on popular host machines, such as x86, PowerPC, ARM, SPARC, Alpha and MIPS. It also has been widely used in many applications. Thus, this motivates us using QEMU as the base infrastructure for our research. Among the two translation modes in QEMU, this research only focuses on *process-level emulation*, however, the proposed methods can also be applied to full-system virtualization. Chapter 2 provides an overview of the QEMU dynamic binary translator and identifies its performance problems.

Trying to solve the performance bottlenecks of QEMU, Chapter 3 introduces a multithreaded and retargetable DBT framework, called *HQEMU*. The framework uses QEMU as its frontend for fast binary code emulation and translation. However, QEMU lacks a sophisticated optimizer to generate more efficient code. To this, LLVM [60], also a popular compiler with sophisticated compiler optimization, is used as the backend, together with a dynamic optimizer to dynamically improve code for higher performance. The framework also takes advantage of the multicore resources and leverages multithreading itself, thus, most of the translation overheads can be off-loaded to other cores. With the hybrid QEMU (frontend) + LLVM (backend) approach, we successfully addressed the dual issue of good translated code quality and low translation overhead on the target applications.

For the issue of emulation overhead, the problem is caused from the translation granularity of the translator. Chapter 3 and 4 investigate the translation granularity from a basic block, a trace to a region. Several new trace/region formation and code optimization techniques are also developed to eliminate the overhead as well as to generate more efficient code.

Chapter 5 tackles the performance scalability issues in translating multi-threaded applications across ISAs. We propose a technique to cache frequently-used shared resources in thread-private data structures. Such caching techniques effectively mitigate the contention on shared data structures. For emulation of atomic instructions, instead of using expensive locking mechanisms provided by the system library and operating system, we use a lightweight approach, i.e. using atomic primitives provided by the host architecture, to achieve efficient emulation of guest atomic instructions.

In Chapter 6, we look at the design issues of a retargetable DBT framework for embedded devices. The DBT could not afford to have aggressive optimizer on *thin clients*. To this, a distributed DBT framework based on the client/server model is proposed—it consists of two dynamic binary translators: an aggressive dynamic binary optimizer on the server to service the optimization requests from thin clients, and a *thin DBT* on each thin client that performs lightweight binary translation and basic emulation functions for its own. In addition, an asynchronous communication scheme is applied to mitigate the translation/optimization overhead and network latency, and allows the binary translation to tolerate network disruption and outage for optimization services on a server.

Chapter 7 presents several DBT systems which are based on the multithreaded and client/server model, and problems of existing trace formation methods. Finally, Chapter 8 concludes this dissertation and discusses the future work.

The main contributions of this dissertation are as follows:

1. A proposal of the multi-threaded retargetable DBT on multicores that achieves low translation overhead and good translated code quality.
2. A proposal of the client/server-based distributed DBT that mitigates the optimization overhead and network latency, and allows to tolerate network disruption.
3. A proposal of the two-level IR conversion approach which simplifies the engineering efforts tremendously.
4. An evaluation of the novel hardware-assisted trace combination technique to improve existing trace selection algorithms.
5. A detailed evaluation of translated code quality, translation granularity, optimization and emulation overhead, and performance scalability using well-known benchmarks.
6. Implementation details of integrating QEMU and LLVM for designing an efficient and retargetable DBT system.

Chapter 2

Overview of Retargetable Dynamic Binary Translation

This chapter provides an overview of a typical retargetable dynamic binary translation system. We first introduce the two key techniques to make ISA virtualization retargetable – *guest CPU virtualization* and *intermediate representation*. After studying the structures and design decisions of a retargetable DBT system, we will realize how they impact the emulation overhead, translated code quality, and performance scalability.

The features and gathered results in this chapter are based on the QEMU[1] dynamic binary translator, as it is the base infrastructure for our research. These features and issues presented by QEMU are also representative for other retargetable DBT systems.

2.1 Overview of the Retargetable DBT: QEMU

QEMU is a state-of-the-art and retargetable DBT system that can conduct translation with several guest and host ISAs, such as x86, PowerPC, ARM and SPARC, etc. Figure 2.1 illustrates the execution flow of QEMU. As the figure shows, the emulation starts by entering a dispatching mode. During the dispatching phase, the dispatcher gets the next program address and determines if the host code pointed by this address is already translated in the code cache. A hash table, which maps guest program addresses to the memory locations of their corresponding translated code in the code cache, is used to perform the lookup. Upon a hash table hit, the execution jumps to the previously translated host code in the code cache. If it is a miss, the dispatcher kick-starts the JIT compiler to translate the untranslated guest code segment.

Right before the translation process, the JIT compiler needs to decide a granularity to fetch and translate the guest code. The known JIT systems choose different translation granularities from a basic block [50], a trace [11, 16, 42], a treegion [33, 35], to a whole procedure [75]. QEMU translates the guest code binary one *basic block* at a time. When a block of the guest binary is fetched and translated to intermediate code (will be discussed in Section 2.2.2), the

[1] This research is based on QEMU version 0.13.

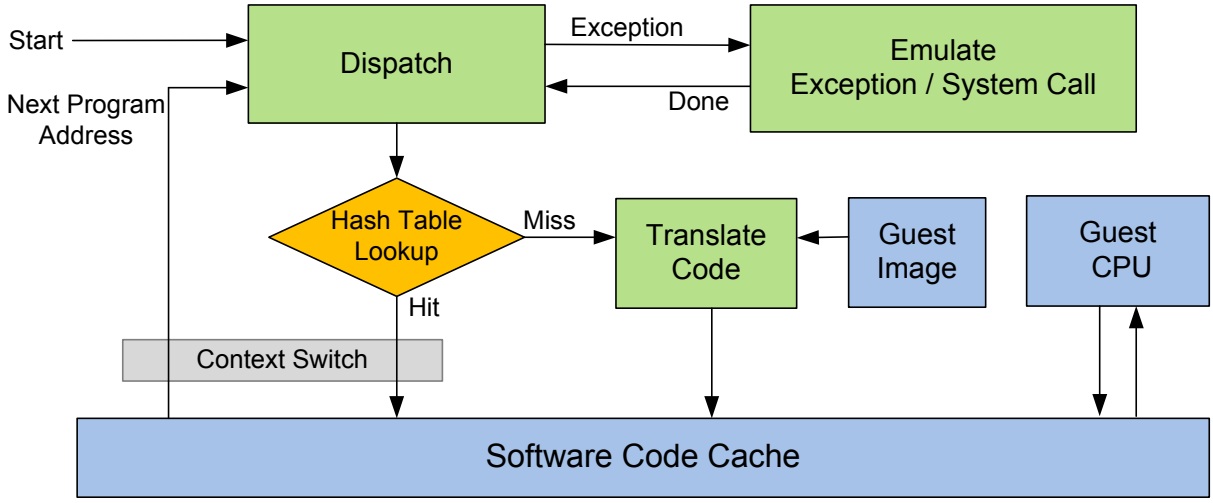


Figure 2.1: Control flow of the QEMU dynamic binary translator.

intermediate code is improved with two simple optimization passes: *register liveness analysis* and *store forwarding optimization*. *Dead code elimination* is also done as a by-product of these two optimization passes. Finally, the intermediate code is translated to the host binary in the software code cache, an entry is added into the hash table, and the emulation continues with the translated code.

The whole emulation process is composed of two parts—the execution within the code cache and the execution by the DBT itself (outside the code cache). Control transfers between the translated code and the dispatcher when the execution enters and leaves the code cache. This process is referred to as *context switch*. Similar to the context switch in an operating system, program contexts need to be saved and restored when the control is switched so that execution can be resumed from the same point at a later time. Two situations the execution transits from the code cache to the dispatcher: (1) determining whether the next instruction to execute is present in the code cache, and (2) handling exception or interrupt, which is then dispatched to the emulation unit for special handling.

2.2 Retargetability

In the following two subsections, we will discuss the key techniques for retargetable ISA virtualization with the CPU virtualization and the design of intermediate representation.

2.2.1 Guest CPU Virtualization

As for a traditional program binary to run on a machine, the program needs a view of the processor architecture. Since the guest ISA can be different to the host ISA in a retargetable DBT, the guest application binary may not be executed natively on the host platform. To make

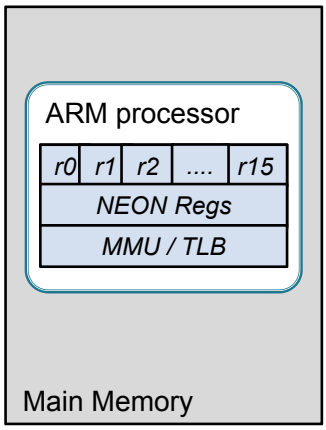


Figure 2.2: An example of the ARM virtual processor.

a guest processor architecture virtualized on a machine with a different ISA, the guest processor is modeled by allocating a *software structure* (i.e. virtual CPU) in the main memory, where each guest processor state is represented by an element in the software structure. Based on the purpose of the emulator, the modeled processor states can range from macro function units to micro-architecture. For example, a functional simulator, e.g. QEMU, manages the processor states of general purpose registers, program counter and flag register, etc., for process-level emulation; a full-system virtualizer further models the processor memory management unit (MMU) which is necessary for handling memory paging. Figure 2.2 illustrates an example of an ARM virtual processor for the full-system virtualizer. More detailed micro-architecture states, such as pipeline stages, cache or interconnection network, are required for a cycle-accurate simulator, e.g. SimpleScalar[4] and PTLsim [89].

This virtual processor plays a key role in a retargetable DBT system throughout the emulation of a guest application. The emulation of a guest application is conducted via operating the virtual processor on behalf of the execution of guest instructions. We take the register movement instruction of ARM as an example:

$$\begin{array}{ll} mov & r1, r0 \\ mov & r2, r0 \end{array}$$

The semantic of the first instruction, moving a 32-bit value from register $r0$ to $r1$, is emulated by loading a 32-bit value from the virtual processor element of register $r0$ to a host register, and then storing it to the element of register $r1$. The same load-and-store operation is conducted for the second instruction as well. In this register movement example, four memory operations are required to emulate these two instructions, compared to zero memory access if the instructions are executed natively.

With this special software structure, the design of virtual processor is retargetable because it can be implemented with any host ISA and without special hardware support. The software solution, however, imposes additional costs such as extra memory access overhead as indicated in the example.

2.2.2 Intermediate Representation

Using intermediate representation (IR) is a common approach in compiler design to achieve retargetability. IR separates the frontend, optimizer and the backend as different modules, which can be manipulated individually. There are two advantages to use IR when building a retargetable DBT. First, it can reduce engineering efforts tremendously when many targets are going to be supported. Assuming that n guests and m hosts need to be translated, the designer needs to implement $n \times m$ translators with direct translation. In contrast, it requires only $n + m$ translation pairs with IR (n from guests to IR and m from IR to hosts), which is much less than that of the direct approach. Second, it is easier to perform optimizations on IR than in either the guest instructions or the host instructions. IR enables those machine-independent optimizations, thus the optimization can be done for supported ISAs—it improves the retargetability of binary translation.

The translation engine (i.e. the JIT compiler) of QEMU is called Tiny Code Generator (TCG) [84]. TCG provides a small set of IR – about 142 different operation codes. When a block of guest binary code is fetched, the frontend re-writes it by using this set of operation codes. After that, only a trivial *dead code elimination* is performed to optimize the intermediate code. After the optimization, each operation code is converted to an equivalent instruction set of the native ISA in the backend. Without further optimizations, there are often many redundant load and store operations left in the generated host codes.

Different to traditional off-line compilers which usually use heavyweight pattern matching [2] or table-driven [36] instruction selection, the TCG backend code emitter uses a *template-based* strategy for runtime code generation. A template is a pre-defined instruction (or set of instructions) or a pre-compiled code segment (i.e. a helper function) that is equivalent to the operation code. During the instruction selection process, the code emitter picks up *one* operation code at a time and substitutes it by a corresponding template—the pre-defined instruction(s) are inlined in the code cache or a call instruction is emitted to jump to the external pre-compiled code segment. Usually, the translation of each operation code is independent to that of other operation codes based on the template-based approach. Furthermore, no inter-template optimization is performed because it is difficult to optimize among the templates of various binary formats. Thus, the code emission of TCG is lightweight because the translation cost is incurred only from populating the templates in the code cache.

Currently, TCG defines about 142 operation codes—a set much smaller than its supported ISAs. These operation codes can only conduct very *basic* operations, such as register movement, simple arithmetic/logical operations, memory load/store, condition and branch, etc. In addition, they support *integer* and *scalar* types only. As a result, a complex instruction or unsupported data type (i.e. floating point and vector types) will be translated to *many* basic operation codes, or by calling out a pre-compiled helper function which emulate its semantic (e.g. emulation of software floating point). The semantics of the original instruction may also be lost after translation.

Figure 2.3 shows an example which translates one complex ARM guest instruction to TCG IR, and then to the x86-64 host binary. In Figure 2.3(a), the ARM load-multiple instruction loads four contiguous data from memory to four registers $r4$ to $r7$. Since there is no corresponding

$ldm \quad r1!, \{r4, r5, r6, r7\}$	1:	mov_i32	$tmp7, r1$	mov	$0x4(\%r14), \%ebp$
	2:	$qemu\_ld32$	$tmp8, tmp7, \$0x1$	mov	$0x8000(\%rbp), \%ebx$
	3:	mov_i32	$r4, tmp8$	mov	$\%ebx, 0x10(\%r14)$
	4:	add_i32	$tmp7, tmp7, \$0x4$	add	$\$0x4, \%ebp$
	5:	$qemu\_ld32$	$tmp8, tmp7, \$0x1$	mov	$0x8000(\%rbp), \%ebx$
	6:	mov_i32	$r5, tmp8$	mov	$\%ebx, 0x14(\%r14)$
	7:	add_i32	$tmp7, tmp7, \$0x4$	add	$\$0x4, \%ebp$
	8:	$qemu\_ld32$	$tmp8, tmp7, \$0x1$	mov	$0x8000(\%rbp), \%ebx$
	9:	mov_i32	$r6, tmp8$	mov	$\%ebx, 0x18(\%r14)$
	10:	add_i32	$tmp7, tmp7, \$0x4$	add	$\$0x4, \%ebp$
	11:	$qemu\_ld32$	$tmp8, tmp7, \$0x1$	mov	$0x8000(\%rbp), \%ebx$
	12:	mov_i32	$r7, tmp8$	mov	$\%ebx, 0x1c(\%r14)$
	13:	add_i32	$tmp7, tmp7, \$0x4$	add	$\$0x4, \%ebp$
	14:	mov_i32	$r1, tmp7$	mov	$\%ebp, 0x4(\%r14)$
(a) ARM guest code		(b) TCG IR		(c) x64 host code	

Figure 2.3: An ARM-to-x86/64 translation example using QEMU. The ARM load-multiple instruction is first converted to multiple TCG IR load operations in the frontend. The template-based code emitter then translates the IR to host binary one operation code at a time in the backend.

load-multiple IR code for it, this instruction is split to four IR loads (instruction 2 to 4 in Figure 2.3(b) represent one load). After that, each operation code is individually translated to one x86-64 instruction which is shown in Figure 2.3(c). As the example shows, this complex guest instruction is translated to 14 operation codes and 14 x86-64 host instructions in the end.

Since there is not much optimization performed at the IR level and backend of QEMU, the translated code quality almost cannot be improved once the guest instructions are converted to the intermediate code. Even if we change the example in Figure 2.3 to using ARM host (the same as the guest) and we know this guest instruction can be translated to the same load-multiple instruction, there are still 14 ARM host instructions being translated due to the limitation of the template-based instruction selection approach. Hence, the translated host code quality still has room for improvement.

The IR design of QEMU implies two issues: (1) The richness of the IR operation codes is important to the translated code quality for simple instruction selection approaches, e.g. the template-based method. (2) An aggressive optimizer and a powerful backend code emitter are required for a weak IR to achieve the optimal translation quality. These implications motivates the needs of a rich IR as well as a powerful optimizer/backend in order to achieve high translation quality for a retargetable DBT system.

2.3 Emulation Overhead

Until this point, we have learned the virtual CPU structures and IR design strategies of a retargetable DBT system. This section identifies four sources of the emulation overheads in

QEMU – *context switch*, *code region transition*, *helper function calls* and *thread serialization*. These overheads depend on the structures and implementation of the DBT framework, and thus a non-retargetable DBT system can also incur the same overhead if it has similar structures. By learning the cause of these overheads, it will be more clear about how to solve these issues.

2.3.1 Context Switch

Context switch happens when the execution transits between the code cache and the dispatcher. During each transition, the emulation incurs a context switch cost. Such context switch overhead comes from two parts: (1) To safely re-enter the dispatcher and code cache, the dispatcher and code cache are often designed as two co-routines of the DBT. Some DBT systems, such as QEMU, implement entering the code cache as if the dispatcher *calling* the routine of code cache, and leaving the code cache as *returning* to the dispatcher. On the contrary, some other systems [6, 79] design leaving the code cache as calling the dispatcher routine. Both approaches incur the function call overhead when execution switches between these two routines. (2) When the execution enters the code cache, the guest program states are loaded from the virtual processor in the memory to the host registers. Furthermore, the modified states in the host registers are synced back to the memory when the execution leaves the code cache so that the operations within the dispatcher can see the most updated guest states. Such loads/stores of guest program contexts also cause overheads.

Chaining [7], an runtime optimization technique to keep execution remaining inside the code cache by linking together blocks of the cached code, is a well-known solution to eliminate the context switch overhead, and is also applied in QEMU.

2.3.2 Code Region Transition

As mentioned in the last subsection, a typical DBT needs to save and restore guest program contexts when the control switches between the execution in the dispatcher and the translated code in the code cache. In the execution of the translated code within the code cache, it may again need to load and store guest program contexts during *code region transition*. The problem is that DBT usually translates one code region at a time, often at the granularity of one basic block. Hence, it conducts register mapping only within this code region. To ensure the correctness of emulation, the values of the guest registers are required to be stored back to the virtual processor before control is transferred to the next code region, and be reloaded at the beginning of the next code region. Even if two code regions have a direct transition path (e.g. through block chaining, shadow stack [21] or IBTC [80]) and also have the same guest to host register mappings, values of the guest registers still need to be stored and reloaded because the translator cannot be sure if any of these code regions could be the jump target of an unknown code region.

Figure 2.4 illustrates an example of the region transition problem. Each box in the code cache represents a code region. As the figure shows, the guest register $r1$ is mapped to different host registers, *%eax* and *%ebx*, in the first and second code region, respectively. During the transition between these two regions, the value in *%eax* is stored to $vcpu.r1$ and reloaded again to *%ebx*

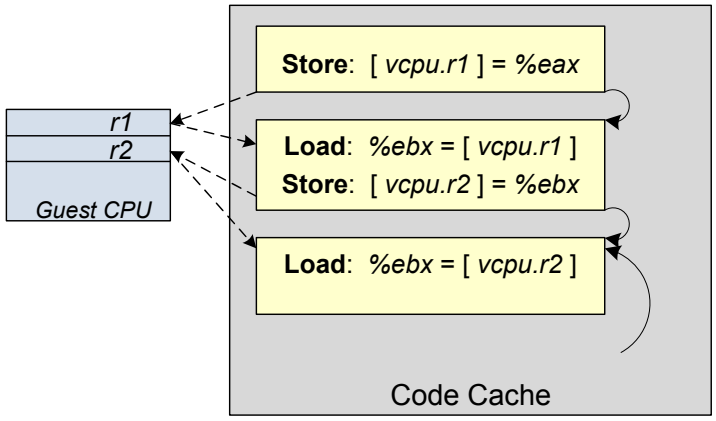


Figure 2.4: A code region transition example. During region transition, the states are synced to the virtual CPU when leaving a region and reloaded at the next region. Each box in the code cache represents a code region. A solid-line arrow represents a transition and a dash-line arrow represents a memory load/store operation from/to the guest CPU.

at the second region so as to complete the value propagation. The transition among the second and third region also requires a store and load operation even if the guest state $vcpu.r2$ is already mapped to the same host register *%ebx*. Both region transitions cause additional loads and stores.

Because of these independently translated code regions and the resulting frequent storing and reloading of registers during code region transitions, the performance could be very poor. In QEMU, it chooses a small translation granularity—a basic block. Inevitably, the guest binary will be split into a huge number of small code regions after translation, which results in very frequent block transitions. This overhead becomes an important problem that impedes the performance of QEMU.

2.3.3 Helper Function Invocation

Using helper functions in a DBT system has several advantages. First, helper functions are usually coded with high-level languages (e.g. C or C++), which is easier to implement complex semantics rather than using the small set of intermediate operation codes. For instance, QEMU uses helper functions to implement complex guest instructions, such as floating point and vector operations, and page-table walks, etc.; Pin [66] developers design profiling tools as external functions, e.g. a cache simulator [53]. Second, the helper functions are pre-compiled to binary code segment by the off-line compiler, which incurs no runtime compilation overhead. And third, compared to inlining code with IR, the memory consumption of code cache is smaller due to reuses of helper functions.

However, using helper functions also has some drawbacks. Invoking a helper function needs to pay the function call overhead. Furthermore, the modified guest states also need to be stored back to the virtual processor before invoking a helper function. This is because the translator cannot be sure if the guest states could be modified by the helper function. Finally, this mechanism

can lose some performance benefits from using function inlining, and it could also hinder the IR optimization because many optimizations cannot cross a function call. Hence, the challenge is to find the best way of either using helper functions or IR operation codes when translating a guest instruction.

2.3.4 Thread Serialization

QEMU uses a *globally* shared code cache, i.e. all execution threads share a *single* code cache, and each guest code segment has only *one* translated copy in the shared code cache. Moreover, the *single* hash table as well as the JIT compiler are also shared by all threads. Because of these DBT structures, QEMU uses a *critical section* with a coarse-grained lock to serialize all accesses to the shared resources. Such design decision makes the DBT yield very efficient memory usage compared to other DBT systems using the thread-private code caches [16, 29, 87]. However, it could cause severe serialization overhead when a large number of guest threads are emulated. As will be shown in Section 2.4.4, the preliminary results using the PARSEC benchmark suite show that such serialization overhead can degrade the emulation performance dramatically.

Many recent innovations have been proposed to address this issue with different strategies: exploiting private structure instead of shared one [29, 87], deploying fine-grained locks [29], and reducing the chance of thread contentions by keeping the execution staying in the code cache [21, 80]. They motivates this research to investigate and solve the problem of poor handling of shared resources in QEMU.

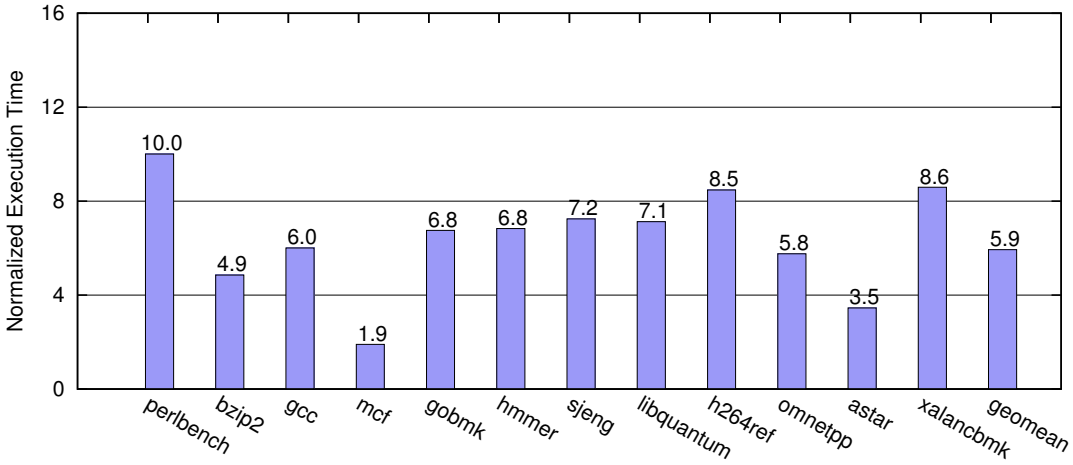
2.4 Performance Analysis

To further understand how the translated code quality, optimization overhead and emulation overhead impact the performance, we quantify these factors by conducting experiments using the QEMU dynamic binary translator. Here we evaluate overall performance, expansion rate and optimization overhead with single-thread benchmarks, and the performance scalability with multi-thread benchmarks.

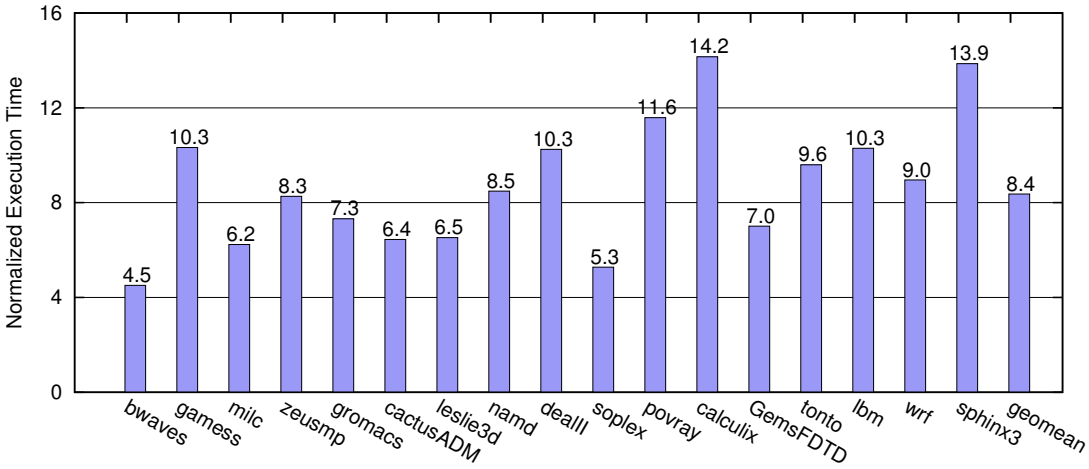
In the following experiments, we use SPEC CPU2006 benchmarks with reference inputs as the single-thread test cases. We run these benchmarks on a host machine with one 3.3 GHz quad-core Intel Core i7 processor and 12 GBytes main memory. All SPEC benchmarks are compiled with GCC 4.4, -O2 optimization, and with SIMD enabled. As for the multi-threaded tests, we use one of the PARSEC benchmarks for the experiment. The benchmark is conducted on a host machine which has eight six-core AMD Opteron processors (48 cores in total) with a clock rate of 2 GHz. The benchmark is parallelized with the *Pthread* model and compiled for x86-32 guest ISA.

2.4.1 Overall Performance

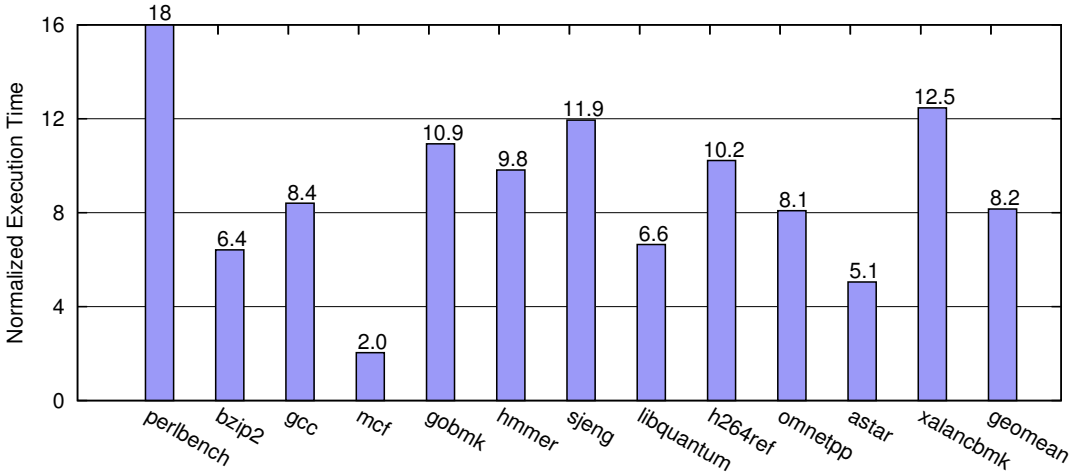
The first experiment is to evaluate the overall emulation performance. Using native run (i.e. running binary code natively) as the baseline performance, the normalized execution time over



(a) x86-32 to x86-64 (CINT)



(b) x86-32 to x86-64 (CFP)



(c) ARM to x86-64 (CINT)

Figure 2.5: QEMU results of x86-x64 and ARM-x64 emulation for SPEC CPU2006 benchmarks with reference inputs. Y-axis shows the normalized execution time over native run.

the native execution time indicates how best the binary translation can achieve for the guest applications. Note that the overall performance is resulted not only from the translated code quality but also from the optimization and emulation overhead. The experiment is conducted with one same-ISA translation and one cross-ISA translation, which emulate x86-32 and ARM guest on the x86-64 host, respectively.

Figure 2.5(a) and 2.5(b) illustrate the overall performance of x86-32 to x86-64 emulation for integer and floating point benchmarks, respectively. In Figure 2.5(a), the normalized execution time of the integer benchmarks ranges from 1.9X (`mcf`) to 10X (`perlbench`) and the geometric mean is 5.9X. The poor performance is also observed in the floating point benchmarks, where the slowdown factors range from 4.5X (`bwaves`) to 14.2X (`calculix`) and the geometric mean is 8.4X. For this same-ISA translation, the performance of most benchmarks is more than 5X slower compared with the native execution.

For floating point benchmarks, the slowdown factor over native run is greater than that of the integer benchmarks. This is partly due to the current translation ability of QEMU TCG. The QEMU TCG translator does not emit floating point instructions of the host machine. Instead, all floating point instructions are emulated with helper functions, which cause the helper function invocation overhead (see Section 2.3.3 for more details).

Figure 2.5(c) illustrates the performance result of ARM to x86-64 emulation for integer benchmarks. As the result shows, the slowdown factor of the cross-ISA translation is about 8.2X on average – greater than that of same-ISA translation. Hence, optimizing the performance of cross-ISA translation is more challenging.

2.4.2 Expansion Rate

Expansion rate, which is the number of host instructions translated per guest instruction (i.e. the total number of host instructions generated divided by that of the guest instructions), is used to show how well the JIT compiler can translate for the guest instructions. Note that we only count the host instructions generated within the code cache, and the instruction counts of extern helper functions are excluded. In this experiment, the expansion rate is measured by conducting the x86-32 to x86-64 emulation. We only evaluate same-ISA translation in this experiment because it is simpler to verify the translation quality when the guest and host ISA are the same.

The expansion rates of SPEC CPU2006 benchmarks are listed in *column 4* and *8* of Table 2.1. As the results show, one x86-32 guest instruction is translated to about 6.1 and 6.2 x86-64 host instructions on average for integer and floating point benchmarks, respectively. The results also verify that QEMU TCG does translate one guest instruction to multiple host instructions.

2.4.3 Optimization Overhead

The optimization overhead is quantified by measuring the time spent for the translation and optimization by the JIT compiler. Table 2.1 lists the number of guest blocks translated and the total translation time for the SPEC CPU2006 benchmarks. The number of guest blocks translated is listed in *column 2* and *6*, and the translation time is listed in *column 3* and *7* for integer and

Benchmark	#Block	Trans. (sec)	Expansion Rate	Benchmark	#Block	Trans. (sec)	Expansion Rate
perlbench	61447	.33	6.2	bwaves	4448	.05	6.2
bzip2	21275	.13	6.1	gamess	49199	.40	6.3
gcc	525367	2.77	6.3	milc	4540	.03	6.2
mcf	3205	.02	6.2	zeusmp	8879	.07	6.3
gobmk	108759	.70	6.2	gromacs	7473	.05	6.2
hmmer	10213	.06	6.3	cactusADM	9445	.06	6.1
sjeng	4918	.03	6.3	leslie3d	6482	.04	6.3
libquantum	3001	.02	6.1	namd	5546	.06	6.3
h264ref	27168	.18	5.8	dealII	15853	.11	5.8
omnetpp	13593	.10	6.0	soplex	16930	.11	6.0
astar	8374	.05	6.2	povray	12905	.08	6.3
xalancbmk	30493	.16	5.8	calculix	15851	.10	6.0
				GemsFDTD	12170	.10	6.2
				tonto	22695	.21	6.3
				lbm	2772	.02	6.3
				wrf	27942	.25	5.8
				sphinx3	7127	.04	6.2
Average			6.1				6.2

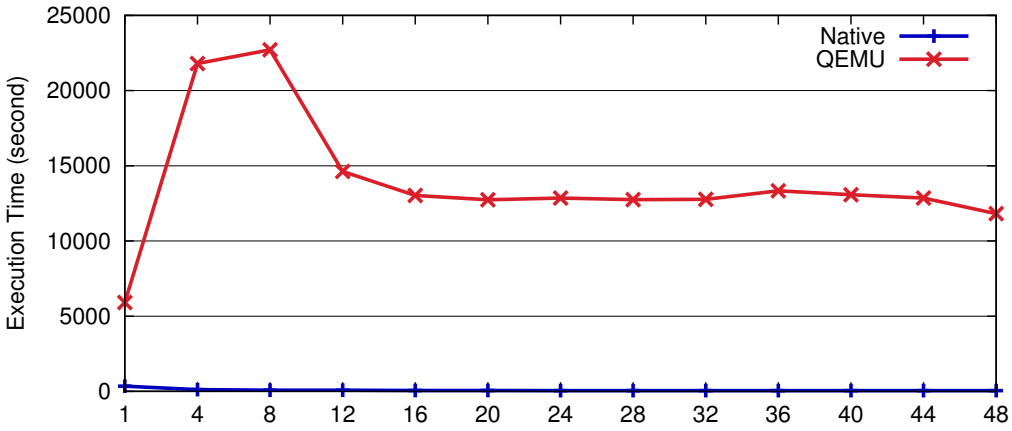
Table 2.1: Measures of x86-32 to x86-64 emulation for SPEC CPU2006 benchmarks with reference inputs using QEMU. *#Block* represents the number of guest blocks translated, and *Trans.* represents the time for code translation in seconds; *Expansion Rate* represents the average number of host instructions translated per guest instruction.

floating point benchmarks, respectively. As the result shows, a huge number of guest blocks is translated within very little time by the QEMU TCG translator – the translation/optimization overhead is almost negligible.

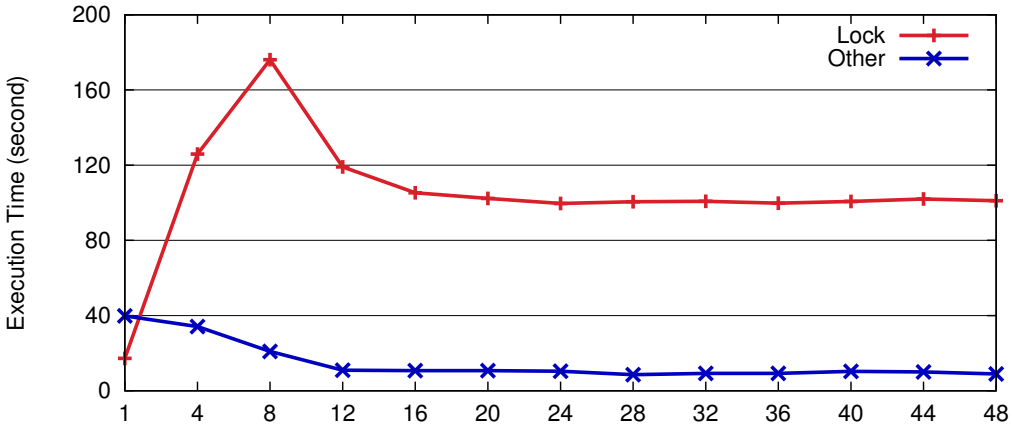
2.4.4 Performance Scalability

In this experiment, we evaluate the performance scalability of QEMU when a large-scale multi-threaded application is emulated. The benchmark `blackscholes`, one of the PARSEC benchmarks, is used as the test case. Figure 2.6(a) illustrates the performance results of `blackscholes` with native input sets. The X-axis is the number of worker threads created via the command line argument. The Y-axis is the total time measured in seconds with the `time` command.

As shown in Figure 2.6(a), the performance of QEMU does not scale well. The execution time increases dramatically when the number of guest threads increases from 1 to 8, then decreases with more threads. It remains mostly unchanged as the number of threads is above 16. The poor scalability of QEMU is mostly due to the sequential lookup of branch targets within the QEMU dispatcher because the hash table is protected in a critical section. Although the compu-



(a) Performance



(b) Breakdown of time

Figure 2.6: (a) `blackscholes` result of x86-32 to x86-64 emulation with native input. The results show that the scalability of QEMU is poor. (b) Breakdown of time with simlarge input. X-axis shows the number of threads, and the unit of time for Y-axis is in seconds.

tation time can be reduced through parallel execution with more threads, the overhead incurred by thread contention can result in significant performance degradation.

Figure 2.6(b) shows the breakdown of time for `blackscholes` with simlarge input set for QEMU. *Lock* and *Other* represent the average time of a worker thread spent in critical sections (including wait and hash table lookup time) and for the remaining code portions, respectively. As the figure shows, the time of *Other* decreases linearly with the number of threads because of increased parallelism. The time of *Lock* increases significantly because the worker threads contend for the critical section within the dispatcher where the serialization lengthens the wait time. Moreover, the time increased from such serialization outweighs the reduced execution time when more worker threads are added. Such high locking overhead dominates the total execution time, and results in poor performance of the parallel benchmark. This is why Figure 2.6(a)

shows that emulating single thread with QEMU results in the best performance compared to its multi-threaded counterparts.

2.5 Summary

The special structures, software virtual processor and intermediate representation, are used to make a DBT system retargetable but also impose additional overhead and problems. Several important issues and challenges are raised in this chapter. (1) From the observation of the experimental results, QEMU is a retargetable DBT system of low translation overhead. However, the overall performance and translated code quality are not good enough and still have room for improvement. The results reveal that the IR design and the template-based code emitter in QEMU are inadequate for generating the optimal host instructions. The low translation overhead but insufficient optimizations are unacceptable to achieve high emulation performance. (2) Emulation overheads incurred from additional memory accesses are also critical to the overall performance, especially from the use of helper functions and code region transitions. (3) Severe contentions on a coarse-grained lock can incur substantial synchronization overhead when multiple application threads are being translated simultaneously. Any inefficiency in handling globally shared data structures could be amplified many times over and could even render the multithreaded application to become slower than running it sequentially.

Such issues motivate us to improve the translation quality but retain low optimization overhead, and to eliminate the emulation overheads. The remainder of this dissertation tackles these challenges for building an efficient and retargetable DBT system.

Chapter 3

Retargetable Dynamic Binary Translation on Multicores

As a DBT is running at the same time the application is being executed, the overall performance of the translated binary on the host machine is very sensitive to the overhead of the DBT itself. A DBT could ill-afford to have sophisticated techniques and optimizations for better codes. However, with the ubiquity of the multicore processors today, most of the DBT overheads could be off-loaded to other cores. The DBT could thus take advantage of the multicore resources and leverage multithreading itself. This allows DBT to become more scalable when it needs to take on more and more large-scale multithreaded applications in the future.

In this chapter, we propose a multithreaded DBT prototype, called HQEMU (Hybrid-QEMU), which uses QEMU as its frontend for fast binary code emulation and translation. However, it lacks a sophisticated optimization backend to generate more efficient code. We thus use the LLVM compiler [60], also a popular compiler with sophisticated compiler optimization as its backend, together with a dynamic optimizer that uses on-chip hardware performance monitor (HPM) to dynamically improve code for higher performance.

In the following sections, Section 3.1 first compares the JIT compiler of LLVM and QEMU to show why low translation overhead and abundant optimization passes are both essential to achieve higher performance. We then elaborate on the design details of our multi-threaded hybrid QEMU+LLVM DBT system in Section 3.2. Section 3.3 provides the trace formation and code optimization techniques which are developed to generate more efficient code. Section 3.4 examines the problems of existing trace selection approaches, and presents our lightweight trace combination strategy. The trace formation and combination strategies discussed in Section 3.3 and 3.4 are effective to eliminate the high overhead incurred from code region transitions. Section 3.5 presents the two-level IR conversion mechanism which is able to easily integrate QEMU and LLVM for achieving retargetability, and Section 3.6 provides the mechanism to reduce the emulation overhead from helper function invocations.

3.1 Comparison of Just-in-Time Compiler of QEMU and LLVM

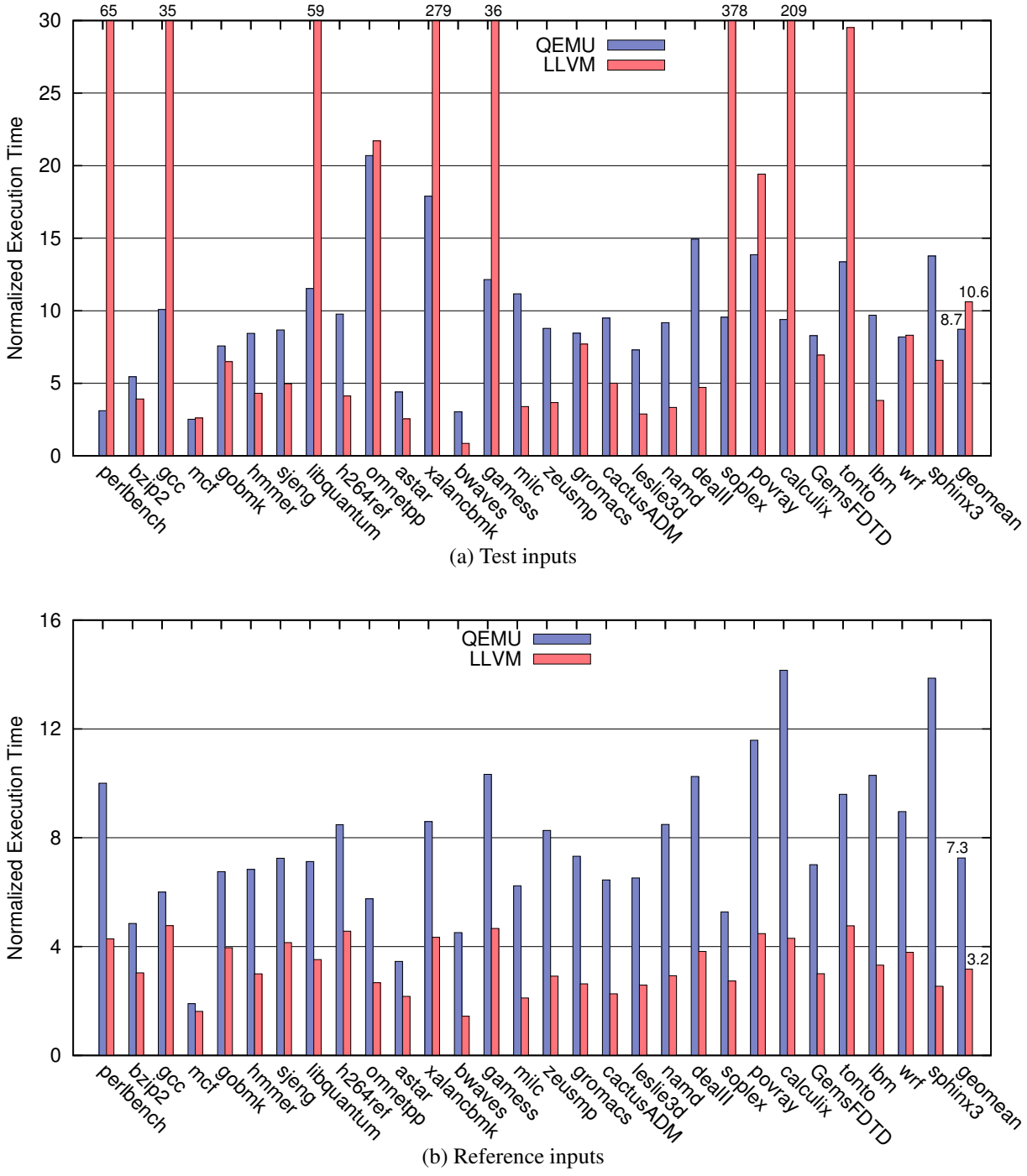
Considering the not-good-enough JIT compiler of QEMU, one design decision that arises is why not replacing it with an aggressive JIT compiler, e.g. LLVM. To verify this idea, we design a first DBT prototype that uses the same components of QEMU except that the QEMU TCG translator is substituted by the LLVM JIT compiler. Before showing the comparison results of LLVM and QEMU, we first give a brief overview of the LLVM compiler infrastructure.

LLVM defines a low-level code representation in static single assignment (SSA) form. Its IR supports common data types, including integer, floating point and vector types. Although a program is described by using low-level representation, higher-level information is also described for effective analysis. This includes type information, explicit control flow graphs, and an explicit dataflow representation (using an infinite, typed register set in SSA form [23])—it allows sophisticated algorithms for pointer analysis, dependence analysis, and data transformations. Moreover, the LLVM JIT runtime system consists of a rich set of aggressive optimization passes for generating high performance code—the default optimization level (-O2) applies greedy global register allocation, LICM, machine code sinking, instruction scheduling, peephole optimizations, etc. LLVM uses a DAG (Directed Acyclic Graph [73]) based instructions selector to translate LLVM internal representation to target specific code. A DAG-based instruction selector basically keeps the programs internal representation in a tree structure and then tries to match subtrees of this to the pre-defined host instructions—if a match is found, the subtree of the representation is converted to the matched instruction(s).

Figure 3.1 shows the performance comparison of QEMU and LLVM for CPU2006 benchmarks. The experiment is conducted with x86-32 to x86-64 emulation, and the results for test and reference inputs are illustrated in Figure 3.1(a) and 3.1(b), respectively. In Figure 3.1(a), most performance results of LLVM are better than or close to those in QEMU. However, 11 benchmarks (i.e. `perlbench`, `gcc` and `soplex`, etc.) have dramatic performance degradation with LLVM compared to QEMU. The reason that LLVM configuration has large slowdowns in these benchmarks is because too much optimization overhead is incurred without being sufficiently amortized for these short-running benchmarks. On the other hand, benchmarks such as `hmmer` and `h264ref` are the cases in which the benefit of optimized code outweighs the optimization overhead, so that the LLVM outperforms the QEMU configuration.

For the long-running benchmarks shown in Figure 3.1(b), LLVM outperforms QEMU for all benchmarks since the optimization overhead is mostly amortized. The speed up from LLVM includes some DBT related optimizations such as indirect branch prediction, as well as regular compiler optimizations such as redundant load/store elimination.

From the design decisions mentioned in Chapter 2 and the observation of the evaluation results, although the code quality of QEMU is not as good as it should be, the whole translation process including the optimizations incurs negligible overhead. Those design considerations make QEMU an ideal choice for emulating short-running applications or applications with few hot blocks. On the contrary, the high-cost optimizations of LLVM are suitable for long-running applications for which optimization overhead could be sufficiently amortized. Since QEMU and LLVM both have their own advantages, we would like to integrate these two frameworks, and



(a) Test inputs

(b) Reference inputs

Figure 3.1: Comparison of QEMU and LLVM of x86 to x86-64 emulation for CPU2006 benchmarks with test and reference inputs. The results show that QEMU is suitable for emulating short-running programs and LLVM is suitable for optimizing long-running programs. Y-axis shows the normalized execution time over native run.

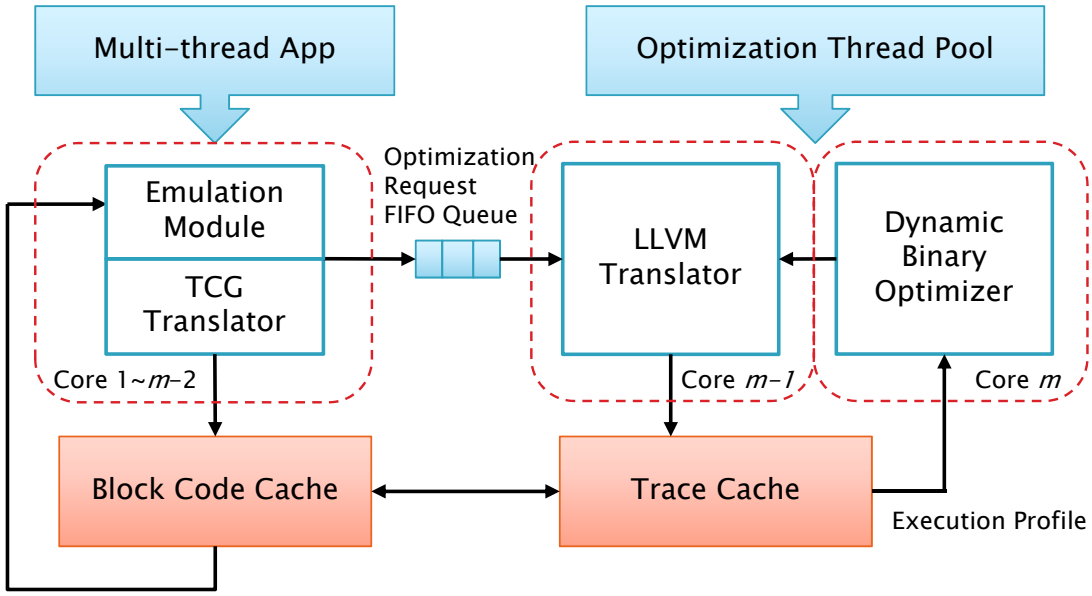


Figure 3.2: The architecture of HQEMU on an *m*-core platform.

thus, the hybrid two-translator DBT system is introduced.

3.2 Hybrid Dynamic Binary Translator

The goal of this dissertation is to design a retargetable DBT system that could emit high-quality host codes, but exert low overhead on the running applications. The requirements of low overhead and high-quality codes are often in conflict with each other on a single-core system. To find a good balance, one needs to select optimization schemes that are highly cost effective. This approach, however, limits many optimization opportunities because it is very difficult to find sufficient optimization schemes that meet such criteria. Instead, we adopt a hybrid multi-threaded approach to deal with such issues.

Figure 3.2 illustrates the organization of HQEMU. It has an enhanced QEMU as its frontend, and an LLVM together with a dynamic binary optimizer (DBO) as its backend. DBO uses a hardware performance monitor based (i.e. HPM-based), feedback-directed runtime optimization scheme. The details of DBO are discussed in Section 3.4. Two code caches: a *block-code cache* and a *trace cache*, are implemented in the DBT system. They keep translated binary codes at different optimization levels.

The two translators are designed for different purposes. The translator in the enhanced QEMU (i.e. TCG) acts as a fast translator. TCG translates guest binary at the granularity of a basic block, and emits translated codes to the *block-code cache*. It also keeps the translated guest binary in its TCG IR format for further optimization in the HQEMU backend. The emulation module (i.e. the dispatcher in QEMU) coordinates the translation and the execution of the guest program. It kicks start TCG when an untranslated block is encountered. The purpose of the

emulation module and the fast translator is to perform the translation as quickly as possible, so we could switch the execution quickly to the translated code in the *block-code cache* for a higher performance. When the emulation module detects that some code region has become hot and is worthy of further optimization, it sends a request to the *optimization request FIFO queue* with the translated guest binary in its TCG IR format. The requests will be serviced by the HQEMU backend translator/optimizer running on another thread (and on another core).

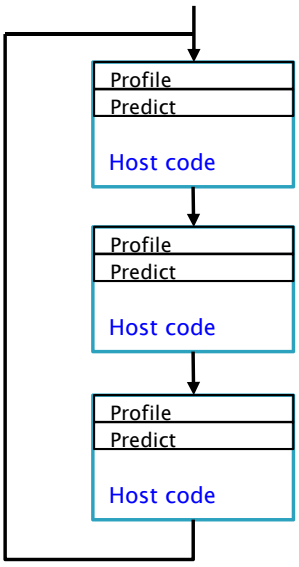
When the backend LLVM translator receives an optimization request from the FIFO queue, it converts its TCG IRs to LLVM IRs directly instead of converting guest binary from its original ISA (see Section 3.5 for more details). Many LLVM compiler optimization passes are performed on the LLVM IR, and finally, highly optimized host code is emitted to the *trace cache*. A rich set of program analysis facilities and powerful optimization passes in LLVM can produce very high quality host codes. For example, redundant memory operations can be eliminated via LLVM's register promotion optimization. LLVM also selects the best host instructions sequences. It can replace several scalar operations by one SIMD instruction, for instance. These analysis and optimization passes could incur considerable overhead. However, the LLVM translator is running concurrently on another thread (and on another core). Hence, such overhead could be hidden without interfering with the execution of the guest program.

The backend LLVM translator could also spawn more worker threads to accelerate the processing of optimization requests if there are many of them waiting in the queue. We also apply the structure of a non-blocking FIFO queue [71] to reduce the overhead of communication among these threads. With the hybrid QEMU+LLVM approach, we could benefit from the strength of both translators. This approach successfully addresses the dual issues of good translated code quality and low translation overhead.

3.3 Trace Optimization

In the execution of the translated code from the code cache, we may need to load and store registers during code region transition. The problem is that DBT usually translates one code region at a time—it performs register mapping only within one code region. To ensure the correctness of emulation, the values of the guest registers are required to be stored back to the memory before control is transferred to the next code region, and be reloaded at the beginning of the next code region. Even if two code regions have a direct transition path (e.g. through block chaining, shadow stack [21] or IBTC [80]) and also have the same guest to host register mappings, values of the guest registers still need to be stored and reloaded because we cannot be sure if any of these code regions could be the jump target of an unknown code region.

Because of these independently translated code regions and the resulting frequent storing and reloading of registers during code region transitions, the performance could suffer significantly. Such high transition overheads can be alleviated by enlarging the granularity of code regions. The idea is to merge many small code regions into larger ones, called *traces*, and thus eliminating the redundant load and store operations by promoting such memory operations to register accesses within traces. Traces are code regions with a single entry and multiple exits.



Algorithm 1: Trace profile and prediction

```
1:  profile_stub_label:
2:  jump < target >
3:  if enable_profile[i] = TRUE then
4:      counter ← counter + 1
5:      if counter ≥ threshold then
6:          enable_predict ← TRUE
7:      end if
8:  end if

9:  predict_stub_label:
10: if enable_predict = TRUE then
11:     if PC exists in recording_list then
12:         enable_predict ← FALSE
13:         CompileTrace(recording_list)
14:     else
15:         Add PC to recording_list
16:     end if
17: end if
```

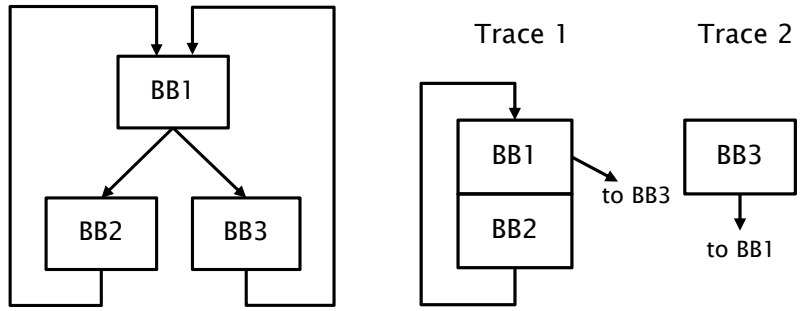
Figure 3.3: An example of trace detection and pseudo code of the profiling and prediction stubs. At the beginning of each translated block, we instrument two small pieces of codes for trace profiling and prediction.

They have been shown to improve performance because of improved locality, increased indirect branch prediction accuracy, and opportunities for inter-procedural optimizations [17, 70]. Through trace formation, we not only can apply more code optimizations but also can eliminate the high overhead of region transitions.

Since such optimizations can be done in separate threads running on different cores without interfering with the emulation of the guest application, we try to explore more optimization opportunities on those traces. A relaxed version of *Next Executing Tail* (NET) [32] is chosen as our trace selection algorithm. In the original NET scheme, it considers every backward branch as an indicator of a cyclic execution path, and terminates the trace formation at such backward branches. We relax such a backward-branch constraint, and stop trace formation only when the same program counter (PC) is executed again. This relaxed algorithm is similar to the *cyclic-path-based* repetition detection scheme in [42].

In HQEMU, a trace is detected and formed by locating a hot execution path through an instrumentation-based scheme. Figure 3.3 gives an example of the trace detection and formation scheme. Two small pieces of codes: a *profiling stub* and a *prediction stub*, are inserted at the beginning of each translated code region in the block-code cache. The *profiling stub* determines whether a block is becoming *hot* or not. The *prediction stub* will then append a hot code block to a list, called *recording_list*. The code blocks on the *recording_list* will be combined into traces later. The pseudo code of these two stubs is shown in Algorithm 1 in Figure 3.3.

To detect a hot trace, we have to locate the head code block of the candidate trace first. During the emulation, the QEMU dispatcher gets the starting PC of the next guest basic block to be executed. The dispatcher looks up a directory to locate the translated host code block pointed



(a) CFG (b) Trace

Figure 3.4: A CFG of three basic blocks and traces generated with NET trace selection algorithm. Since NET can only form traces of straight fall-through path or simple loop, two separate traces are formed in (b) for the two-loop CFG in (a).

to by this PC. If there is a miss in the directory, the emulation module translates the guest block and add an entry to the directory. If it is a hit, the basic block has been translated before and a cyclic execution path is found. This basic block is a potential trace head, and its associated profiling routine is enabled. The counter is incremented each time this block is executed. When the counter reaches a threshold, the prediction routine is activated to record the blocks following the head block executed in *recording_list*. When the prediction routine detects that a head block is already in the recording list, a cyclic path is formed and the trace prediction stops. A request is issued to the LLVM translator through the *optimization request FIFO queue*. The LLVM translator periodically checks whether there are requests on the FIFO queue.

After optimizations by LLVM, the head block of the trace is patched a direct jump (line 2 in Algorithm 1) and the execution is redirected from the unoptimized codes to the optimized codes. This jump patching is processed asynchronously by the LLVM translator, and is transparent to the executing threads. We use self-branch patching mechanism proposed in [44] to ensure the patching is performed correctly when a multi-thread application is being emulated. The store/load of registers to/from memory within a trace is optimized by promoting these memory operations to register accesses. Since a trace is formed because of its hotness, significant block transition overhead is avoided.

3.4 Trace Merging

Although the overhead of region transitions at the basic-block level can be reduced with the trace formation and optimization, the redundant load/store operations during transitions among traces could still incur noticeable overheads. The reason is that the only type of traces NET algorithm could handle are with either straight fall-through paths or simple loops. It cannot deal with a hot code region that has a more complex control flow graph (CFG). Figure 3.4(a) shows a simple code example with three basic blocks. Applying NET on this code region will result in two separate traces as shown in Figure 3.4(b). The known flaws of such trace selection algorithms

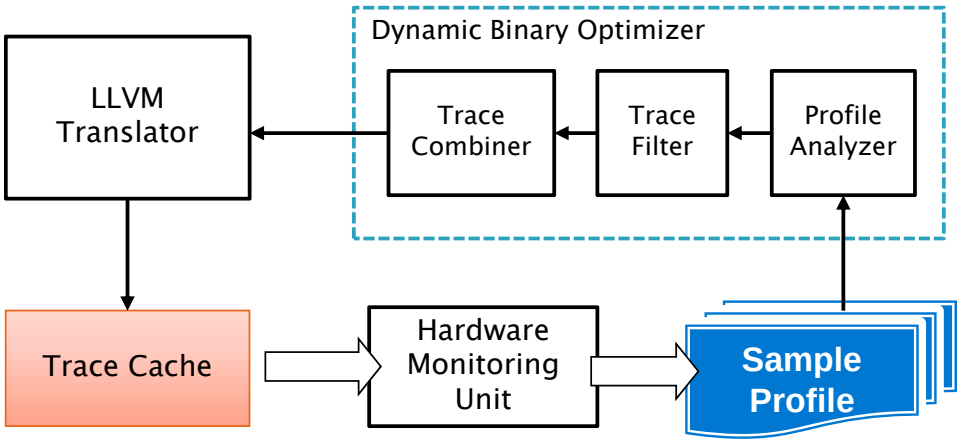


Figure 3.5: Workflow of the HPM-based trace merging in DBO.

include *trace separation* and *early exits* [45]. In order to overcome such problems, we force the merging of problematic traces that frequently jump among themselves. For example, if there are frequent jumps between the two traces shown in Figure 3.4(b), we will force the merging of those two traces into one with its CFG as shown in Figure 3.4(a).

The biggest challenges of such trace merging are (1) how to efficiently detect such problematic traces, and (2) when to merge them at runtime. One feasible approach is to use the same instrumentation-based NET algorithm described in Section 3.3, but insert routines to detect the separation of traces and early exits. This approach, however, will incur substantial overhead because they are likely to be frequently-executed hot code regions. Instead, we use a feedback-directed approach with the help of on-chip hardware performance monitor (HPM) to support trace merging. The workflow of such trace merging in DBO is shown in Figure 3.5.

The DBO consists of three components: a profile analyzer, a trace filter and a trace combiner. At first, as DBO continues to receive sampled profile, the profile analyzer collects sampled PCs and accumulates the sample counts for each trace to determine the degree of hotness of each trace. In the second step, the trace filter selects the hot candidate traces for merging. In our algorithm, a trace has to meet three criteria to be considered as a *hot* trace: (1) the trace is in a stable state; (2) the trace is in the 90% cover set (to be explained later), and (3) the sampled PC count of the trace must be greater than a threshold.

To determine if a trace has entered a stable state, a circular queue is maintained in the trace filter to keep track of the traces executed in the most recent N sampled intervals. The collection of traces executed in the most recently sampled interval is put in an entry of the circular queue, and the oldest entry at the tail of the queue is discarded if the queue overflows. We consider a trace is in a *stable state* if it appears in all entries of the circular queue. The top traces that contribute to 90% of total sample counts are collected as the *90% cover set*.

The trace combiner then chooses the traces that are likely to cause trace separation for merging. Note that, in trace optimization, we apply the concept in NET algorithm to collect the basic blocks that form a cyclic path for optimization. The same concept is applied here in trace merg-

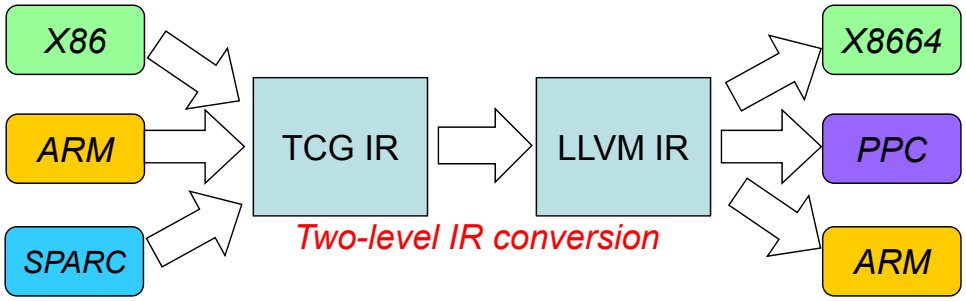


Figure 3.6: Execution flow of the two-level IR conversion.

ing. All traces that form cyclic paths after merging are collected by the trace combiner. However, we do not limit the shape of the merged trace to a simple loop here. Any CFG that has nested loops, irreducible loops, or several loops in a trace, can be formed as a merged trace. Moreover, it is likely to have several groups of traces for trace merging at a time.

Finally, the groups of traces merged by the trace combiner are passed to the LLVM translator through the *optimization request FIFO queue* for further optimizations. After a merged trace is optimized by the LLVM translator, its initial sample count is set to the maximum sample count of its component traces. Moreover, the sample counts of the component traces are reset to zero so that they will not affect the formation of the next 90% cover set for future trace combination.

3.5 Retargetability

The goal of HQEMU is to have a *single* DBT framework to take on application binaries from *several different ISAs* and retarget them to host machines with *different ISAs*. Using a common IR is an effective approach to achieve retargetability, which is used in both QEMU (i.e. TCG) and LLVM. By combining these two frameworks, HQEMU inherits their retargetability with minimum effort. In HQEMU, when LLVM optimizer receives an optimization request from the FIFO queue, it converts its TCG IR to LLVM IR directly instead of converting guest binary from its original ISA which is used in [50, 54]. Such *two-level IR conversion* simplifies the translator tremendously because TCG IR only consists of about 142 different operation codes – much smaller than in most existing ISAs. Without such two-level IR conversion, for example, supporting full x86 ISA requires implementing more than 2000 x86 opcode to LLVM IR conversion routines. Figure 3.6 illustrates the execution flow of the two-level IR conversion. Since the translation from the guest ISAs to TCG IR is already supported by TCG, all guest ISAs supported by QEMU are automatically supported by HQEMU when the 142 conversion routines from TCG IR to LLVM IR are completed.

A retargetable DBT does not maintain a fixed register mapping between the guest architectural states and the host architectural states. It thus has extra overhead compared to same-ISA DBTs (e.g. Dynamo [7]) or cross-ISA DBTs (e.g. IA-32 EL [9]), which usually assume the host ISA has the *same* or *richer* register set than the guest ISA. Moreover, retargetable DBTs allow

Hardware settings – CPU / Memory size	
x86/64	3.3 GHz quad-core Intel Core i7 975 / 12 GB
PPC/64	2.0 GHz dual-core PPC970FX / 4 GB
ARM	1.3 GHz quad-core ARMv7r / 2 GB

Optimization flags	
Native-x86/64	$DEFAULT
Native-PPC/64	$DEFAULT -maltivec
Native-ARM	$DEFAULT -mfpu=vfp
Guest-x86/32	$DEFAULT -m32 -msse2 -mfpmath=sse
Guest-ARM	$DEFAULT -ffast-math -msoft-float -mfpu=neon -ftree-vectorize

Table 3.1: Experiment configurations. DEFAULT="-O2 -fno-strict-aliasing"

flexible translation, such as adaptive SIMDization to any vector size or running 64-bit binary on 32-bit machines. This is hard to achieve by the dedicated DBTs.

3.6 Helper Function Inlining

Helper functions, which are used to emulate complex instructions such as floating point and vector operations, have been shown in Chapter 2 that aggressively using them could cause severe emulation overhead. Inlining the helper functions in the code cache could be a solution to reduce the overhead, but it seems impossible to do that because they are pre-compiled to binary code segment. However, inlining helper functions can be made in our LLVM translator with the help from the LLVM extensions, *llvm-gcc* [63] or *DragonEgg* [30]. Such facilities allow to pre-compile the helper functions into the code in the form of *LLVM IR*. Upon translating a helper function, its LLVM IR is inlined instead of using a function call. In our current implementation, the IR of a helper function is inlined only when the inlining can get benefits, for example, when most blocks of the helper function become dead or the size of inlined IR is small enough.

3.7 Performance Results

In this section, we present the performance evaluation of HQEMU by using single-threaded benchmarks. SPEC CPU2006 benchmark suite is chosen as the test programs in this experiment. Detailed analysis of overall performance and overhead of the proposed approaches are provided to verify the effectiveness of our framework.

Experimental Setup

All performance evaluation is conducted on three host platforms listed in Table 3.1. The SPEC CPU2006 benchmark suite is tested with both test and reference inputs and for two different ISAs, ARM and x86, to show the retargetability of HQEMU. All benchmarks are compiled

with GCC 4.4.2 for the x86 guest ISA and GCC 4.4.1 [22] for the ARM guest ISA. LLVM version 3.0 is used for the x86 and PPC host and version 2.8 for the ARM host, and the default optimization level (-O2) is used for JIT compilation. Because SPEC CPU2006 benchmarks are single-threaded programs, we run only one thread with the LLVM translator and this thread is capable of handling all optimization requests. The trace profiling threshold is set at 50, and the maximum length of a trace is 16 basic blocks. We use Perfmon2 [77] for performance monitoring with HPM. The sampling interval used in the experiments is set at 1 million cycles/sample. The size of the circular queue, N, for trace merging in the dynamic optimizer is set at 8. We compare the results to the native runs whose programs are compiled to the host executable with SIMD enabled. All compiler optimization flags used for each architecture are listed in Table 3.1. Four different configurations are used to evaluate the effectiveness of HQEMU:

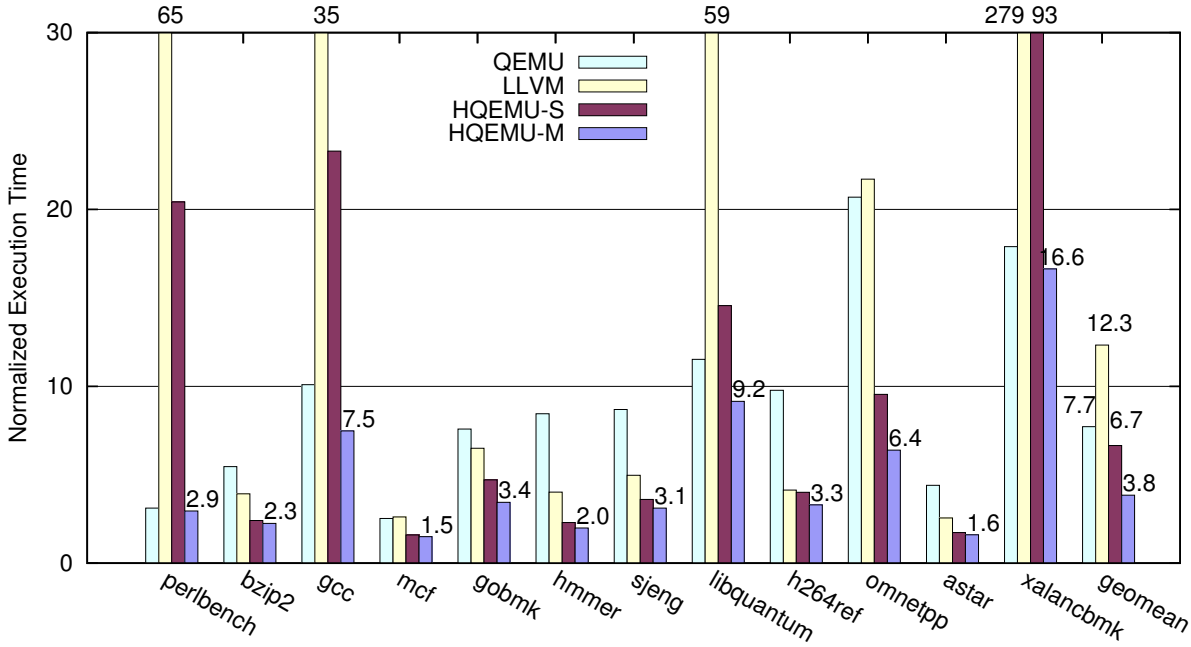
- **QEMU** which is the vanilla QEMU version 0.13 with the fast TCG translator.
- **LLVM** which uses the same modules of QEMU except that the TCG translator is replaced by the LLVM translator.
- **HQEMU-S** which is the single-threaded HQEMU with TCG and LLVM translators running on the same thread.
- **HQEMU-M** which is the multi-threaded HQEMU, with TCG and LLVM translators running on separate threads.

In both QEMU and LLVM configurations, code translation is conducted at the granularity of basic blocks without trace formation. In HQEMU-S and HQEMU-M configurations, trace formation and trace merging are used. IBTC is used in all configurations except QEMU.

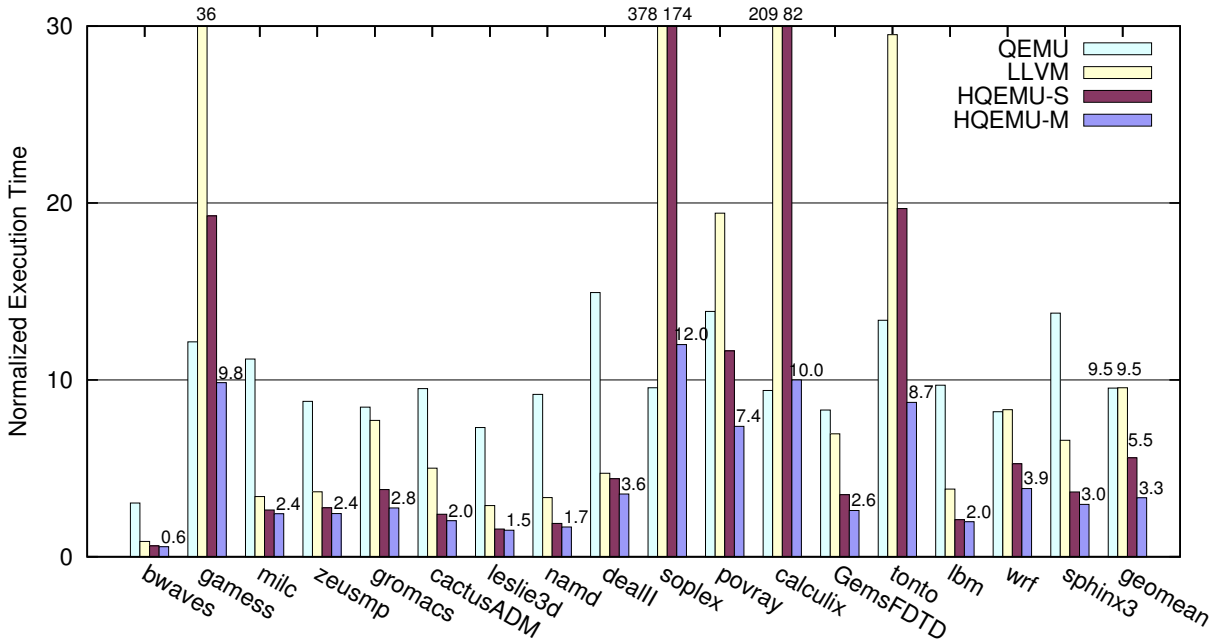
Overall Performance of SPEC CPU2006 for Same-ISA Translation

Figure 3.7 and 3.8 illustrate the overall performance of x86-32 to x86-64 emulations against the native runs. The Y-axis is the normalized execution time over native execution time. Note that in all figures, we do not provide the confidence intervals because there was no noticeable performance variation among different runs. Figure 3.7(a) and 3.7(b) show the results of CINT and CFP benchmarks with test input sets, respectively. In Figure 3.7(a), the slowdown factors of QEMU over native execution range from 2.5X to 21X and the geometric mean is 7.7X. Most performance results of LLVM are better than or close to those in QEMU except for four benchmarks: `perlbench`, `gcc`, `libquantum` and `xalancbmk`. The reason that LLVM configuration has large slowdowns in these four benchmarks is because too much translation overhead is incurred without being sufficiently amortized. Similar worse performance of HQEMU-S over QEMU is also observed for these four benchmarks due to the same reason. However, the performance of HQEMU-S is much better than that of LLVM because HQEMU-S only translates hot traces but LLVM requires to translate all guest blocks.

As for HQEMU-M, all benchmarks run faster than in both QEMU and LLVM configurations, including the four benchmarks that did not perform well in LLVM configuration compared to QEMU. The performance difference is significant. In Figure 3.7(a), the average slowdown of CINT is 7.7X for QEMU, and 12.3X for LLVM, while the slowdown to native run is

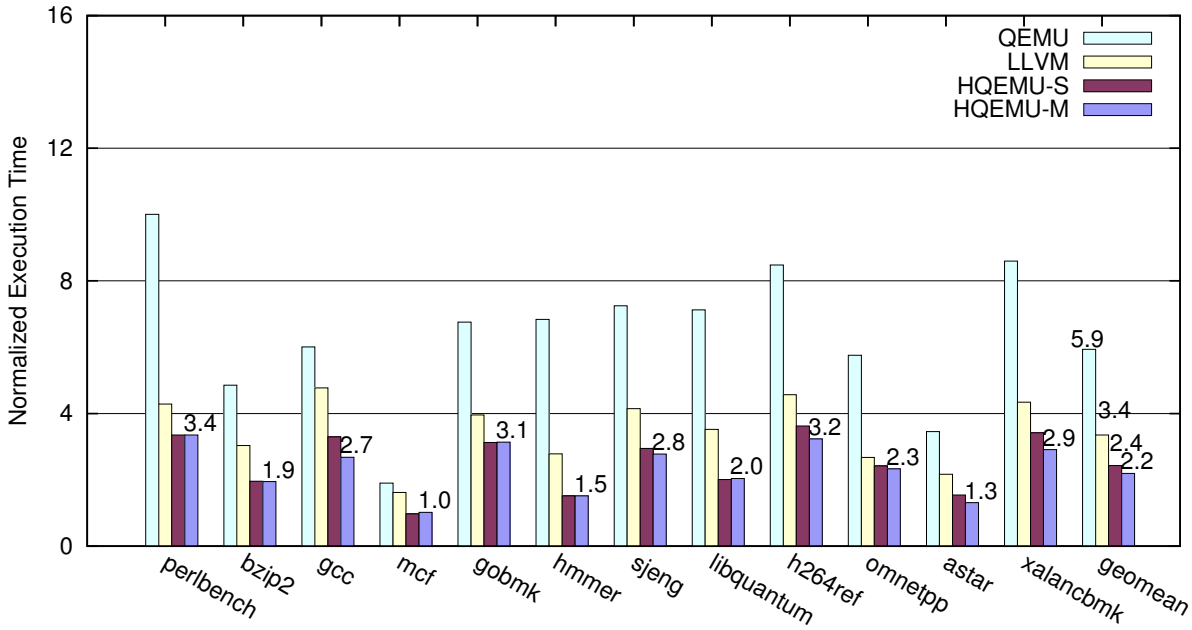


(a) CINT (Test input)

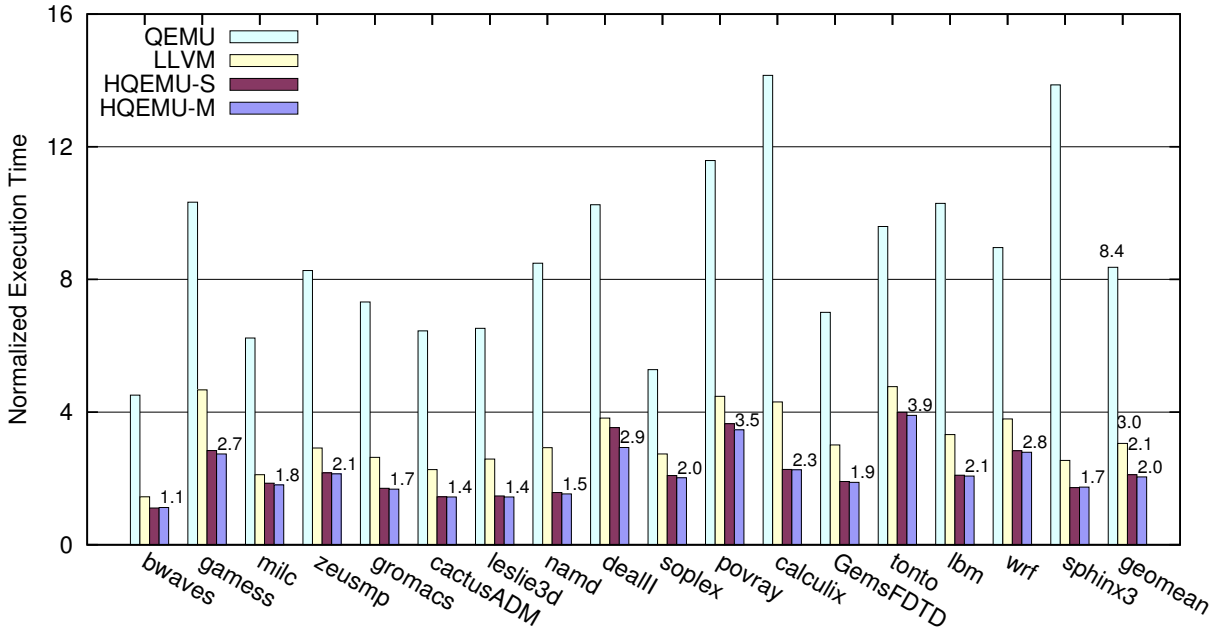


(b) CFP (Test input)

Figure 3.7: SPEC CPU2006 benchmark results of x86 to x86-64 emulation with test inputs. Y-axis shows the normalized execution time over native run.



(a) CINT (Ref input)



(b) CFP (Ref input)

Figure 3.8: SPEC CPU2006 benchmark results of x86 to x86-64 emulation with reference inputs. Y-axis shows the normalized execution time over native run.

only 3.8X for HQEMU-M. In Figure 3.7(b), the average slowdown of CFP are both 9.5x for QEMU and LLVM, while the slowdown is only 3.3X for HQEMU-M. Although the performance of HQEMU-S is not as impressive as in HQEMU-M, it still outperforms both QEMU and LLVM. Using small test inputs, fast translation becomes more important, and QEMU outperforms LLVM, based on the averaged slowdown numbers. However, many of the benchmarks can still benefit from better optimized code even with short runs (i.e. with test inputs). This is where HQEMU shines. It benefits from quick start-up as in QEMU, but when the code reuse is high, it switches its execution to the optimized trace code. The longer the code runs, the greater the benefit from optimized traces.

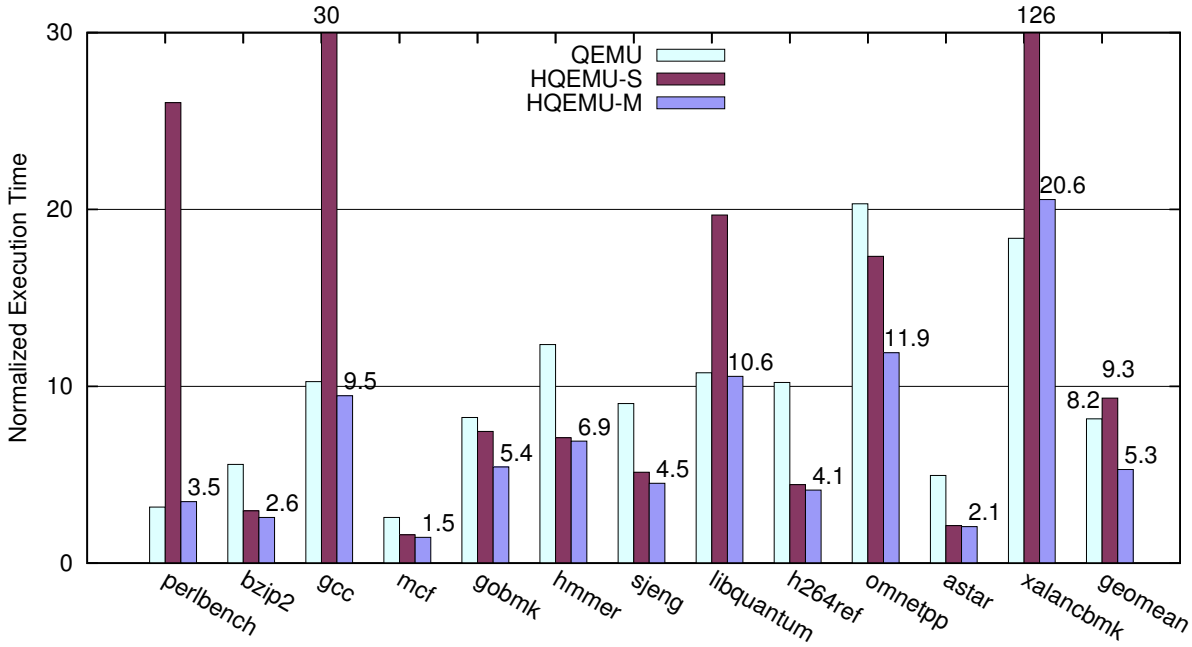
Figure 3.8(a) and 3.8(b) present the results of SPEC CPU2006 benchmarks with reference inputs. Unlike results from the test inputs, the programs spend much more time running in the optimized code caches. As the result shows, the LLVM configuration outperforms QEMU since the optimization overhead is mostly amortized. The speed up from LLVM includes some DBT related optimizations such as block chaining and indirect branch prediction, as well as regular compiler optimizations such as redundant load/store elimination. Redundant load/store elimination is effective in reducing expensive memory operations. Trace formation and trace merging in HQEMU further eliminate many redundant load/store instructions related to architecture state transitions. Through trace formation, HQEMU achieves significant improvement over both QEMU and LLVM. Using reference inputs, the benefit of HQEMU-M is not as significant as in Figure 3.7(a) and 3.7(b) when compared to HQEMU-S. This is because the translation overhead is playing less of a role using reference inputs. As shown in Figure 3.8(a) and 3.8(b), HQEMU-M is about 2.2X and 2X slower than native runs for CINT and CFP benchmarks, respectively. Compared to QEMU, HQEMU-M is 2.6X and 4.1X faster for CINT and CFP, respectively.

For CFPs, the speedup of LLVM and HQEMU over QEMU is greater than that of CINT. This is partly due to the current translation ability of QEMU/TCG. The QEMU/TCG translator does not emit floating point instructions of the host machine. Instead, all floating point instructions are emulated via helper function calls. By using the LLVM compiler infrastructure, such helper functions can be inlined and allow floating point host instructions to be generated directly in the code cache.

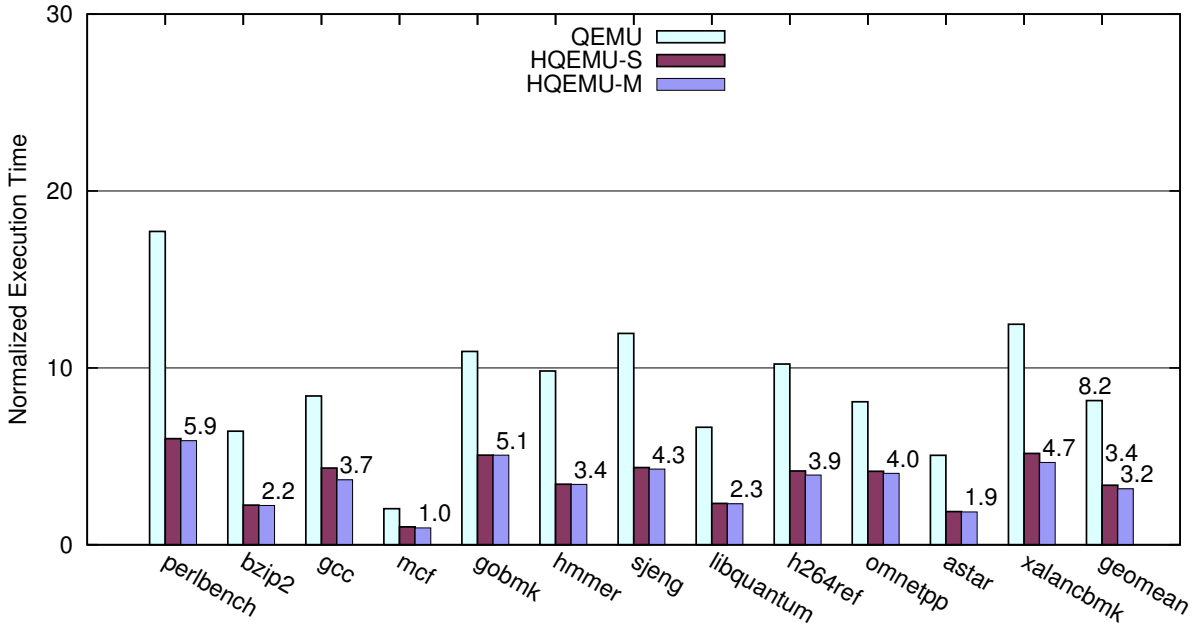
Overall Performance of SPEC CPU2006 for Cross-ISA Translation

Figure 3.9 to 3.11 illustrates the performance results of ARM to x86-64, ARM to PPC64 and x86-32 to ARM emulation over native execution (i.e. running binary code natively), respectively. For PPC64 and ARM host, the trace merging is not used.[1] The performance results in Figure 3.9 and 3.10 are similar to the results in Figure 3.8 – HQEMU-M is 2.5X and 2.9X faster than QEMU for CINT with reference inputs on x86-64 and PPC64 hosts, respectively, and is only 3.2X and 3.1X slower than the native runs, respectively. As for x86-32 to ARM emulation (in Figure 3.11), HQEMU-M achieves 2.8X speedup over QEMU for CINT with reference inputs, and is only 2.7X slower than the native runs. The results show the retargetability of HQEMU and that the optimizations used in HQEMU can also achieve performance portability.

[1] We failed to enable hardware performance monitor on PPC64 and ARM platform.

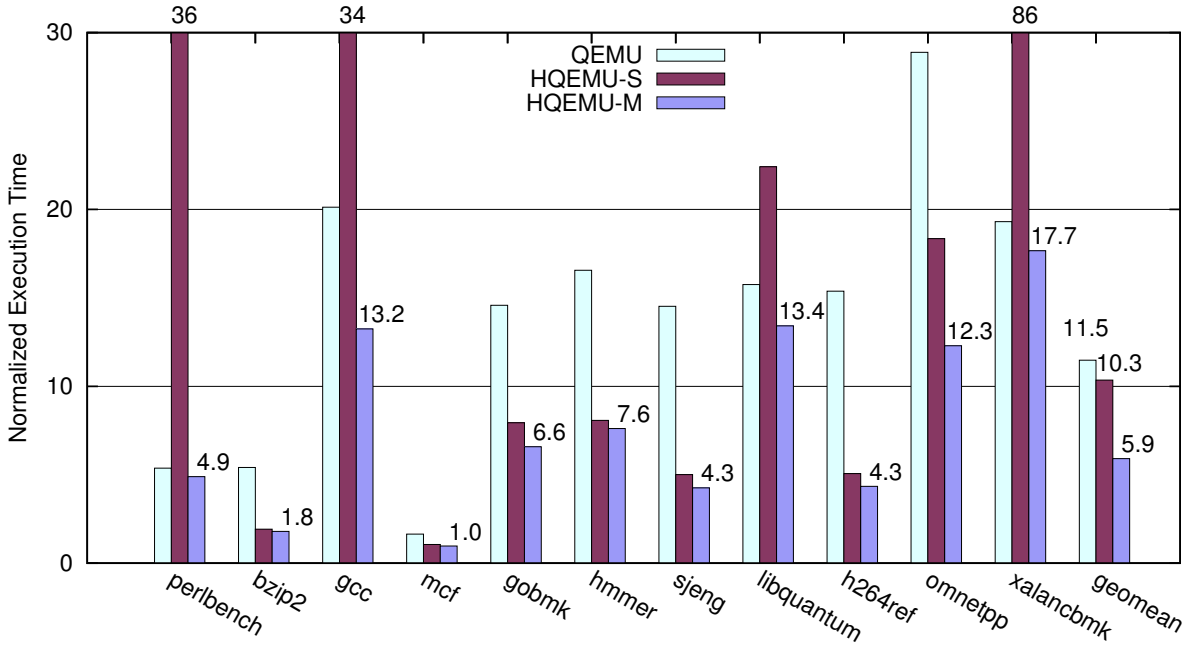


(a) ARM-x86/64 (Test input)

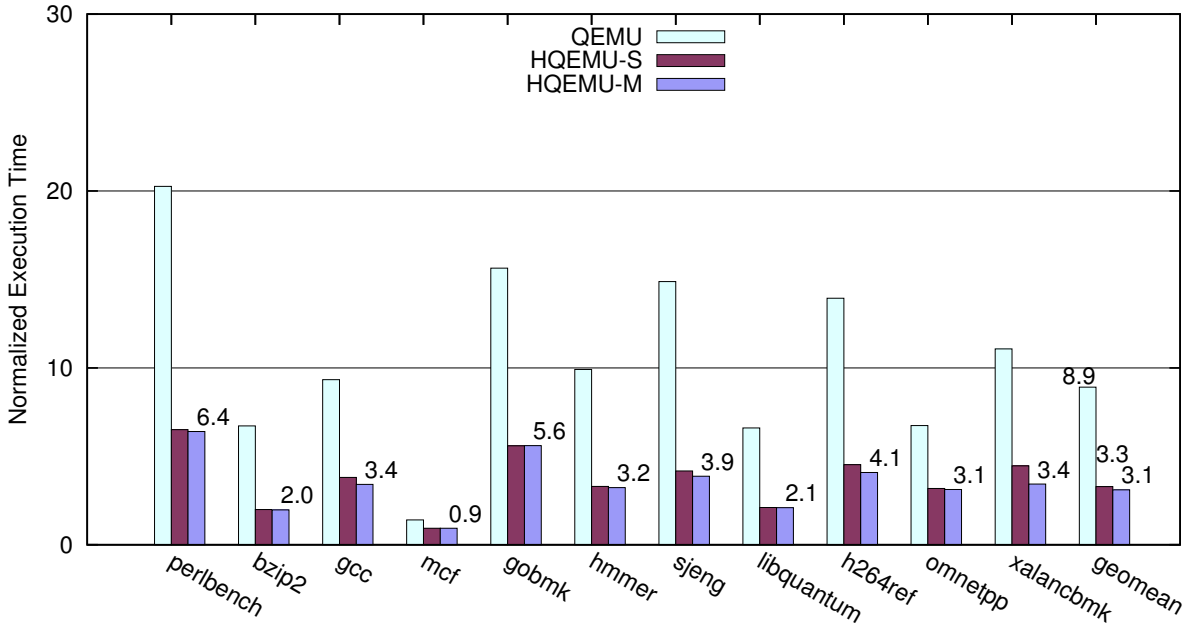


(b) ARM-x86/64 (Ref input)

Figure 3.9: CINT2006 results of ARM-x86/64 emulation with test and reference inputs. Y-axis shows the normalized execution time over native run.

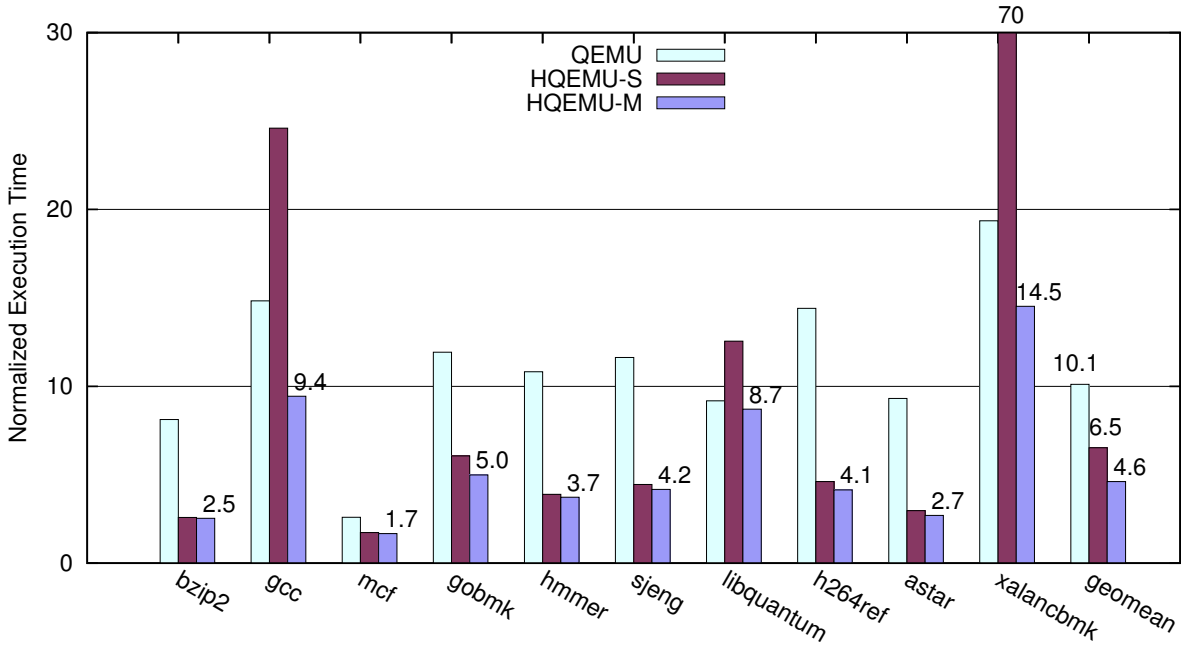


(a) ARM-PPC64 (Test input)

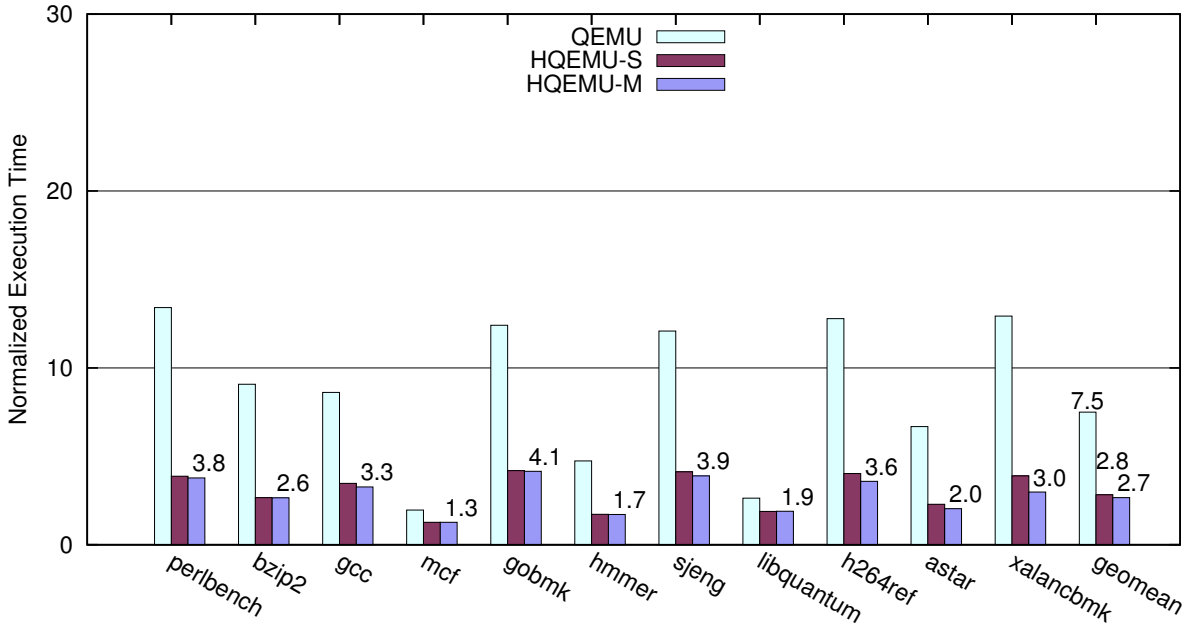


(b) ARM-PPC64 (Ref input)

Figure 3.10: CINT2006 results of ARM-PPC64 emulation with test and reference inputs. Y-axis shows the normalized execution time over native run.



(a) x86-ARM (Test input)



(b) x86-ARM (Ref input)

Figure 3.11: CINT2006 results of x86/32-ARM emulation with test and reference inputs. Y-axis shows the normalized execution time over native run.

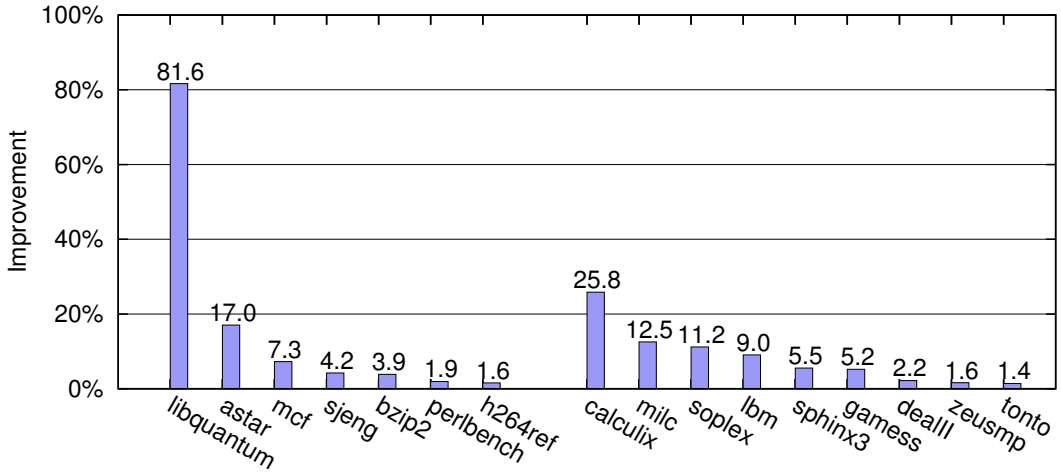


Figure 3.12: Improvement of trace merging with x86 to x86-64 emulation for SPEC CPU2006 with reference input. Y-axis shows the improvement rate over the emulation without trace merging.

The above results show that QEMU is suitable for emulating short runs or programs with very few hot blocks. The LLVM configuration is better for long running programs with heavy reuse of translated codes. HQEMU has successfully combined the advantages of both QEMU and LLVM, and can efficiently emulate both short- and long-running applications. Furthermore, the trace selection and merging in HQEMU expand the power of LLVM optimization to significantly remove redundant load/stores. With HQEMU, cross-ISA emulation is getting closer to the performance of native runs.

Results of Trace Formation and Merging

To evaluate the impact of trace formation and trace merging, we use x86-32 to x86-64 emulation with SPEC CPU2006 benchmarks as an example to show how such optimizations eliminate the emulation overhead incurred from code region transitions. In this experiment, the total number of memory operations in each benchmark is measured for (a) LLVM, (b) HQEMU with trace formation only, and (c) HQEMU with both trace formation and merging. The difference between (a) and (b) represent the number of redundant memory accesses eliminated by trace formation; the difference between (b) and (c) represents the impact of trace merging.

Hardware monitoring counters are used to track the events, `MEM_INST_RETIRED:LOADS` and `MEM_INST_RETIRED:STORES`, and to collect the total number of memory operations. Table 3.2 lists the results of such measurements. Column two presents the total number of traces generated in each benchmark; column three lists the total translation time of traces by the LLVM compiler and its percentage over total execution time. Each trace is associated with a version number and is initially set to zero. After trace merging, the version number of the new trace is set to the maximum version number of the traces merged plus one. The number of traces merged and the maximum version number are listed in column four and five, respectively. The reduced numbers

CINT2006							
Benchmark	# Trace	Trans. Time	# Merge	Version	(b)-(a)	(c)-(b)	Expan.
perlbench	13102	20.9 (1.7%)	6	4	126.7	7.1	3.8
bzip2	3084	5.2 (0.5%)	43	4	224.0	27.3	3.2
gcc	159769	215 (25.4%)	40	5	210.6	3.1	3.9
mcf	276	.6 (0.6%)	13	3	31.3	9.0	3.7
gobmk	43228	54.5 (3.9%)	57	4	136.8	3.4	3.9
hmmer	938	1.9 (0.2%)	0	0	136.6	.0	4.2
sjeng	1438	1.8 (0.1%)	33	4	159.6	36.4	3.9
libquantum	221	.4 (0.1%)	2	1	24.8	319.4	3.7
h264ref	6308	12.6 (0.6%)	1	1	396.4	16.4	3.1
omnetpp	1773	3.4 (0.4%)	7	3	39.9	6.4	4.0
astar	1074	1.8 (0.3%)	37	8	87.2	34.8	3.5
xalancbmk	3220	7.4 (1.0%)	0	0	136.2	.0	3.8

CFP2006							
Benchmark	# Trace	Trans. Time	# Merge	Version	(b)-(a)	(c)-(b)	Expan.
bwaves	364	1.3 (0.1%)	1	1	81.5	.7	3.4
gamess	10624	27.7 (1.2%)	61	5	402.9	27.1	3.4
milc	601	1.4 (0.2%)	10	2	18.5	31.2	3.4
zeusmp	1659	6.5 (0.6%)	25	6	163.5	9.5	3.7
gromacs	768	2.6 (0.3%)	1	1	136.8	-1.2	3.5
cactusADM	977	2.2 (0.1%)	1	1	226.0	-9.3	3.6
leslie3d	707	1.8 (0.2%)	0	0	348.4	.3	2.8
namd	1085	3.3 (0.5%)	6	1	224.1	3.7	3.1
dealII	3911	6.3 (0.6%)	11	3	79.6	11.8	3.5
soplex	2461	6.3 (1.2%)	11	1	47.0	48.9	4.0
povray	1958	4.5 (0.6%)	1	1	76.9	-6.4	4.1
calculix	3484	6.8 (0.3%)	15	3	393.5	372.9	3.7
GemsFDTD	1635	3.6 (0.3%)	0	0	253.8	.3	3.1
tonto	4997	12.0 (0.6%)	12	2	177.3	17.9	3.2
lbm	164	0.4 (0.1%)	1	1	78.3	13.6	3.4
wrf	5441	13.2 (0.7%)	20	2	349.0	7.4	3.2
sphinx3	1328	2.4 (0.2%)	0	0	324.5	.0	3.9

Table 3.2: Measures of traces with x86 to x86-64 emulation for SPEC CPU2006 benchmarks with reference input. *#Trace* is the number of traces translated; *Trans. Time* is the time for trace translation by LLVM; *#Merge* represents the number for traces merged; *Version* is the maximum version of the trace; *Expan.* is the expansion rate of LLVM translated code. Total number of memory operations in each benchmark is measured as (a) LLVM, (b) HQEMU with trace formation only, and (c) HQEMU with both trace formation and merging. (Unit of time: second. Unit of column six and seven: 10^{10} memory-ops.)

of memory operations after trace formation (b)-(a) and trace merging (c)-(b) are listed in column six and seven, respectively. Column eight lists the expansion rate of the LLVM translated code. The improvement rate by trace merging is shown in Figure 3.12.

From the results of column six and seven in the table, we can see that most redundant memory operations can be eliminated by trace formation in almost all benchmarks. `libquantum`, `astar` and `calculix` are the benchmarks that have the most significant improvement from trace merging (shown in Figure 3.12). As for `libquantum`, its hottest code region is composed of three basic blocks, and its control flow graph (CFG) is shown in Figure 3.4(a). The code region is then split into two separate traces by the NET trace selection algorithm. During trace transitions, almost all general purpose registers of the guest architecture need to be stored and reloaded again. In addition, there are billions of transitions between these two traces during the entire execution. Through trace merging, HQEMU successfully merges these two traces into one big code region with its CFG shown in Figure 3.4(a). It keeps the execution staying within this region without the aforementioned transition overhead. Thus, the performance of `libquantum` is improved by 82%. The other two benchmarks also exhibit similar trace separation behaviors.

From the expansion rate results of column eight in the table, the LLVM translator on average translates one x86-32 instruction to 3.6 x86-64 instructions, where the average expansion rate of QEMU TCG is about 6.1 (see Table 2.1). Hence, HQEMU does improve the translation quality.

Overhead of Trace Formation

As shown in column three of Table 3.2, most benchmarks spend less than 1% of the total time conducting trace translation. `gcc` is an exception. It has a lot of basic blocks, but no clear hot regions. About 160 thousand traces were generated at runtime that took about 215 seconds for emulating the x86-32 guest architecture (280 seconds for emulating ARM). The translation time is about 25% of the total execution time. Thanks to the multi-threaded approach in HQEMU, this significant translation overhead can be hidden by running the translation thread on a different core to minimize the impact to the emulation thread. This also matches the performance results of gcc in Figure 3.8(a) and 3.9(b) that show HQEMU-M improves over HQEMU-S by a factor of 22% and 18% for x86 and ARM guest, respectively. Compared to the block translation overhead, the QEMU/TCG spends only 3 seconds translating all basic blocks (0.3% of the total execution time).

To evaluate the impact of using different numbers of LLVM optimization threads, we conduct the x86-32 to x86-64 emulation with reference inputs and vary the number of LLVM threads from 1 to 3. Since most benchmarks spend less time performing trace translation, there is no noticeable performance difference with the change in the number of LLVM threads for all benchmarks except for `gcc`. The performance result of `gcc` is presented in Figure 3.13. As the result shows, the normalized execution time is reduced to 2.46X when adding one more LLVM optimization thread and further reduced to 2.41X with three threads (about 11% improvement compared with using only one LLVM thread).

Figure 3.14 illustrates the breakdown of instructions the emulation thread spends within block code cache, trace cache and dispatcher in the HQEMU-M configuration for SPEC CPU2006

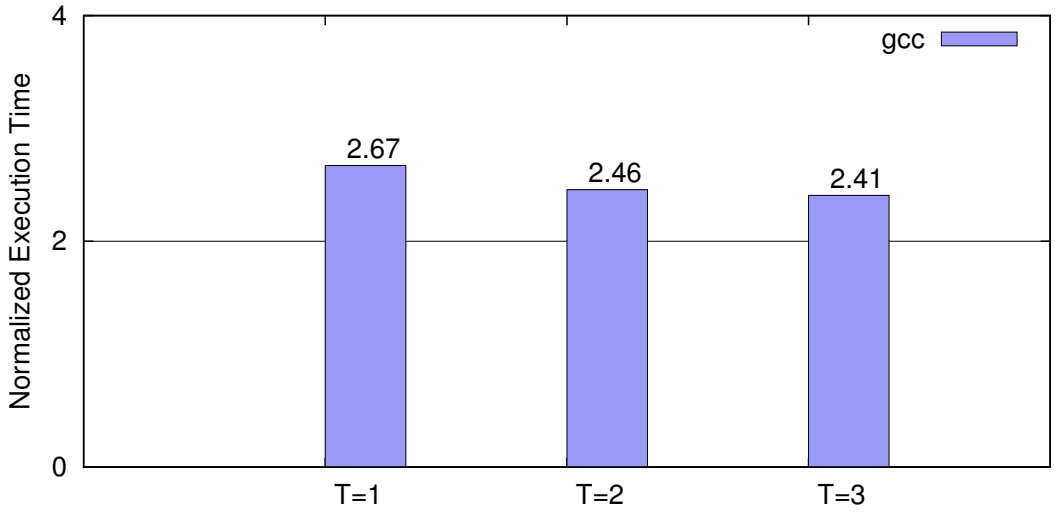


Figure 3.13: x86 to x86-64 emulation performance of `gcc` with different number of LLVM translation threads with reference inputs. X-axis shows the number of LLVM threads, and Y-axis shows the normalized execution time over native run.

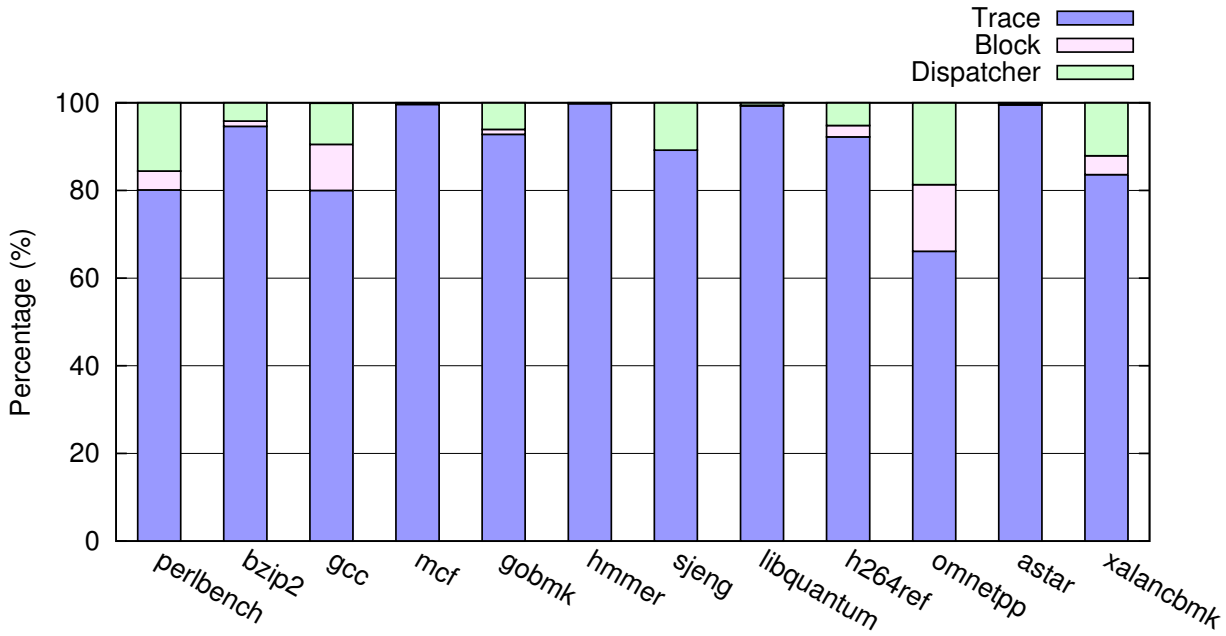
benchmarks with reference inputs. As the figure shows, most of the instructions are executed within the trace cache. On average, 90% and 95% of the total instructions in CINT and CFP benchmarks are executed within the trace cache, respectively. These results show that the optimized code from the LLVM translator is effectively utilized by the emulation thread most of the time in those benchmarks.

Overhead of Trace Merging

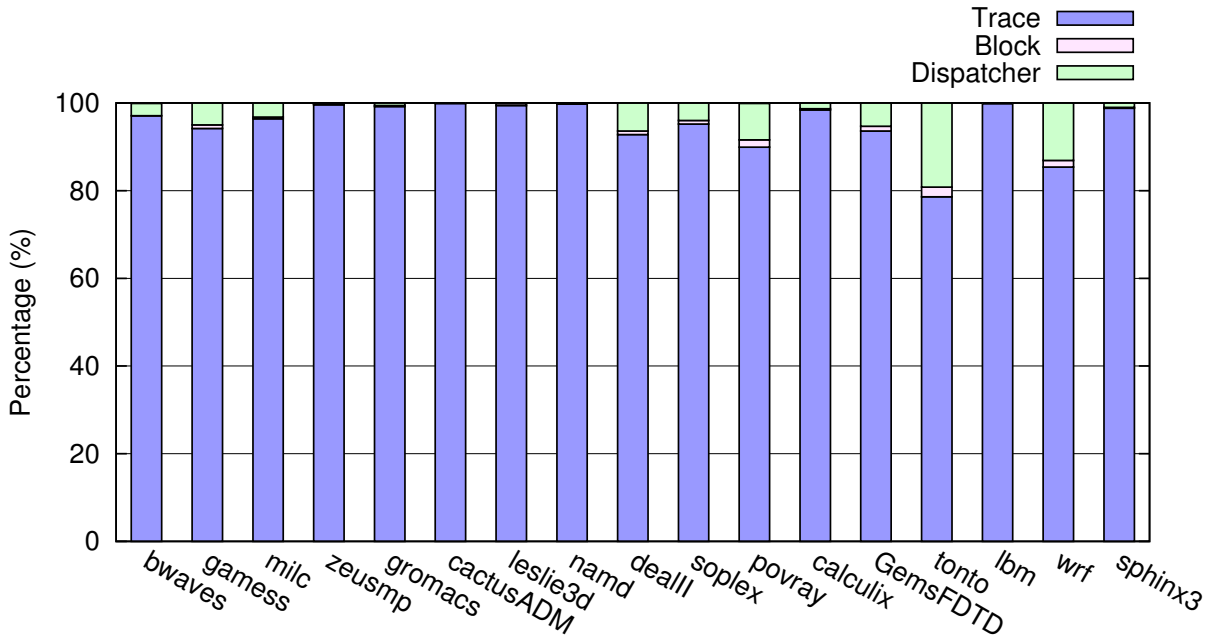
In this sub-section, we discuss the overhead of trace merging based on two approaches: HPM sampling and instrumentation. For HPM sampling, we measure the overhead by enabling HPM sampling but do not actually merge traces as they are detected. The HPM sampling overhead ranges from 0.7% (`milc`) to 3.7% (`xalancbmk`) and is 1.4% of the total execution time on average for SPEC CPU2006 benchmarks. As for instrumentation, we insert profiling and prediction routines of NET in code regions of all traces to detect cases for trace merging. When the hot traces form a cyclic path, these traces are merged. We present the overhead by comparing the performance of instrumentation-based approach to those of HPM sampling. The slowdown factor is normalized to the time of HPM sampling and is shown in Figure 3.15. The average overhead of instrumentation-based approach is about 24.9% for CINT and 11.7% for CFP. The results indicate that HPM sampling could be very cost effective for the detection of trace merging.

Comparison of LLVM Optimization Levels

To evaluate the impact of LLVM optimization, we compare the performance of SPEC2006 benchmarks with two different LLVM JIT optimization levels, -O0 and -O2. The results of -O1



(a) CINT



(b) CFP

Figure 3.14: Breakdown of time with x86 to x86-64 emulation for SPEC CPU2006 benchmarks with reference input. The results show that the execution is mostly within the trace cache—the optimized code from the LLVM translator is effectively utilized by the emulation thread most of the time.

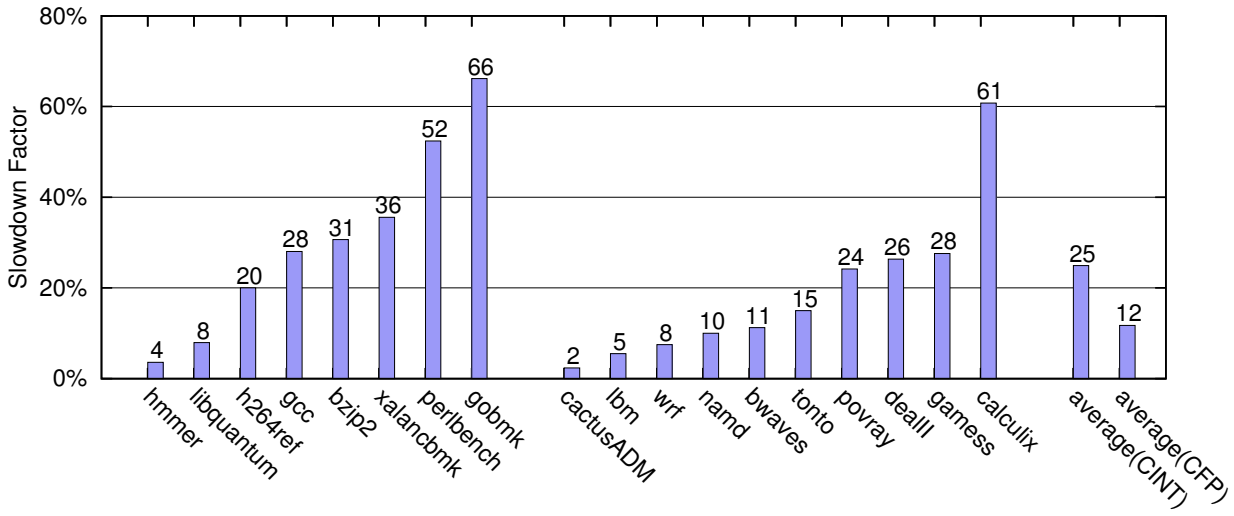
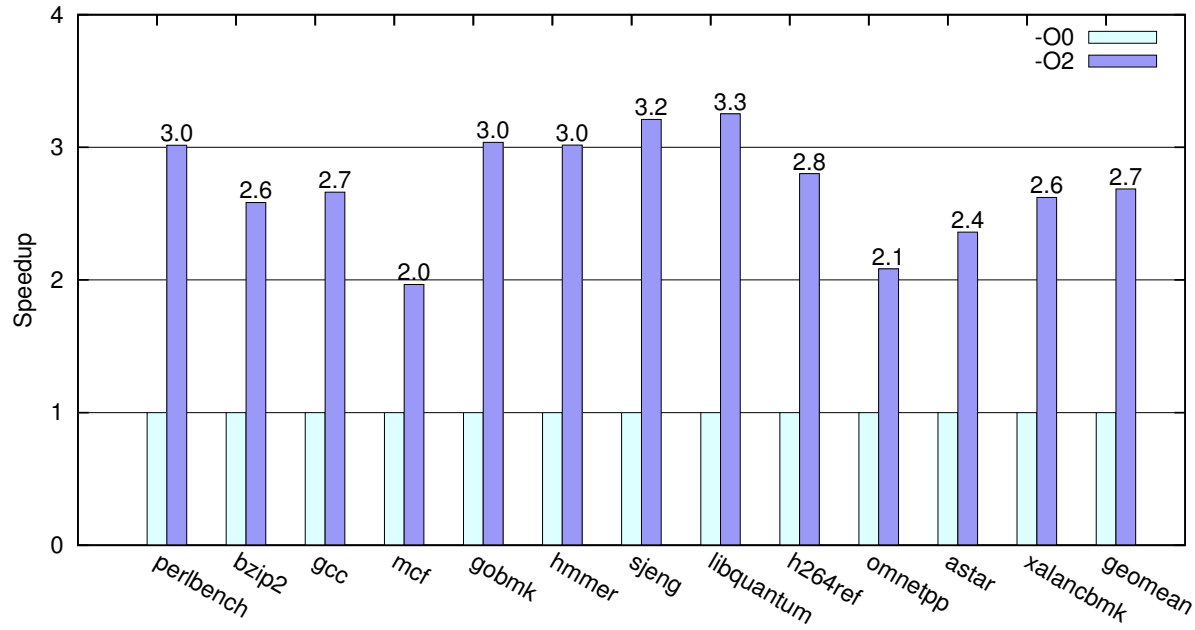


Figure 3.15: The slowdown factor of instrumentation-based approach over HPM sampling for trace merging.

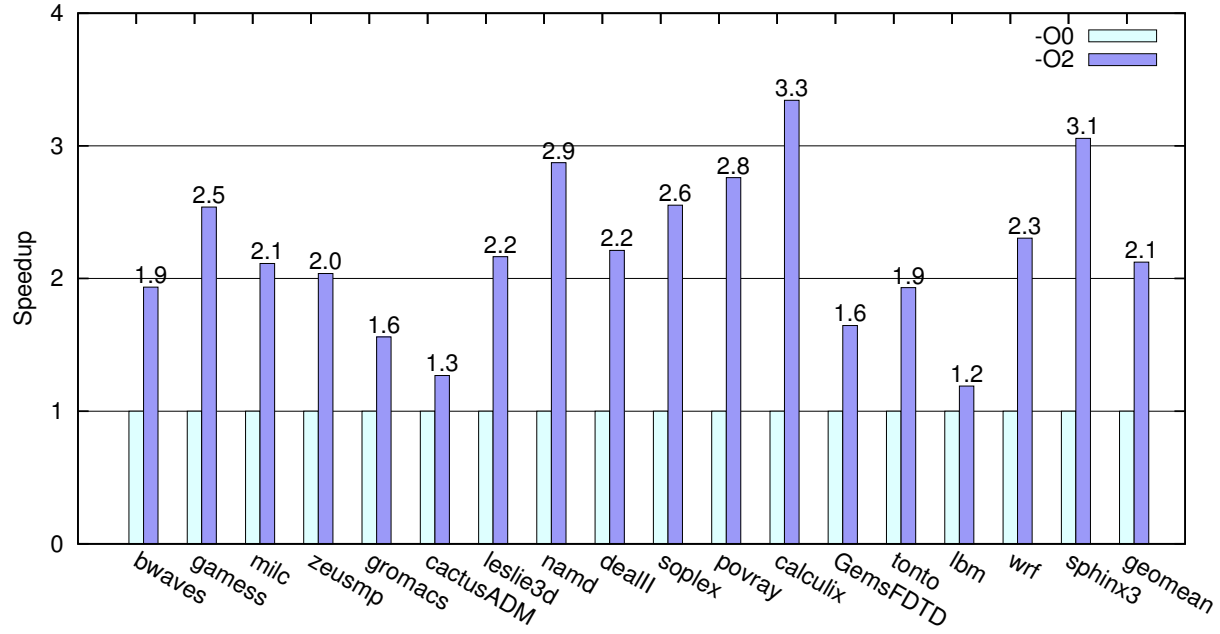
and -O3 are not shown because they use the same set of optimization passes as the -O2 optimization level and thus with the same performance. The -O0 level only applies a fast intra-block register allocation algorithm and dead code/block elimination; the -O2 level applies greedy global register allocation together with several optimizations such as LICM, machine code sinking, instruction scheduling, peephole optimizations, and sophisticated pattern matching for instruction selection, etc. Figure 3.16 shows the performance speedup of -O2 over -O0 for x86-32 to x86-64 emulation with reference inputs. As the result shows, the performance is improved by about 2.7X and 2.1X on average by applying such aggressive optimizations for CINT and CFP benchmarks, respectively. The performance gain mostly comes from better host instructions selected and better register allocation, which achieves minimal stack memory operations.

3.8 Discussion

In this chapter, we described the implementation of the hybrid QEMU (frontend) + LLVM (backend) DBT framework. The proposed architecture presents four major contributions. First, by leveraging the multicore architecture and multithreaded approach, we successfully address the dual issues of high-quality translated code and low translation overhead. We showed that this approach can be beneficial to both short- and long-running applications. Second, the problem of code region transition overhead is overcome by enlarging the translation granularity. Small code regions are combined to a larger one through trace formation and trace merging. We presented a novel trace combination technique to improve existing trace selection algorithms. It could effectively combine/merge traces based on the information provided by the on-chip HPM. We demonstrated that such feedback-directed trace merging optimization can significantly improve the overall code performance and is cost-effective. Third, we also leverage the LLVM compiler



(a) CINT



(b) CFP

Figure 3.16: x86 to x86-64 emulation performance for CPU2006 benchmarks with reference inputs with different LLVM optimization levels. Y-axis shows the performance speedup over the optimization level of -O0.

infrastructure so that helper functions can be inlined in the code cache to eliminate overhead from helper function invocations. Finally, a two-level IR conversion is also used to simplify the translator design efforts tremendously. Such strategy enables HQEMU to support all guest architectures that are supported by QEMU.

Experimental results show that HQEMU could improve the performance by a factor of 2.6X and 4.1X over QEMU, and are only 2.3X and 2X slower than the native execution for x86 to x86-64 emulation using SPEC CPU2006 integer and floating point benchmarks, respectively. For ARM to x86-64 emulation, HQEMU shows a gain of 2.5X speedup over QEMU for the SPEC integer benchmarks. With HQEMU, cross-ISA emulation is getting closer to the performance of native runs.

Chapter 4

Region Formation

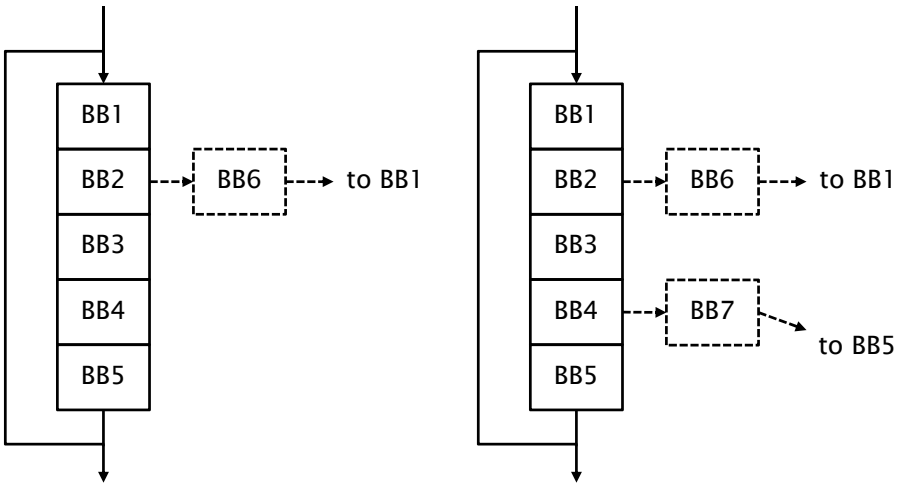
Now that we have introduced a hybrid two-translator DBT system and the multi-threaded strategy has been evaluated to successfully hide the optimization overhead, we can explore more optimization opportunities for the optimization threads. As we have learned from Chapter 3, the overhead of code region transitions can be eliminated by enlarging the translation granularity. We have successfully expanded the translation granularity from a block to a trace using trace formation, and from a trace to a region with the HPM-based trace merging. These two mechanisms are effective to keep the execution staying within a trace, as well as to eliminate the redundant loads and stores from code region transitions.

This chapter re-investigates the HPM-based sampling approach. Using the *hardware* sampling mechanism is lightweight, however, it has several shortcomings (to be explained in Section 4.2). In this chapter, we enhances HQEMU with an alternative *software-based* region formation approach, which is an extension to an existing region formation algorithm, called NETPlus. It can also improve existing trace formation algorithms like the HPM-based trace merging does, but reduce more emulation overhead of redundant memory operations. We will first introduce the software-based region formation approach in Section 4.1, and the comparison with the HPM-based approach is discussed in Section 4.2.

4.1 Software-Based Region Formation

Recall that the NET trace selection algorithm has the flaws that often lead to *trace separation* and *early exits*, both of which incur trace transition overhead—saving and restoring guest register states between host registers and main memory during each trace transition. It could significantly degrade the overall performance of typical retargetable DBT systems. NETPlus [25] enhances NET by expanding the trace formed by NET in the following steps.

NETPlus begins trace formation in the same manner as NET when the trace head is executed a certain number of times. It then begins adding subsequently executed blocks to the trace until a stopping condition is matched, such as another trace, a backward branch, or a maximum length is reached. However, when it finds a stopping condition, it will then perform a *forward search* for any loops back to the trace head by following all possible targets of branches. Trace formation



(a) NETPlus Region (b) Extended NETPlus Region

Figure 4.1: An example of code region formation with NETPlus and extended NETPlus region detection algorithms.

will then continue as long as the block to be added is along one of the paths found to be part of a loop back to the trace head. Figure 4.1(a) shows an example of the region formed by NETPlus. Along the side exit of `BB2` exists a path to `BB6` and then to the trace head `BB1`. Thus, the block `BB6` is included in the final NETPlus region.

The region formation algorithm used in HQEMU makes some changes to NETPlus:

1. The initial NET trace is formed by our relaxed NET algorithm as mentioned in Chapter 3.3 —the stop condition of backward branch is relaxed. We stop NET trace formation only when the next branch target is another trace head, a maximum length is reached, or the same program counter (PC) is executed again.
2. HQEMU performs the forward search by following the target of *direct* branches only. (We assume NETPlus follows any branches including direct and indirect branches because no specific description about this is mentioned in its paper.)
3. Our forward search along the side exits does not confine to the cyclic paths back to the trace head as NETPlus does. HQEMU includes the blocks of any path that would link back to any block of the NET trace body.

A threshold is also set in order to limit the maximum depth of the forward search. Figure 4.1(b) illustrates an example of our region formation approach. In Figure 4.1(b), our approach can form the region of CFG containing the paths of both `BB2-BB6-BB1` and `BB4-BB7-BB5`, where the path `BB4-BB7-BB5` is not allowed with the NETPlus algorithm.

As we have learned from the experiment results in Chapter 3, the separation of code regions, either among blocks or among traces, is one of the major performance problems of QEMU, and significant performance improvement can be achieved through combining those separate code regions with frequent transitions. With the original NETPlus algorithm, the searched paths that

go to the body but not the head of the trace (e.g. trace of `BB7-BB5`) are not included in the final region and will become separate traces—causing potential transition overhead. This is the reason why we extend the NETPlus algorithm and include every possible path that links back to the trace body. We try to form the regions that can keep the execution staying within it as possible as we can.

Although this extension could make the CFG of a formed region become more complex and might miss some LLVM optimization opportunities, we expect the aggressive optimization passes of LLVM could still produce efficient code. Furthermore, the increased optimization overhead due to larger compilation granularity is acceptable because it will be off-loaded to another threads with HQEMU.

4.2 Analysis of the Software-Based and HPM-Based Region Formation

Although the HPM sampling mechanism is lightweight, it has several shortcomings. First, the performance monitoring hardware has accuracy issues in either time-based or event-based sampling [18, 69]. Second, it needs to collect sufficient sample profiles in order for accurate analysis and optimizations [18]—often costs a sufficiently long period. Hence, better optimization results usually come from sampling long programs, and short-running programs are sometimes not improved much. And finally, the hardware counters currently are not virtualized. Therefore, the hardware performance monitoring cannot be used under a virtualized environment, e.g. in a guest virtual machine.

The major problem of the sampling-based mechanism is that it is likely to miss some optimization opportunities due to missing some important sample profiles. For example, we would like to merge the two traces, `BB6` and `BB1-to-BB5` in Figure 4.1(b), if they have frequent trace transitions among them. Using a moderate sampling rate, unfortunately, it is highly possible that the sample profiles are always collected from the large trace `BB1-to-BB5`, and the trace `BB6` is never sampled if it consists of very few instructions. Thus, the merging of these two traces may never be activated throughout the program's lifetime.

The problem with such small trace can be overcome by NETPlus because it will be selected and combined with the NET trace. Moreover, the software-based region formation can find the hot paths earlier in the program execution, compared to the HPM-based one. This is beneficial especially for the emulation of short-running applications. Finally, the software-based approach can be applied in any virtualized environment.

Different to the HPM-based region formation that only merges those traces having been executed, the software-based region formation expands the region by *prediction*. Hence, the predicted paths (i.e. the paths found via forward search) might not be executed—the generated host code and additional compilation time are wasted.

4.3 Performance Results

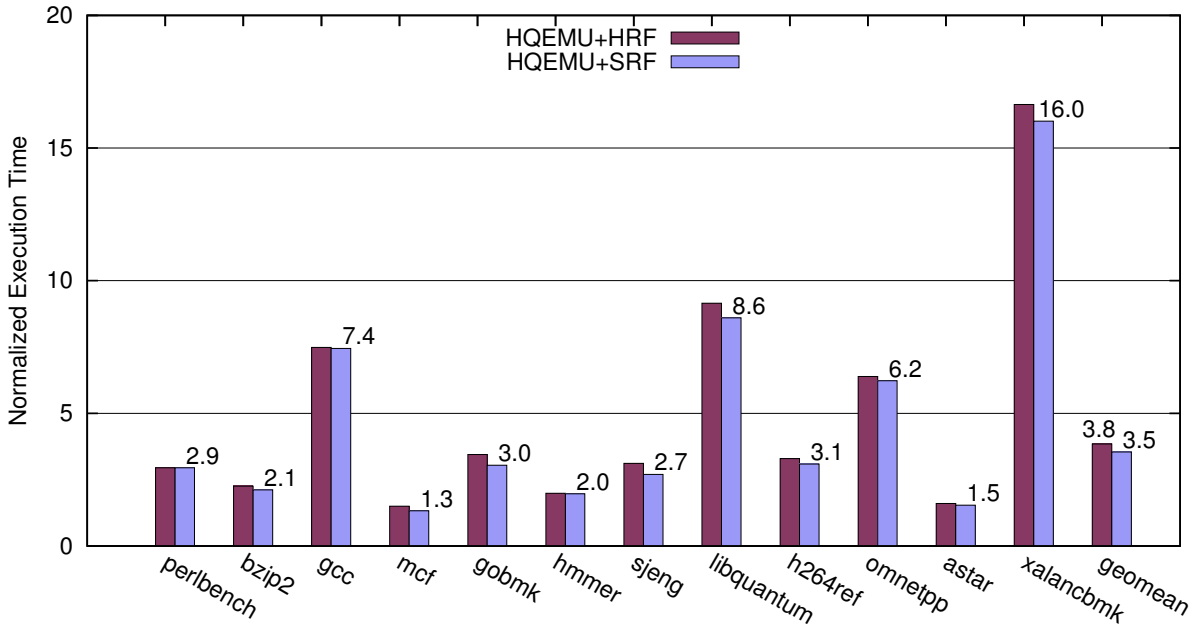
In this section, we report the performance comparison of the HPM-based and NETPlus-based region formation algorithm in HQEMU using single-thread benchmarks. SPEC CPU2006 benchmark suite is used in this experiment, and the software and hardware settings are the same as those in Section 3.7. We conduct the evaluation with one same-ISA translation and one cross-ISA translation, which performs x86/32-to-x86/64 and ARM-to-x86/64 emulation respectively. The maximum depth of the forward search is set to 8 basic blocks. In the following figures, we measure the normalized execution time over the native run for the two mechanisms: hardware-based region formation (**HQEMU-HRF**, HPM-based) and software-based region formation (**HQEMU-SRF**, NETPlus-based).

Figure 4.2 presents the x86-32 to x86-64 emulation results of the integer benchmarks with test and reference inputs. As for the results of test inputs shown in Figure 4.2(a), the performance of HQEMU-SRF is the same as or slightly better than HQEMU-HRF for most benchmarks. The slight performance gain is mostly because the HPM sampling is disabled with the software-based approach, thus, fewer overhead is incurred. `gobmk` and `sjeng` are the two benchmarks that have more performance improvement with HQEMU-SRF. This is because these two benchmarks suffer frequent early exits within many hot traces due to unbiased branches. The hardware-based sampling does not gather enough sample profiles to trigger trace merging for the short runs. In contrast, the software-based approach combines these separate hot traces to a region early in the program execution, and thus the transition overhead can be eliminated.

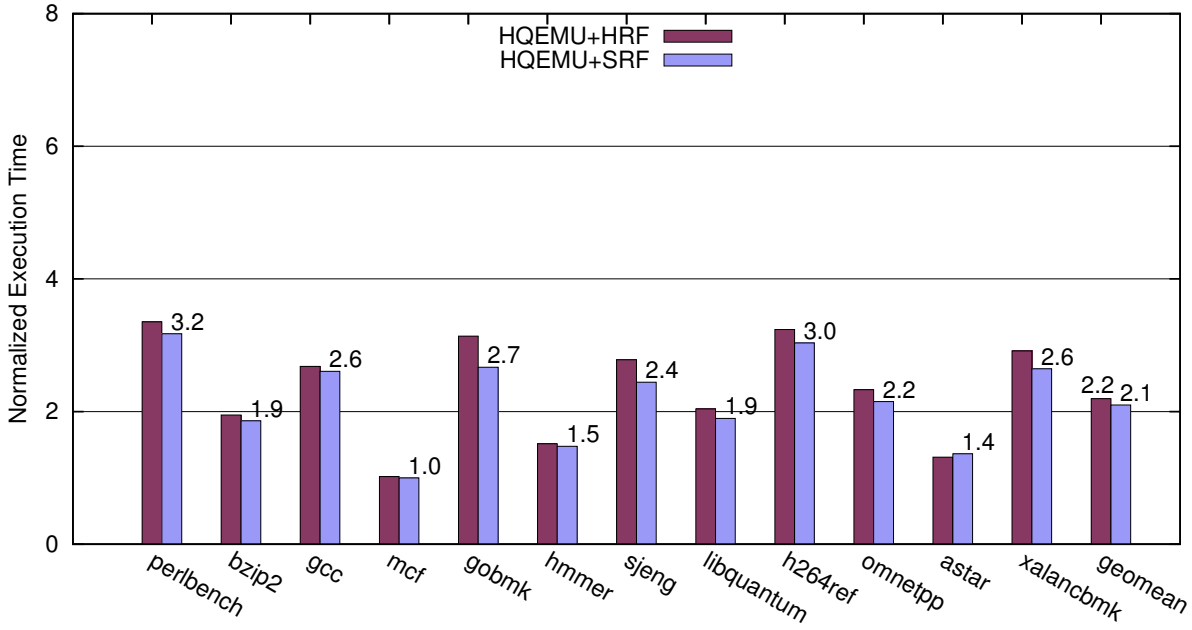
In Figure 4.2(b), performance improvement is also observed for these two benchmarks with reference inputs. Ideally, the hardware-based approach should be able to merge the problematic traces because the execution time is long enough to collect sample profiles. However, it still misses merging some important hot traces because they are too small to produce sufficient samples. In contrast, such small traces can be found with the software-based region formation, thus, the performance can be enhanced. As for `libquantum`, HQEMU-SRF also successfully combines the two separate traces in Figure 3.4(b) into one big code region with its CFG shown in Figure 3.4(a), thus it does not suffer the huge trace transition overhead of using NET trace formation only. Furthermore, HQEMU-SRF can combine more small hot traces. This is why the software-based approach can outperform the hardware-based region formation for `libquantum`.

As shown in Figure 4.2(b), the performance of HQEMU-SRF is a little slower than that of HQEMU-HRF for benchmark `astar`. This is because the CFG of the hot region of `astar` contains a long depth of basic blocks. Since we limit the maximum search depth to 8 basic blocks for HQEMU-SRF, it cannot cover the hot blocks over 8 search steps. However, the depth of the region with HQEMU-HRF is lengthened after each trace merging (we can see in column 5 of Table 3.2, the maximum version of the trace(s) for `astar` is 8, meaning that the region consists of at least 8 traces). Hence, the region formed by the hardware-based approach is better.

As for the emulation of floating point benchmarks, the results are not shown because no noticeable performance difference are observed between the two region formation mechanisms. This is because both approaches perform well for combining separate traces with the benchmarks.

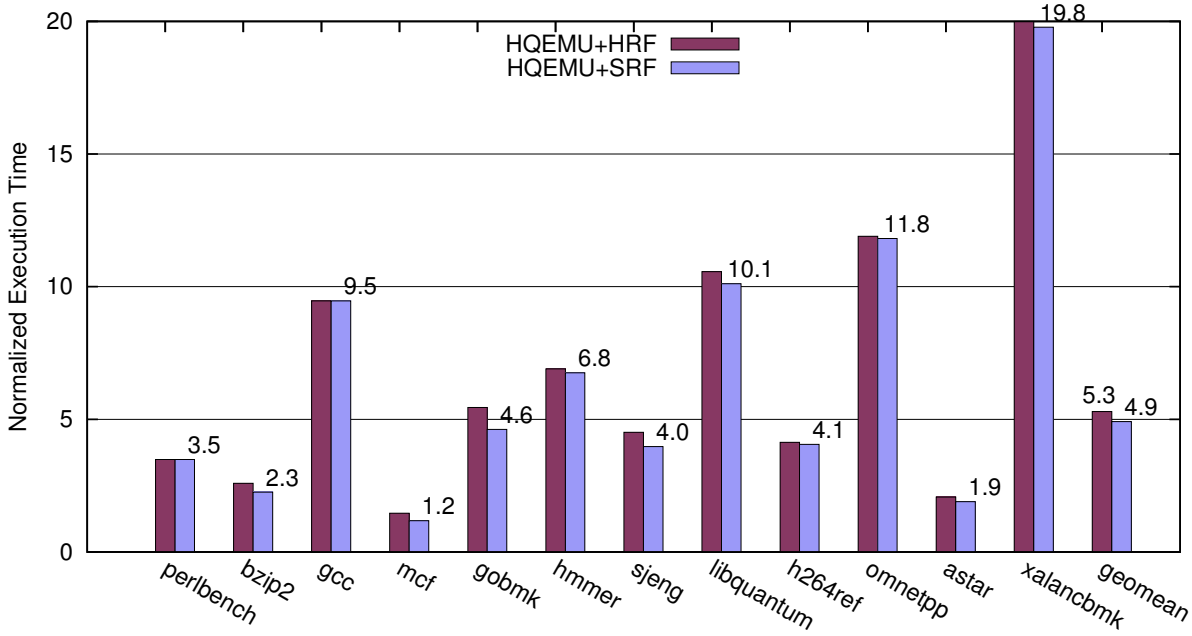


(a) Test input

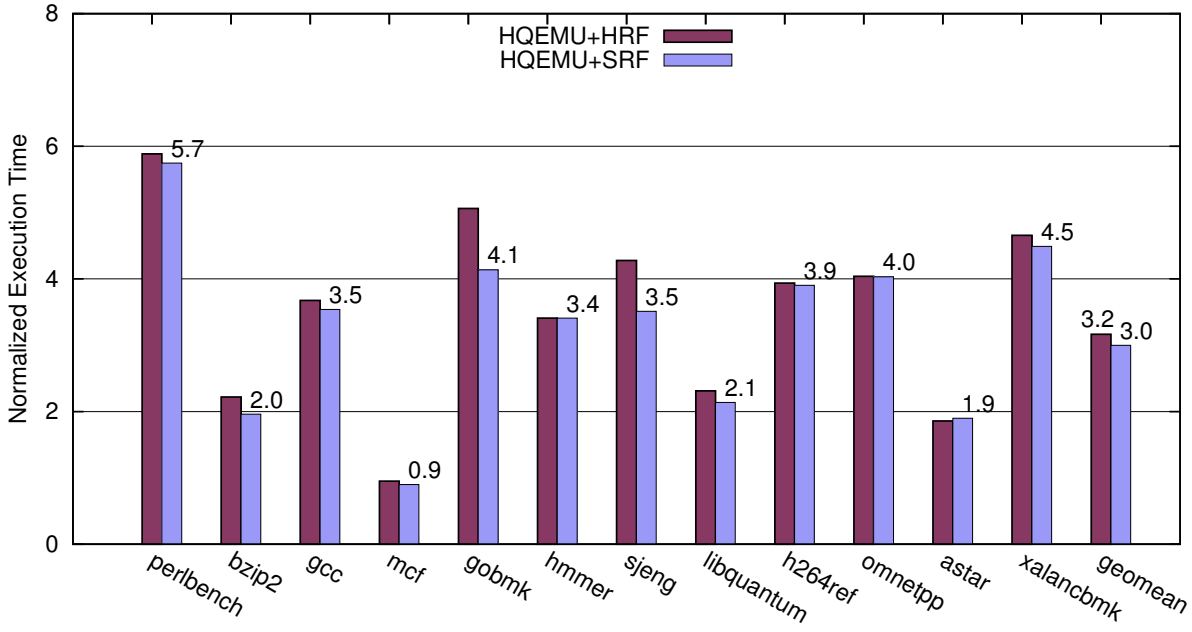


(b) Reference input

Figure 4.2: Comparison of hardware-based and software-based region formation for x86-32 to x86-64 emulation with CINT2006 benchmarks. Y-axis shows the normalized execution time over native run.



(a) Test input



(b) Reference input

Figure 4.3: Comparison of hardware-based and software-based region formation for ARM to x86-64 emulation with CINT2006 benchmarks. Y-axis shows the normalized execution time over native run.

Figure 4.3 shows the comparison results of ARM to x86-64 emulation for integer benchmarks. The results are similar to those of emulating x86-32 guest. Benchmark `gobmk` and `sjeng` have the most significant performance gain with HQEMU-SRF over HQEMU-HRF. On average, these two benchmarks are improved about 15% and 22% with the software-based region formation for x86-32 and ARM guest, respectively.

4.4 Summary

Although the HPM sampling mechanism is lightweight for performing trace merging, it has the issues of sampling accuracy, long profiling delay and limited usage for virtual machine. With the same goal of the HPM sampling approach, the software-based region formation based on the NETPlus algorithm is also able to expand the NET trace, and solve the problem of trace separation and early exits.

Through the predictive forward search, the software-based region formation combines the potential separate traces early in the program execution. This is helpful for emulating short-running applications, where the hardware-based sampling may not be able to gather sufficient meaningful sample profiles in a short run. The small separate traces that consist of few instructions are hard to be detected by the hardware-based sampling. But they can be also combined with the software-based approach. Therefore, the software-based region formation is beneficial to both short-running and long-running applications.

Chapter 5

Optimizing Performance Scalability

QEMU has two modes in emulating a application binary: (1) *full-system emulation*, in which all OS kernels involved are also emulated, and (2) *process-level emulation* (the focus of this dissertation), in which only application binaries are emulated. When running a multithreaded application in process-level emulation, QEMU creates one host thread for each guest thread, and all these guest threads are emulated concurrently.

In Chapter 2, we have learned that the overhead of a DBT becomes more critical to the overall performance when it comes to translating multi-threaded applications. Severe contentions on shared data structures in a DBT can incur substantial synchronization overhead when multiple application threads are being translated simultaneously. Moreover, the emulation of atomic instructions, which are required to implement synchronization primitives in multithreaded guest applications, becomes more challenging not only for its correctness but for its efficiency as well. The overhead of synchronization could result from poor implementation of atomic regions when emulating guest atomic instructions.

As a large number of threads needs to be translated and optimized at runtime, any inefficiency in handling globally shared data structures and the emulation of atomic instructions in a DBT could be amplified many times over. This chapter addresses these inefficient designs of QEMU and describes two optimization strategies, *indirect branch translation caching* and *lightweight memory transactions*, used in HQEMU to mitigate those problems.

5.1 Indirect Branch Handling

QEMU uses a globally-shared code cache, i.e. all executing threads share a single code cache, and each guest block has only one translated copy in the shared code cache. A single directory (i.e. hash table) that records the mapping from a guest code block to its translated host code region is maintained by all threads. An execution thread first looks up the directory to locate the translated code region. If not found, it kicks start the TCG to translate the untranslated guest code block. Since the code cache, the directory and the TCG translator are shared by all execution threads, QEMU uses a *critical section* with a coarse-grained lock to serialize all accesses to the shared structures. Such a design yields very efficient memory space usage, but it could cause

severe contention to the shared code cache and directory when a large number of guest threads are emulated.

5.1.1 Indirect Branch Translation Cache

Indirect branches, such as *indirect jump*, *indirect call* and *return* instructions, cannot be linked in the same way as *direct* branches because they can have *multiple jump targets*. Making the execution threads go back to the dispatcher for the branch target translation each time when an indirect branch is encountered may cause huge performance degradation. The degradation is due to the overhead from (1) saving and restoring program contexts when a context switch occurs, and (2) the contention for the shared directory (protected in a critical section) to find the branch target address when a large number of threads are emulated.

To mitigate this overhead, we try to avoid going back to the dispatcher for branch target translation. For the indirect branches that leave a block or exit a trace, the *Indirect Branch Translation Cache* (IBTC) [80] is used. The translation of an indirect branch target with IBTC is performed as a fast hash table look-up inside the code cache. Upon an IBTC miss, the execution thread still goes back to the dispatcher, performs expensive translation of indirect branch with the shared directory, and caches the lookup result in the IBTC entry. Upon an IBTC hit, the execution jumps directly to the next translated code region so that a context switch back to the dispatcher is not required. The IBTC in our framework is a big hash table shared by all indirect branches, including indirect jump/call and return instructions. That is, the translation of branch targets looks up the same IBTC for all indirect branches. Our IBTC is much like a direct-mapped cache, meaning that each cache entry has only one slot and is replaced if there is a conflict miss. Furthermore, we set up one IBTC for each execution thread. Such thread-private IBTC can avoid contention during the branch target translation.

5.1.2 Indirect Branch Inlining

During trace formation, the prediction routine might record two successive blocks following the path of an indirect branch. For those blocks in a trace with path via indirect branch, we use the *Indirect Branch Inlining* (IB *inlining*) approach [17] to facilitate the translation of indirect branch target. When translating an indirect branch instruction in the predecessor block, a comparison of the value in the indexing register against the address of the successor block is inlined in the trace code. Upon a match, the execution can continue to the translated code of the successor block without leaving the trace. If the comparison does not match, meaning that the prediction fails, this indirect branch will leave the trace and its execution is also redirected to the IBTC. Such IB inlining is advantageous because such an indirect branch must be hot to be selected in a trace, and thus the prediction of this comparison is most likely to be accurate.

With IBTC and IB inlining, we effectively reduce the overhead by keeping the execution threads staying in the code cache for as long as possible.

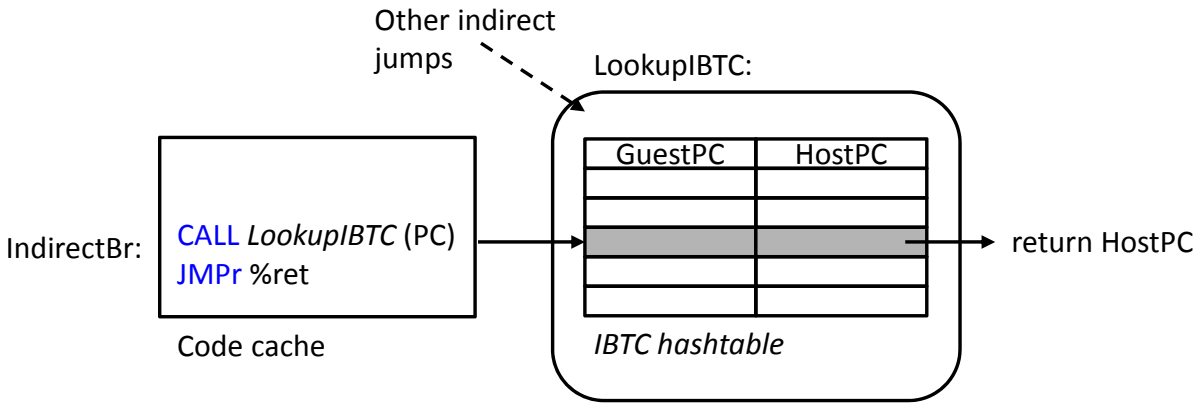


Figure 5.1: Implementation of IBTC hash table lookup routine in HQEMU. The IBTC lookup is implemented as a helper function. The IBTC hash table is declared with the `__thread` identifier, and thus, each execution thread has its own private IBTC hash table and no table locking is required.

5.1.3 Implementation of Indirect Branch Translation Cache

Figure 5.1 illustrates the implementation of our IBTC hash table. The IBTC lookup is implemented as a *helper function*. The function takes the guest address as its parameter and returns the corresponding host address. Each indirect branch instruction (the left box in Figure 5.1) is translated as calling the lookup routine and then jumping indirectly (JMPr) to the address returned from this lookup routine. The IBTC hash table is declared with the `__thread` identifier. Thus, each execution thread has its own private IBTC hash table and no table locking is required. This implementation has the following advantages: (1) the use of helper function is portable across different host architectures, (2) the threads incur no locking overhead while performing table lookup, and (3) Luk [66] and Kim [55] reported that the hardware BTB prediction rate of the indirect jump will be poor if the lookup's indirect jump is shared by all indirect branches in the application (i.e. all indirect jumps lead to the same dispatch code and a single BTB entry is required to provide all the target addresses). HQEMU does not suffer such problem because we inline the IBTC lookup's indirect jump (i.e. the *JMPr* instruction in Figure 5.1) in the code cache.

The performance of IBTC hash table depends on its number of hash entries. In HQEMU, we set the number of entry to 65536 and it can achieve more than 99% lookup hit rate for all CPU2006 benchmarks with reference inputs.

We also implemented another IBTC with Pin's indirect branch chain [66] for comparison purpose. Figure 5.2 shows the performance of the indirect branch chain approach (using the IBTC hash table approach as the baseline for comparison) for CINT2006 benchmarks with reference inputs. In this experiment, the maximum chain length is set to 16. We made one change to original Pin's algorithm that when the list of chain reaches the maximum chain length, the address comparison will go to our IBTC hash table of replacement instead of using another hashed comparison chain list. We also compare the performance of attaching a new chain entry to the

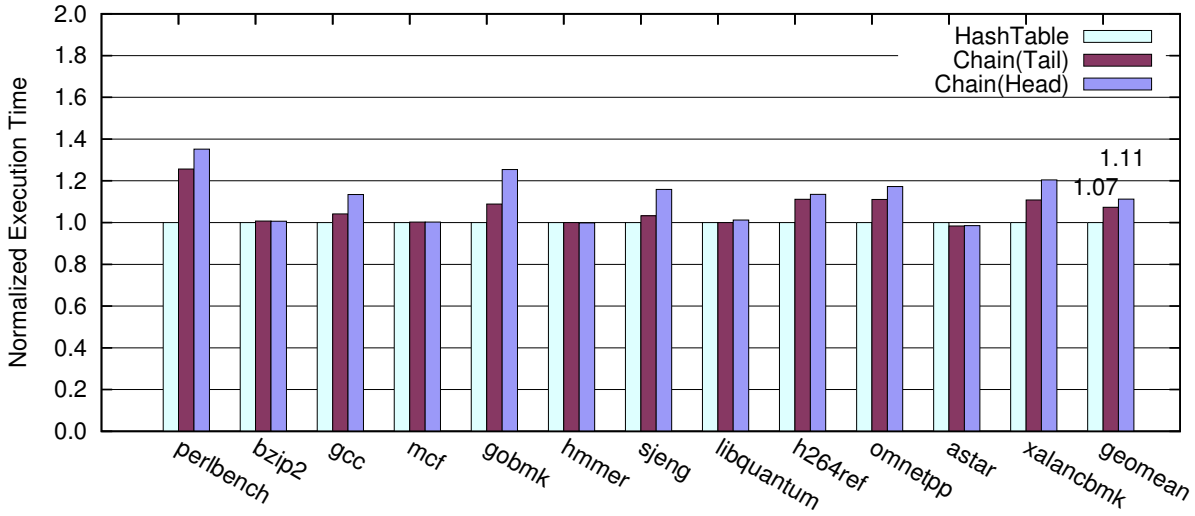


Figure 5.2: Performance comparison of Pin's indirect branch chain with IBTC hash table for CINT2006 with reference inputs. Tail and Head refer to attaching a new chain entry to the tail or head of the chain list, respectively. Y-axis shows the normalized execution time over using IBTC hash table.

	perl	bzip2	gcc	mcf	gobmk	hmmer	sjeng	libqua.
Tail	6.69	5.50	3.44	1.03	4.52	1.22	3.44	3.22
Head	7.15	5.25	5.27	1.47	7.81	3.02	5.42	1.36
	h264ref	omnetpp	astar	xalanc.				
Tail	5.77	3.92	1.02	3.22				
Head	5.67	4.23	1.01	3.56				

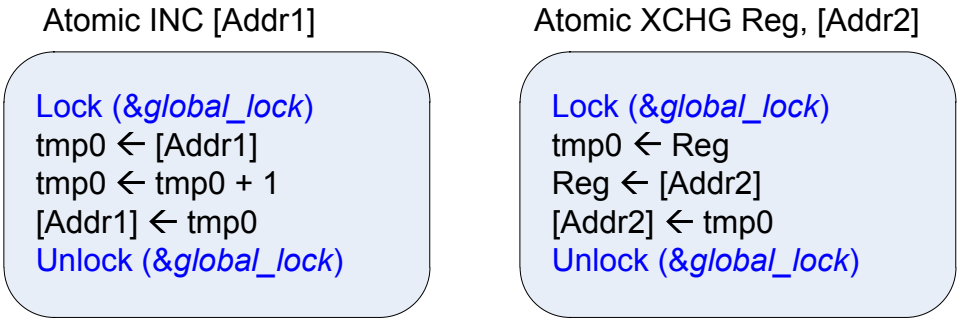
Table 5.1: Average search depth of indirect branch chain for CINT2006 with reference inputs.

tail or head of the chain list.

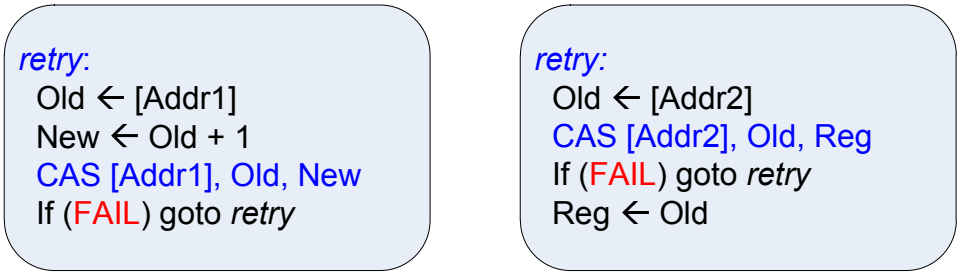
As Figure 5.2 shows, the indirect branch chain approach results in about 7% and 11% slowdown on average with attachment in tail and head, respectively, compared with IBTC hash table. Table 5.1 lists the average search depth with indirect branch chain. The result shows that the indirect branch chain approach requires several steps to reach the matched chain entry. Since HQEMU can achieve 99% hit rate (i.e. 99% indirect branches executed requires only one comparison), we demonstrate that our IBTC hash table can achieve better performance.

5.2 Atomic Instruction Emulation

The emulation of *guest atomic instructions*, which are usually used to implement synchronization primitives, poses another design challenge. The correctness and efficiency of emulating atomic instructions are critical to a DBT system, especially when emulating multi-threaded ap-



(a) The same global lock variable *global_lock*



(b) Lightweight memory transactions

Figure 5.3: An example of translating two atomic instructions using (a) global lock and (b) lightweight memory transactions. QEMU uses the same global lock variable surrounding the translated code region of all guest atomic instructions. HQEMU uses lightweight memory transactions to provide the atomic region for all such critical sections.

plications. To ensure the correctness, the DBT system must guarantee that the translated host code be executed atomically. To emulate the *atomicity* of a translated *guest atomic instruction*, QEMU uses a simple approach by surrounding the translated code region that corresponds to the *guest atomic instruction* with a *lock-unlock pair* to make it a *critical section* on the host machine. Thus, concurrent accesses to the emulation code by different threads are serialized. However, QEMU uses the *same lock variable* for *all* such critical sections. Figure 5.3(a) shows how two guest atomic instructions, *atomic INC* and *atomic XCHG*, are translated into critical sections protected by lock-unlock pairs using the *same* lock variable *global_lock*. The reason that QEMU uses the same global lock variable for all such critical section is because the DBT cannot determine if any two memory addresses are aliases at the translation time.

Although the global lock scheme of QEMU is portable, it has several problems: (1) Wang et al. [87] proved that this approach could still have correctness issues that may cause deadlocks; (2) accesses to non-aliased memory locations (e.g. two independent mutex variables in the guest source file) by different threads are serialized because of the same global lock; (3) the performance is poor due to the high cost of the locking mechanism.

The overhead of accessing the global lock depends on the design of the locking mechanism. For example, the locking mechanism in QEMU is implemented using NPTL [72] synchroniza-

Optimization flags	
Native-x86/64	-O3 -funroll-loops -fprefetch-loop-arrays
Native-ARM	-O3 -funroll-loops -fprefetch-loop-arrays -mfpu=vfp
Guest-x86/32	-O3 -funroll-loops -m32 -msse2 -mfpmath=sse

Table 5.2: Optimization flags for PARSEC benchmarks.

tion primitives, which use Linux *futex* (a fast user-space mutex) [34]. When an execution thread fails to acquire or release the global lock, the thread is put to sleep in a wait-queue, and is waken later via an expensive futex system call. Such expensive switching between user and kernel mode and the additional contention caused by false protection of non-aliased memory accesses could result in significant performance degradation.

To solve the problems incurred by the global lock, we use *lightweight memory transactions* proposed in [87] to address the correctness issues, as well as to achieve efficient atomic instruction emulation. The lightweight memory transaction based on the multi-word compare-and-swap (CASN) algorithm [41] allows translated code of atomic instructions to be executed optimistically. It detects data races while emulating an atomic instruction using the atomic primitives supported by the host architecture, and re-executes this instruction until the entire emulation is atomically performed. Figure 5.3(b) illustrates the translation of the same two guest atomic instructions using lightweight memory transactions. At first, the value of the referenced memory is loaded to the temporary register, *Old*. The new value after the computation is atomically stored in the memory if the value in the memory is the same as *Old*. Otherwise, the emulation keeps retrying if the CAS transaction fails.

Based on this approach, the protection of memory accesses with a global lock can be safely removed because the lightweight memory transactions can guarantee correct emulation of atomic instructions. Moreover, the performance will not degrade much because the false protection of non-alias memory accesses and the overhead of expensive locking mechanism are eliminated as a result of the removal of global lock.

5.3 Performance Results

In this section, we evaluate the emulation performance of HQEMU for multi-threaded programs. PARSEC [13] version 2.1 is used as the testing benchmarks. The performance comparison of using global lock and lightweight memory transactions is also reported.

Experimental Setup

The experiments are conducted on two systems: (1) eight six-core AMD Opteron 6172 processors (48 cores in total) with a clock rate of 2 GHz and 32 GBytes main memory, and (2) the ARM platform listed in Table 3.1. The PARSEC benchmarks are evaluated with the `native` and `simlarge` input sets for x86-64 and ARM platform, respectively. All benchmarks are parallelized with the Pthread model and compiled for x86-32 guest ISA. Table 5.2 lists the compiler

optimization flags used. We compare their performance to native execution with three different configurations:

- **QEMU** which is the vanilla QEMU version 0.13 with the fast TCG translator.
- **QEMU-Opt** which is the enhanced QEMU with IBTC optimization and block chaining across page boundary.
- **HQEMU** which is the multi-threaded HQEMU with IBTC optimization and the extended NETPlus-based region formation.

For all configurations, atomic instructions are emulated with lightweight memory transactions so that the benchmarks can be emulated correctly.

Overall Performance of PARSEC

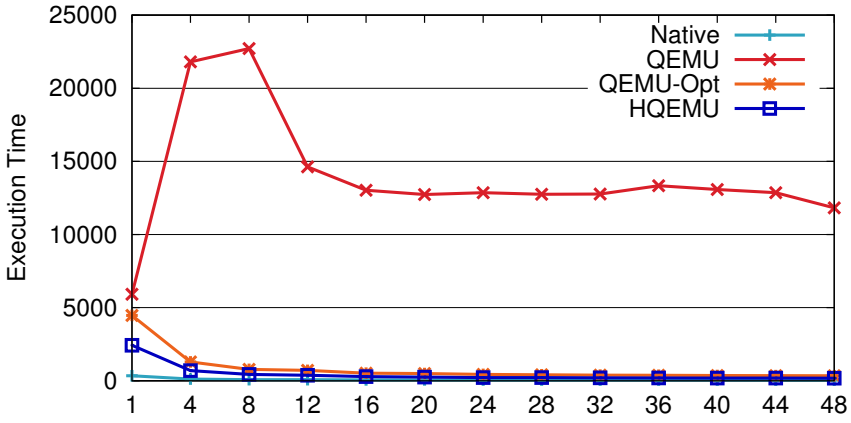
Figure 5.4 illustrates the performance results of each PARSEC benchmark with native input sets. The X-axis is the number of worker threads created via the command line argument. The Y-axis is the elapsed time measured with the `time` command. As shown in Figure 5.4(a) to Figure 5.4(i), the performance of QEMU does not scale well for all benchmarks. In all sub-figures, the execution time increases dramatically when the number of guest threads increases from 1 to 8, then decreases with more threads. It remains mostly unchanged as the number of threads is above 16. The poor scalability of QEMU is mostly due to the sequential translation of branch targets within the QEMU dispatcher because the mapping directory is protected in a critical section. Since IBTC optimization is not applied in QEMU, the execution threads frequently enter dispatcher for branch target lookups. Although the computation time can be reduced through parallel execution with more threads, the overhead incurred by thread contention can result in significant performance degradation.

With IBTC optimization, the huge overhead that includes the cost of context switches and thread contention in the dispatcher, is alleviated by keeping the execution threads staying in the code cache. Performance gains from IBTC can be observed from the comparison of QEMU-Opt and QEMU.

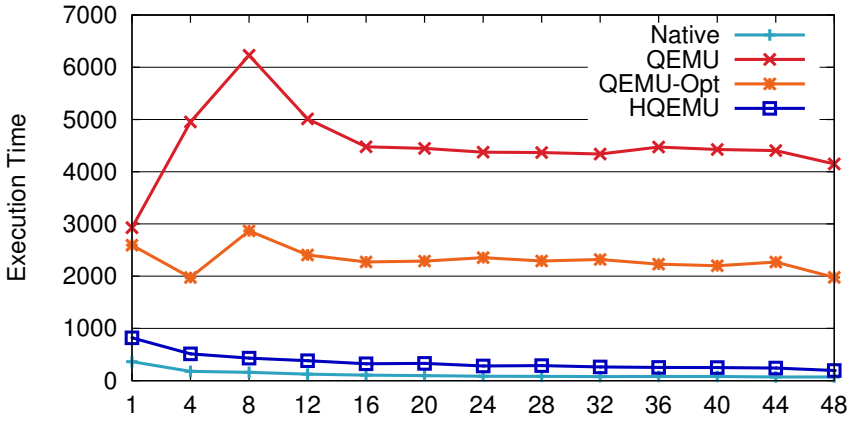
The region formation of HQEMU further improves the indirect branch prediction via indirect branch inlining. It also eliminates a large amount of redundant load/store instructions related to architecture state emulation. Highly-optimized host code generated by LLVM makes HQEMU achieve significant performance improvement over both QEMU and QEMU-Opt. The performance curve with HQEMU is very similar to that of native execution.

Figure 5.5 shows the slowdown factors of QEMU, QEMU-Opt and HQEMU over native execution with 32 guest threads[1]. The slowdown factors of QEMU over native execution range from 10X to 830X, and the geometric mean is 86X. Significant improvement, about 6X speedup on average, is achieved with QEMU-Opt over QEMU because of the IBTC optimization. This result shows that the design of shared mapping directory within critical section in the QEMU dispatcher is inadequate for emulating multi-threaded guest applications. HQEMU achieves about 26X and 4.3X speedup compared with QEMU and QEMU-Opt, respectively, and its slowdown

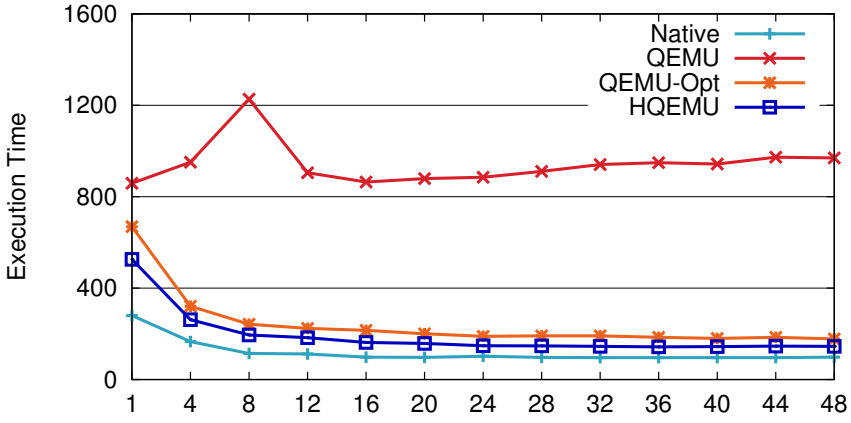
[1] We use 32 threads because it is the maximum available thread number for all benchmarks.



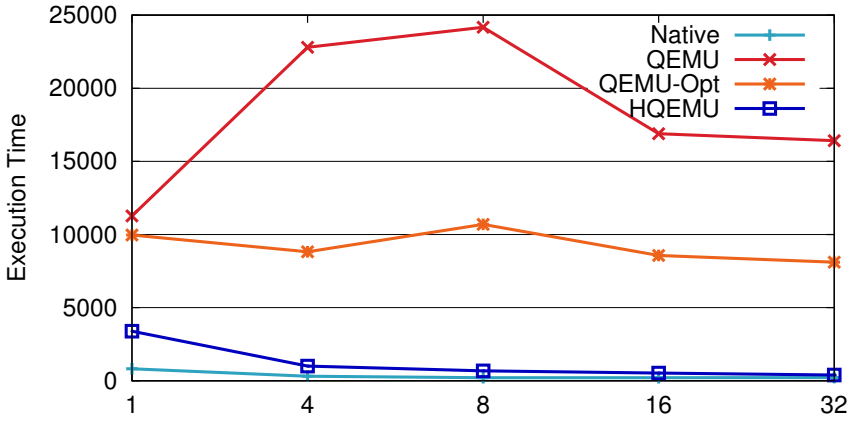
(a) blackscholes



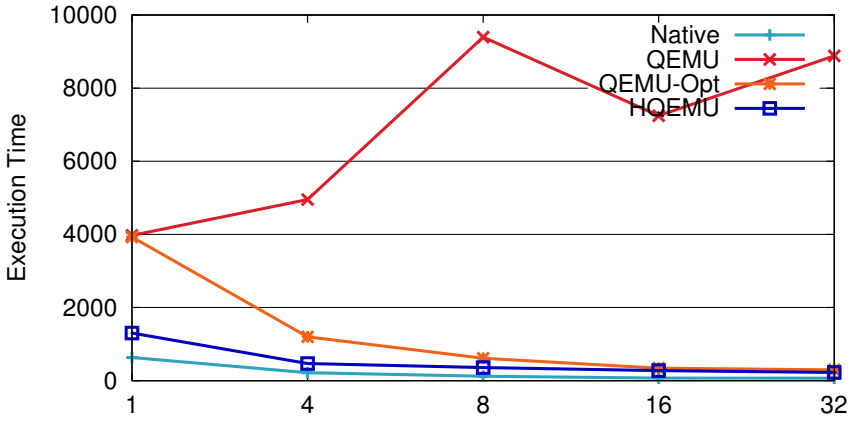
(b) bodytrack



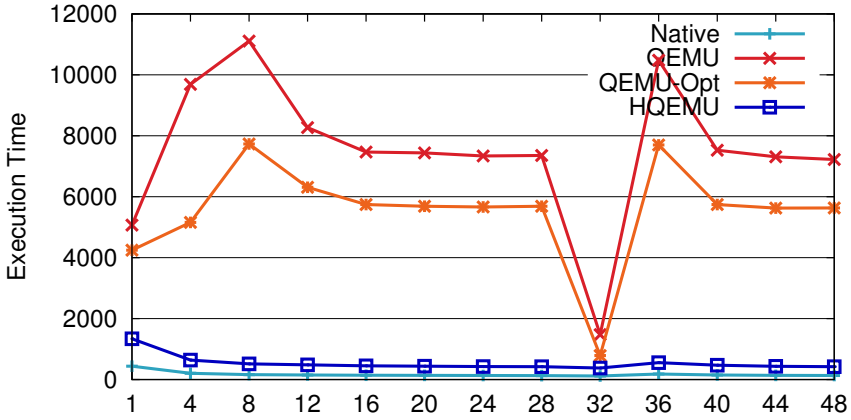
(c) canneal



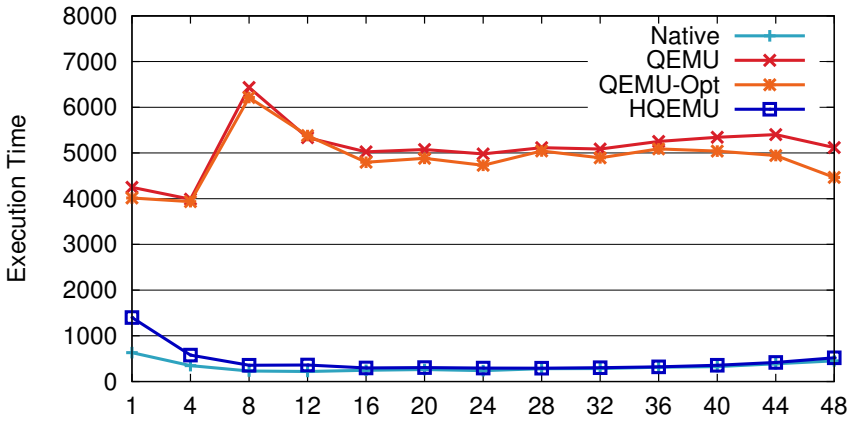
(d) facesim



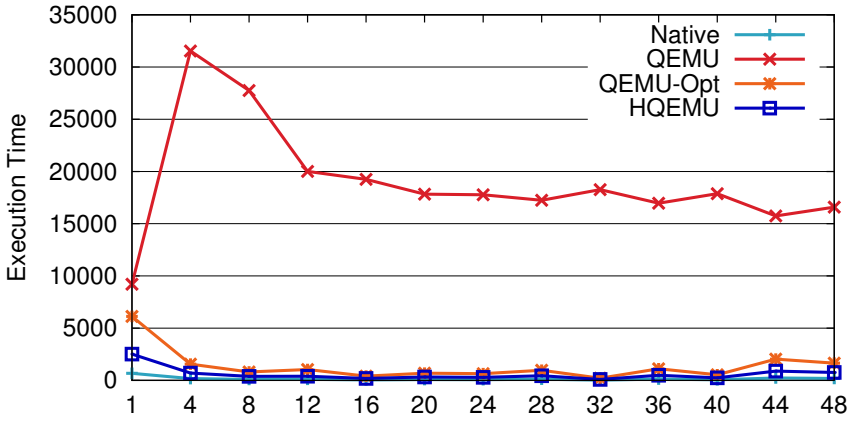
(e) fluidanimate



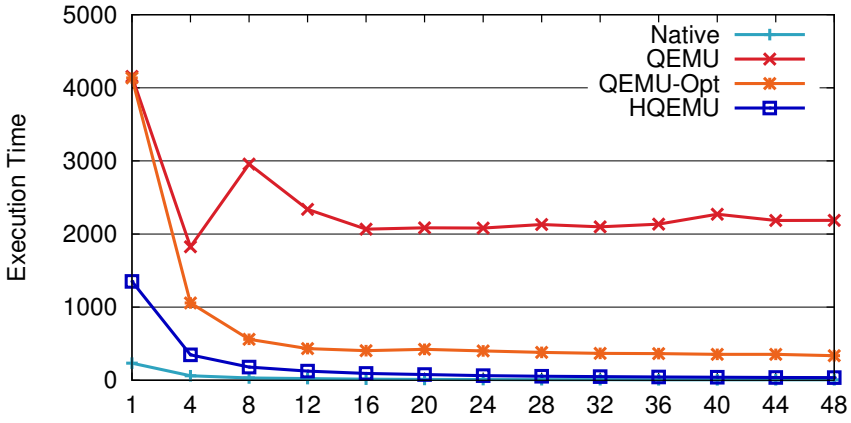
(f) raytrace



(g) streamcluster



(h) swaptions



(i) vips

Figure 5.4: PARSEC results of x86 to x86-64 emulation with 32 threads and native input sets. X-axis shows the number of threads, and the unit of time for Y-axis is in seconds.

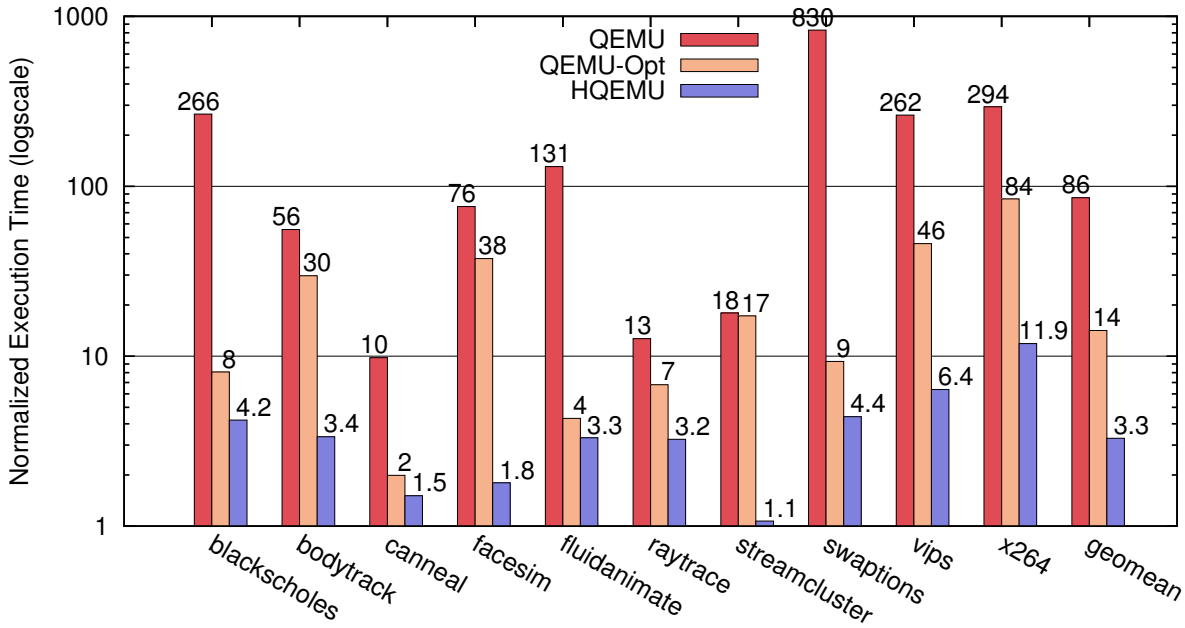


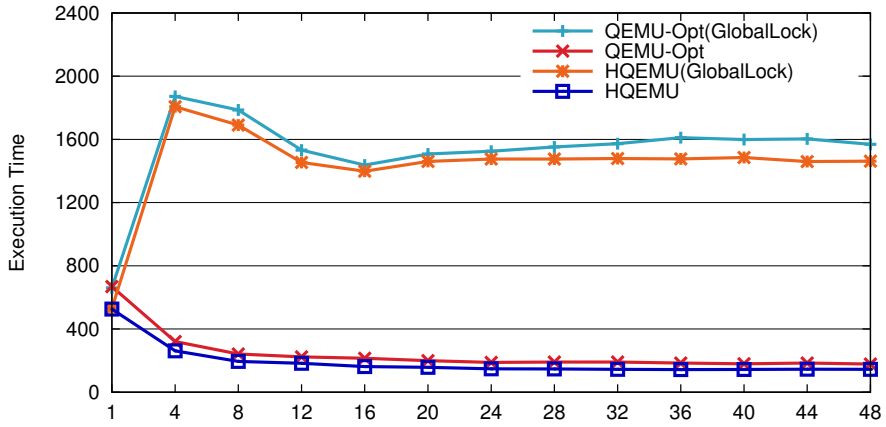
Figure 5.5: PARSEC results of x86 to x86-64 emulation with 32 threads and native input sets. Y-axis shows the normalized execution time over native run in logscale.

factor is only 3.3X on average over native execution with 32 emulated threads.

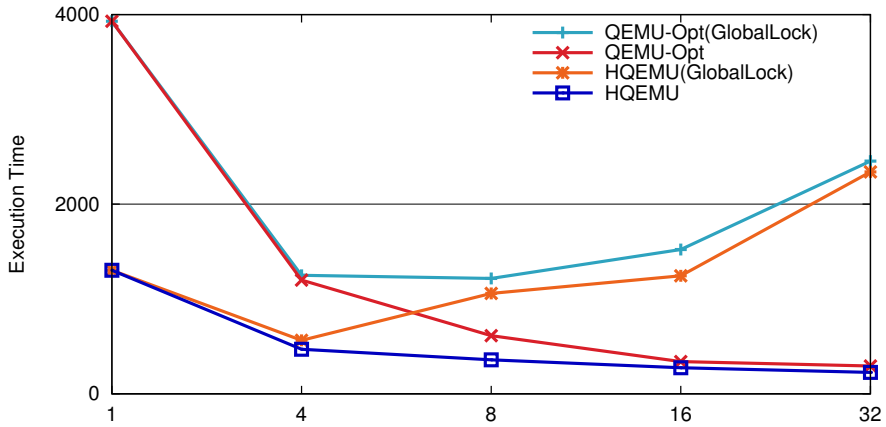
Performance of Lightweight Memory Transactions

In this experiment, we evaluate performance of atomic instruction emulation by comparing lightweight memory transactions with the global lock scheme. Because the global lock scheme could sometime cause deadlocks while running PARSEC benchmarks, we annotate the addresses that could cause data races and insert lock-unlock pair while translating the instructions that reference these addresses. Figure 5.6 only shows the results of two benchmarks, `canneal` and `fluidanimate`, because the performance of these two benchmarks are most sensitive to the global lock scheme. The comparison is conducted with QEMU-Opt and HQEMU.

As shown in Figure 5.6, the scalability is poor with the global lock scheme for both QEMU-Opt and HQEMU. The performance remains poor in `canneal` when emulating multiple threads, and the performance of `fluidanimate` even starts to degrade when the number of threads increases over 4 threads. In contrast, the performance improves linearly with the number of threads using lightweight memory transactions. On average, using lightweight memory transactions we can achieve 10X speedup over the global lock scheme with 32 threads for both `canneal` and `fluidanimate`. Benchmark `streamcluster` also shows slight improvement (2X with 32 threads) with lightweight memory transactions. No significant improvement is observed for the rest of benchmarks because they have fewer atomic instructions or fewer thread contentions for shared memory locations at runtime.



(a) canneal



(b) fluidanimate

Figure 5.6: Comparison of atomic instruction emulation with software memory transactions and global lock scheme. X-axis shows the number of threads, and the unit of time for Y-axis is in seconds.

PARSEC Results for x86 to ARM Emulation

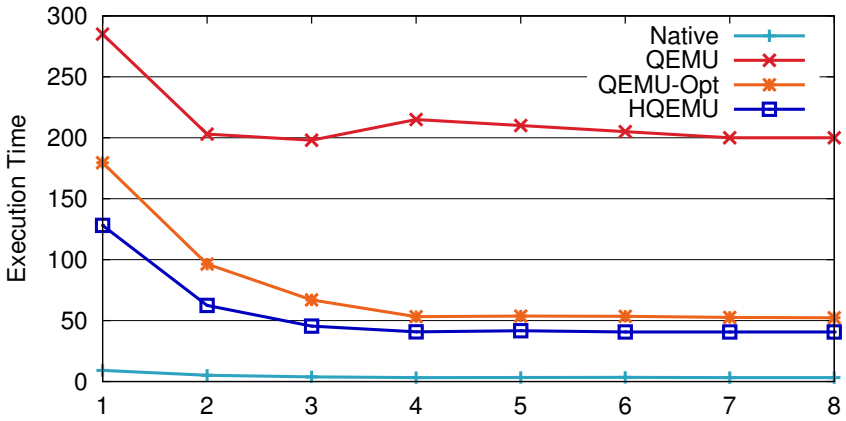
Figure 5.7 illustrates the results of three benchmarks for x86-32 to ARM emulation with simlarge input sets. As shown in Figure 5.7(a) and 5.7(b), the performance of QEMU does not scale well for benchmark `blacksholes` and `swpations`. The execution time starts to increase when the number of guest threads increases beyond 3. This is because larger number of threads will likely cause serialization of threads in the QEMU dispatcher. The performance of QEMU on the ARM platform does not degrade as significantly as that on the x86-64 platform (Figure 5.4(a) and 5.4(h)). This is because synchronization on the ARM platform's single chip multiprocessor (CMP) is much less expensive than that on the AMD Opteron machine whose 8 processors are based on the non-uniform memory architecture (NUMA). For QEMU-Opt and HQEMU, the performance results are similar to those on the x86-64 host.

Figure 5.7(c) shows the performance results of `canneal` compared with the global lock scheme. In Figure 5.7(c), the result of native run is not shown because some source code of `canneal` is written in assembly language which currently does not support the ARM architecture. Thus, the only way to run `canneal` on the ARM platform is through binary translation. As the figure shows, using lightweight memory transactions for atomic instruction emulation also achieves better performance than the global lock scheme, with about 26% improvement when emulating 4 execution threads with HQEMU.

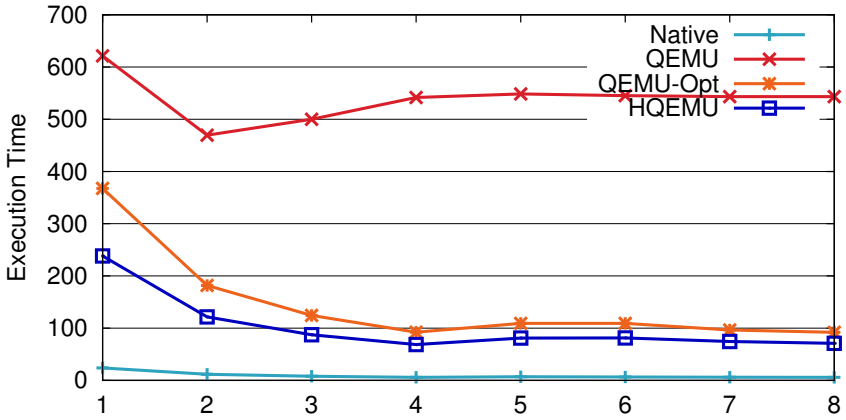
5.4 Summary

In this chapter, two optimization schemes, indirect branch translation caching (IBTC) and lightweight memory transactions, are introduced to reduce the contention on shared resources when emulating a large number of application threads. Our thread-private IBTC can keep the execution remaining inside the code cache, avoiding expensive context switch overhead as well as alleviating the severe contention during the indirect branch target translation.

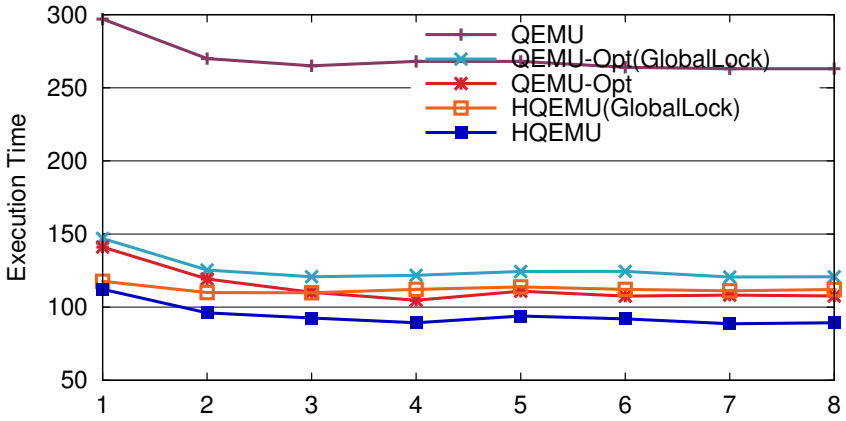
Based on the lightweight memory transaction scheme, the protection of memory accesses with a global lock can be safely removed. Moreover, the performance will not degrade much because the false protection of non-alias memory accesses and the overhead of expensive locking mechanism are eliminated as a result of the removal of global lock. We show that these two optimizations can significantly reduce the emulation overhead of a DBT and make it more scalable. The performance curve with HQEMU is very similar to that of native execution.



(a) blackscholes



(b) swaptions



(c) canneal

Figure 5.7: PARSEC results of x86-32 to ARM emulation with simlarge input sets. X-axis shows the number of threads, and the unit of time for Y-axis is in seconds.

Chapter 6

Dynamic Binary Translation of Client/Server Model

Considering the fast growing smart phone and tablet market, for example, many popular applications are either compiled for the ARM ISA or including libraries in ARM native code in the APK package (such libraries may be called via JNI). For vendors offering products in ISAs different from ARM, their customers may not be able to enjoy those applications. Using DBT to migrate such applications is one way to get around this dilemma. For example, DBT has been used to help application migration on workstations and PCs such as FX!32 [48] and IA-32 EL [9]. They have enabled IA-32 applications to be executed on Alpha and Itanium machines successfully in the past.

With rapid advances in mobile computing, multi-core processors and expanded memory resources are being made available in new mobile devices. This trend will allow a wider range of existing applications to be migrated to mobile devices, for example, running desktop applications in IA-32 (x86) binaries on ARM-based mobile devices. However, performance of the translated binaries on the mobile devices is very sensitive to the optimization overhead and the quality of the translated code. Such performance could significantly affect the energy consumption of the mobile devices because it is directly linked to the number of instructions executed and the overall execution time of the translated code. Hence, even though the capability of today's mobile devices will continue to grow, the concern over translation efficiency and energy consumption will put more constraints on a DBT for mobile devices, in particular, for *thin mobile clients* than that for severs.

With increasing network accessibility and bandwidth in various environments, e.g. wireless LAN or 4G networks in public areas, an increasing number of network servers are becoming accessible to thin clients. Those network servers are usually equipped with a substantial amount of resources. This opens up opportunities for DBT on thin clients to leverage much powerful servers. However, designing such a DBT for a client/server environment requires many critical considerations.

In this chapter, we look at those design issues for developing a distributed DBT system based on the client/server model. We propose a DBT system that consists of two dynamic binary

translators: an *aggressive dynamic binary translator/optimizer* on the server to service the translation/optimization requests from thin clients, and a *thin DBT* on each thin client that performs lightweight binary translation and basic emulation functions for its own. In the following sections, Section 6.1 first discusses the design challenges. Section 6.2 provides the organization of the two-translator client/server-based DBT system, and an optimization of asynchronous translation is presented in Section 6.2.3.

6.1 Design Issues

It is crucial for a distributed DBT system to divide system functionality between server and client so as to minimize the communication overhead. The *all-server* approach places the entire DBT system on the server side. The client sends the application binary to the server at first. The DBT system on the server then translates the guest binary into server ISA binary, runs the translated code directly on the server, and sends the results back to the client. However, the *all-server* DBT is not feasible for applications that need to access peripherals or resources (e.g. files) on the client. For example, an application running on the server will not be able to display a picture on the client because it can only access the screen of the server.

To ensure correct execution of all applications, we must run at least portions of the translated code on the client when needed. Although it is possible to divide the execution of the translated code between a client and the server dynamically by the client, system resources such as heaps and stacks, must be carefully monitored so that the division of work can be carried out correctly. It is desirable to have such a capability, but supporting that is quite challenging, and we leave this as an avenue for future work. In this thesis, we assume that the translated code is executed entirely on the client.

It is also crucial for the process of translation and code optimization in a distributed DBT system to tolerate network disruptions. To do so, we need the client to perform stand-alone translation and emulation when network connection or translation service on a remote server becomes unavailable. While in a normal operation mode, the DBT system should take advantage of the compute resources available on the server to perform more aggressive dynamic translation and optimization. Based on these considerations, we proposed a DBT system that consists of two dynamic binary translators: (1) a *thin DBT* that performs lightweight binary translation and basic emulation function on each thin client, and (2) an *aggressive dynamic binary translator/optimizer* on the server to service the translation/optimization requests from thin clients.

6.2 The Client/Server-Based DBT Framework

The organization of the proposed two-translator distributed DBT system and its major components are shown in Figure 6.1.

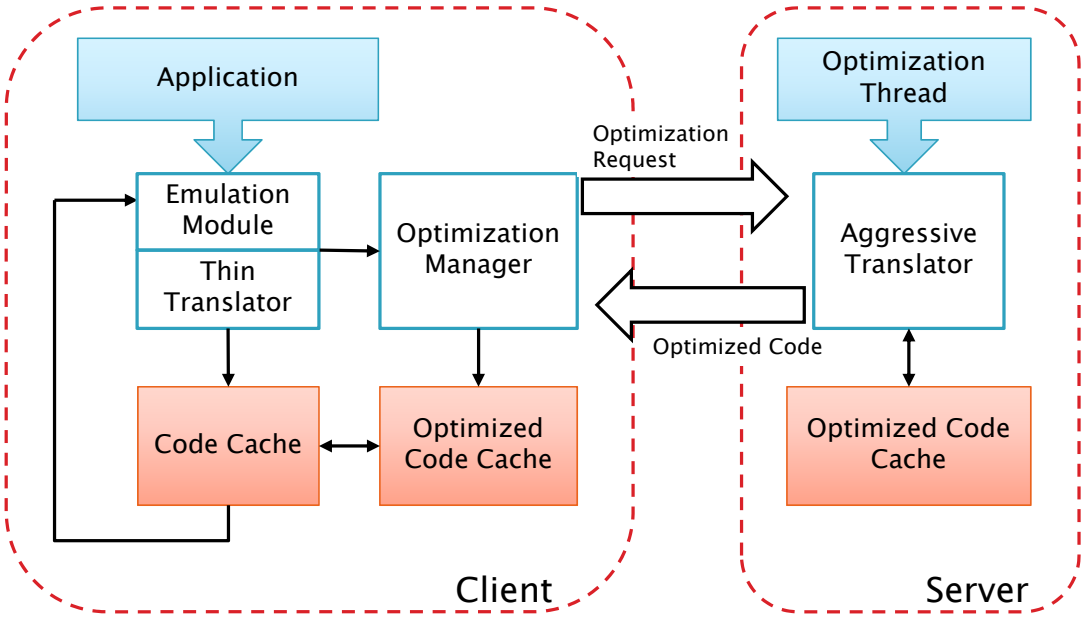


Figure 6.1: The architecture of the distributed DBT system.

6.2.1 Client

The design goal of the DBT on a thin client is to emit high-quality translated code while keeping the overhead low. The DBT on a thin client consists of all components available in a typical DBT system (the left module enclosed by the dotted line in Figure 6.1). In order to achieve low translation overhead, the thin client uses a *lightweight translator*. It contains only basic function to translate a guest binary to its host machine code. It does not have aggressive code optimizations nor analysis during translation. Although the performance of the generated host binary may be low due to non-optimized code generation, the thin client can be self-sufficient without the server if the network becomes unavailable.

When the lightweight DBT detects a section of code worthy of further optimization, it issues a request to the server for optimization service. Sending all code sections of a program to the server for optimization is impractical because it will result in substantial amount of communication overhead, and not all optimized code can achieve the desired performance gain. In particular, optimizing *cold* code regions might not be beneficial to the overall performance. Hence, we must ensure that the performance gain brought on by the optimization can amortize the communication and translation overhead on the server. Our strategy is to translate *all* code regions by the lightweight DBT on the thin client. Only the *hot* code regions that require repeated execution are sent to the server for further optimization. In this work, we first target *cyclic execution paths* (e.g. *loops* or *recursive functions*) as the optimization candidates since the optimized code is supposed to be highly-utilized. Frequently used procedures are also ideal candidates, however, recognizing procedure boundaries for an executable with debugging symbols stripped is challenging. Optimizations for hot procedures are currently under design and implementation. In this work, we

use the software-based region formation (i.e. extended NETPlus scheme) presented in Chapter 4 as our hot region detection approach. The reason to use the software-based region formation instead of the HPM-based trace merging is threefold. First, the thin client hardware may not have HPM support. Second, even if the clients allow HPM, most of the embedded software are short-running for which HPM sampling may not gain benefits. Finally, the creation of NET traces is necessitated before using HPM sampling. The trace separation of NET detection algorithm increases the number of sent optimization requests, and thus increases the network communication overhead.

As a hot code region is detected, the client does not attach any address information in the optimization request regarding where the optimized code should be placed. The reason is that, to provide such address information, the client needs to lock and reserve a sufficiently large memory region in the optimized code cache before sending the request to the server. There are three potential drawbacks in such an approach: (1) the client has no way of estimating how large the server-optimized code will be. Hence, it tends to over-commit the reserved space and cause substantial memory fragmentation in the optimized code cache; (2) a long wait time may be incurred by the locking; (3) the next optimization requests are blocked until the pending optimization is completed. Instead, our approach avoids such blocking, and the serialization only occurs when the optimized code needs to be copied into the optimized code cache.

Since the starting address where the optimized code will be placed is not known to the server at runtime, the server cannot perform relocation on the relative addresses (e.g. chaining code regions or trampoline) for the client. Hence, after the optimized code is received back from the server, the optimization manager is responsible for the relocation on relative addresses and committing the optimized code to the optimized code cache. The old code is patched with a branch to the optimized code so that the subsequent execution will be directed to the optimized, better-quality host binary. In current DBT prototype, the optimization manager is run on a dedicated thread. Unless there are large amount of code segments requiring optimizations, the optimization manager thread is usually idle and does not interfere with the execution thread.

6.2.2 Server

The more powerful server allows its translator to perform more CPU-intensive optimizations and analyses that often require a large amount of memory resources, and thus likely be infeasible on the resource-limited thin clients. For example, building and traversing a large CFG requires considerable computation and memory space. Furthermore, the server can act as a memory extension to the thin clients. The optimized code cache on the server can keep all translated and optimized codes for the thin clients throughout their emulation lifetime. The thin client can take advantage of this persistent code cache when it requests the same application that has been translated for another client previously. When the server receives an optimization request on a section of code that is already available in its optimized code cache, the server can send the optimized code back to the client immediately without re-translation/optimization. The response time can be significantly reduced.

As mentioned in Section 6.2.1, the starting address where the optimized code will be placed

is not known to the server, the server cannot perform relocation on the relative addresses for the client. To assist the client on relocation, the server attaches the patching rules to the optimized code and wrap them as a directive to the client so the client can patch the addresses accordingly.

6.2.3 Asynchronous Translation

The two translators in our distributed DBT system work independently and concurrently. When a thin client sends an optimization request to the server, its lightweight translator will continue its own work without having to wait for the result from the server. Such asynchrony is enabled by the optimization manager running on another thread on the client. The advantage of such an asynchronous translation model is threefold, (1) the network latency can be mitigated, (2) the translation/optimization overhead incurred by the aggressive translator is offloaded to the server, and (3) the thin client can continue the emulation process while the server performs further optimization on its request. With the asynchronous translation model, the DBT is able to complete the emulation even with network disruption or outage for translation/optimization services on a server.

6.2.4 Implementation Details

In our implementation, the distributed DBT prototype is developed based on HQEMU [47]. QEMU TCG is used as the lightweight translator on the client. TCG translates guest binary at the granularity of a *basic block*, and emits translated code to the code cache. The emulation module coordinates the translation and the execution of the guest program. It kick-starts TCG when an untranslated block is encountered. The purpose of the emulation module and the lightweight translator is to perform the translation as quickly as possible, so we could switch the execution to the translated code as early as possible. When the emulation module detects that some cyclic execution path has become *hot* and is worthy of optimization, a request is sent to the server for further optimization. For the more aggressive translator/optimizer on the server, we use an enhanced LLVM compiler because it consists of a rich set of aggressive optimizations and a JIT runtime system.

In this work, we use the two-level IR conversion scheme (presented in Section 3.5) to conduct the translation from the guest binary code to LLVM IR. One implementation of the two-level IR conversion based on the client/server model is to perform the first conversion from the guest binary to TCG IR on the client, and then transfer the TCG intermediate code to the server for the second conversion. Since the expansion rate with QEMU TCG is high (shown in Table 2.1), the size of the converted TCG IR is much greater than that of the guest binary—this approach will produce too much network communication overhead.

Hence, we use an alternative approach as shown in Figure 6.2. The guest code segments of the hot region are packed together and sent to the server directly. After that, the two-level IR conversion is performed completely on the server side. Assuming that the average size of one TCG IR instruction is the same as that of one guest instruction, and one guest instruction is roughly converted to six IR instructions as listed in Table 2.1, the network communication size

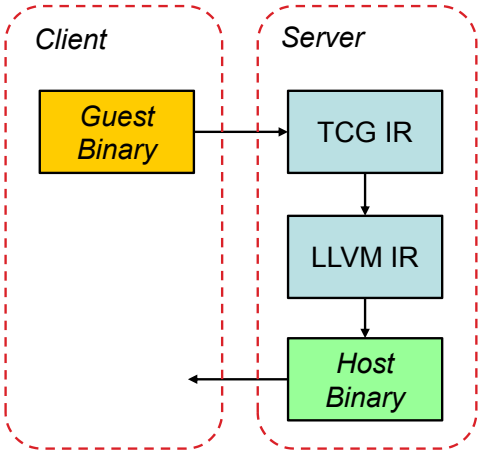


Figure 6.2: Two-level IR conversion in the distributed DBT system.

	Server	Client
Processor	Intel Core i7 3.3 GHz	ARMv7 1.0 GHz
# Cores	4	2
Memory	12 GBytes	1 GBytes
OS	Linux 2.6.30	Linux 2.6.39
Network	100Mbps Ethernet / WLAN 802.11g	
Extra optimization flags		
Native (ARM)	-O3 -ffast-math -mfpu=neon -mcpu=cortex-a8 -ftree-vectorize	
x86/32	-O3 -ffast-math -msse2 -mfpmath=sse -ftree-vectorize	

Table 6.1: Experiment configurations for client/server-based DBT.

is only one-sixth with the second approach compared to the first one.

6.3 Performance Results

In this section, we present the performance evaluation of the server/client-based DBT framework. We conduct the experiments with emulation of single-thread and multi-thread programs from short-running to long-running benchmarks. The comparison of asynchronous and synchronous translation and the impact of persistent code cache are evaluated. Detailed analysis of the overall performance is also provided to verify the effectiveness of the proposed scheme.

Experimental Setup

All performance evaluation is conducted using an Intel quad-core machine as the server, and an ARM PandaBoard embedded platform [76] as the thin client. The detailed hardware configuration is listed in Table 6.1. The evaluation is conducted with the x86/32-to-ARM emulation. In

our client/server DBT system, The x86-32 guest binary is fed to the DBT running on the ARM board. The x86-64 server receives optimization requests from the client and sends the translated ARM binary back which are then executed on the ARM processors. We use TCG of QEMU version 0.13 as the thin translator on the client side and use the JIT runtime system of LLVM version 2.8 as the aggressive translator/optimizer on the server side. The default optimization level (-O2) is used for the LLVM JIT compilation. The network communication between the client and server is through TCP/IP protocol.

We use MiBench [39] with large input sets, and SPEC CPU2006 integer benchmark suite with test and reference input sets as the short-, medium- and long-running test cases for our performance studies. A subset of the SPLASH-2 [88] benchmarks are also used for studying the emulation of multi-thread guest applications. Although SPEC and SPLASH-2 benchmarks are not widely applied embedded benchmarking sets, we chose to present the results of them for the purpose of cross-comparison because they are the standard benchmark suites which have been widely used to evaluate the DBT systems.

All benchmarks are compiled with GCC 4.4.2 for the emulated x86-32 executables. Except for the default optimization flags in each benchmark suite, we added extra optimization flags, which are listed in Table 6.1, when compiling the benchmarks. We compare the results to the native runs whose executables are compiled by GCC 4.5.2 on the ARM host also with the extra optimization flags in Table 6.1.

We compare the following three configurations:

- **QEMU**, which is the vanilla QEMU version 0.13 with the TCG thin translator.
- **Client-Only**, which is the multithreaded HQEMU presented in Chapter 3. It uses the software-based region formation, and the DBO is disabled.
- **Client/Server**, which is the distributed DBT framework proposed in this chapter.

The QEMU configuration is used to demonstrate the performance of the thin translator itself, and also as the baseline performance when the network is unavailable. The evaluation in the following experiments is measured with unlimited code cache size and optimized code cache size on the thin client, and based on the 100Mbps Ethernet unless changes of the cache size or network infrastructure are mentioned.

Emulation of Short-Running Applications: MiBench

The single-thread MiBench is used as the small and short-running benchmark suite. Figure 6.3 illustrates the overall performance results for MiBench with large input sets. The Y-axis is the speedup over QEMU. As the figure shows, the performance of several benchmarks for Client-Only is slightly slower than QEMU, such as `tiff2rgba`, `tiffmedian` and `sha`, ... etc. For benchmark `jpeg`, `stringsearch` and `pgp`, the performance of Client-Only is even worse than QEMU. The reasons for such poor performance are: (1) The execution thread continues its execution while the aggressive optimizer running on another thread takes too much time to complete. Thus, the optimized code may miss the best timing to be utilized. (2) the instrumentation-based region detection approach incurs considerable overhead. (3) The interference by the optimization thread with the execution threads also incurs some penalty. The performance on the em-

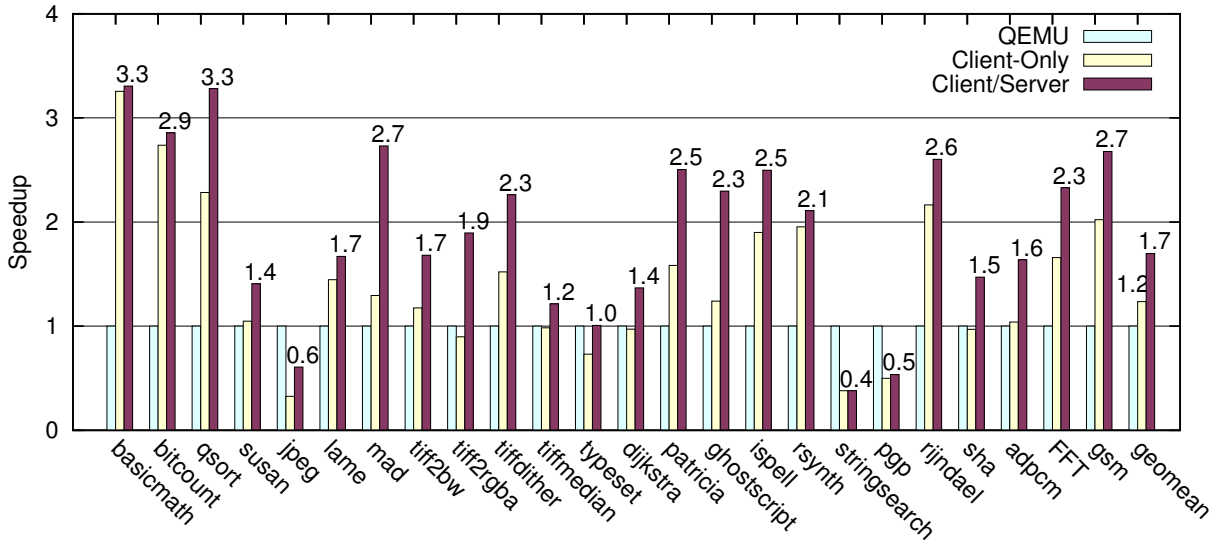


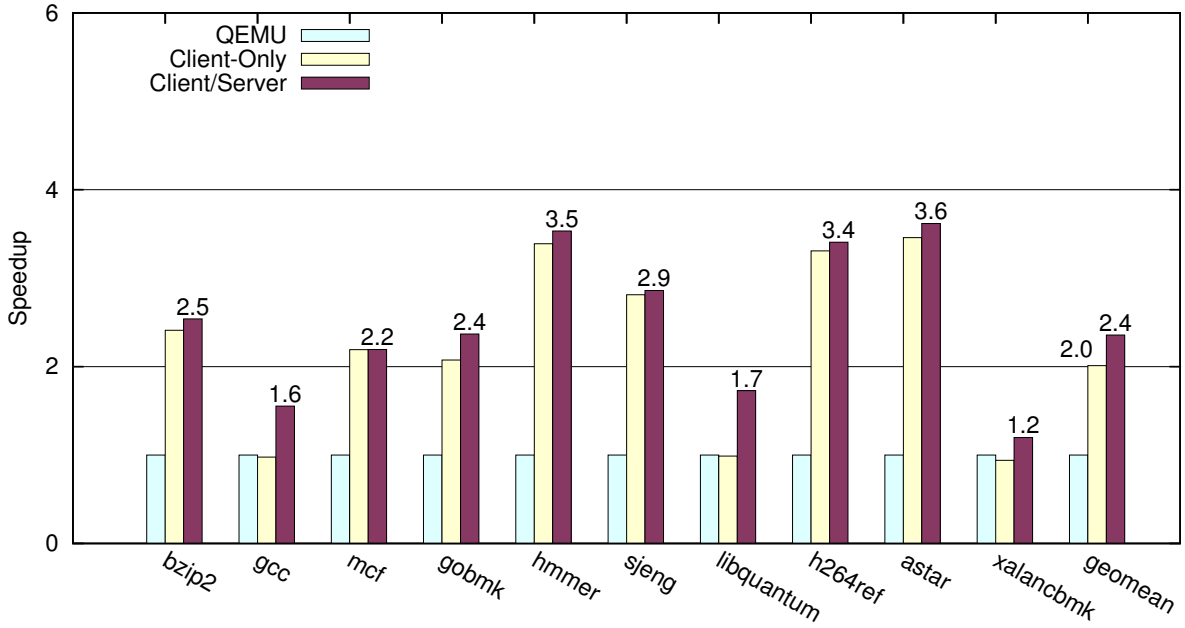
Figure 6.3: MiBench results of x86-32 to ARM emulation with large data sets. Code cache size: Unlimited.

bedded platform is very sensitive to these three factors especially for such short-running benchmarks. We can observe from the three most short-running programs of all benchmarks, `jpeg`, `stringsearch` and `pgp`, their performance is poor with the Client-Only configuration. In contrast, the performance of the other benchmarks is significantly improved compared with QEMU because of the benefits of the optimized code from the aggressive translator. On average, Client-Only achieves 1.2X speedup over QEMU.

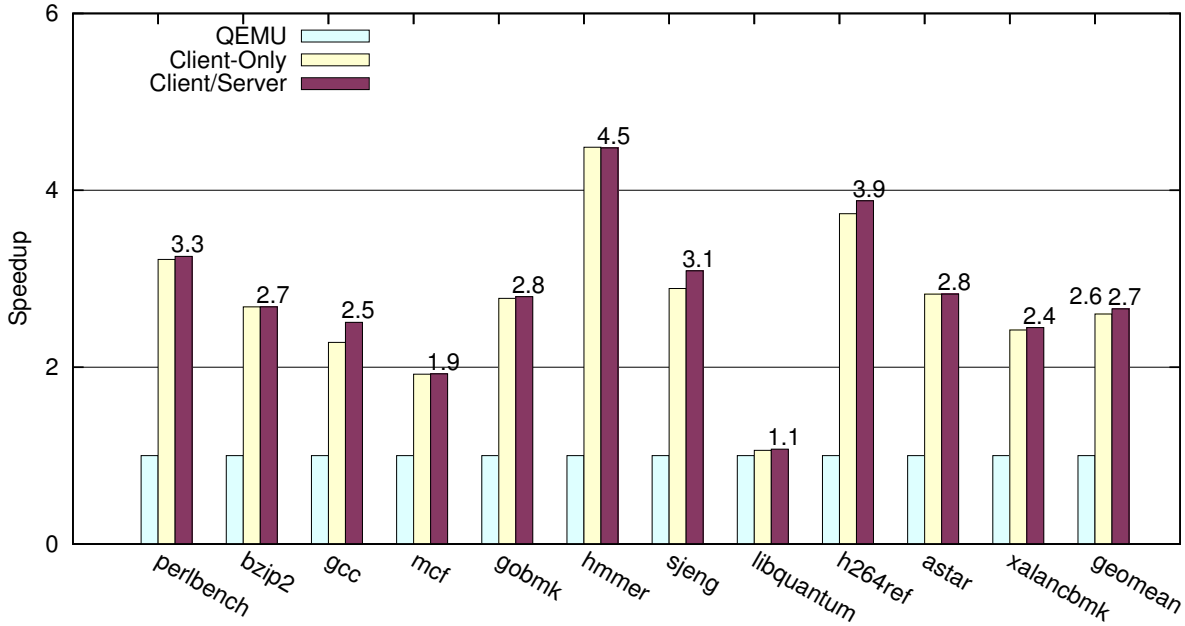
As for the Client/Server mode, all benchmarks outperform their Client-Only counterparts. This is because the translation/optimization of code region is much more efficient by the powerful server than by the ARM processors. Thus, the thin client can enjoy the optimized code earlier while the same optimization request may still be queued in the optimization request queue for the Client-Only mode. In addition, the execution thread incurs less interference by the optimization. Therefore, the benefit from optimized code can amortize the penalty from the hot code region detection and results in significant performance improvement over both Client-Only and QEMU. The performance of the benchmark, `jpeg`, `stringsearch` and `pgp`, are still slower than QEMU because the instrumentation overhead are too high to be amortized for these three very short-running programs. Our DBT framework, on average, is about 1.7X faster than QEMU and achieves 37% performance improvement over Client-Only with the client/server model for MiBench benchmarks.

Emulation of Medium-Running and Long-Running Applications: SPEC CINT2006 Benchmarks

The emulation of medium-running and long-running programs are configured by running SPEC CINT2006 benchmarks with test and reference input sets, and their results are shown in Figure 6.4 (a) and 6.4(b), respectively. The performance of `perlbench` and `omnetpp` in Figure



(a) CINT (Test input)



(b) CINT (Ref input)

Figure 6.4: CINT2006 results of x86-32 to ARM emulation with test and reference inputs. Code cache size: Unlimited.

6.4(a), and `omnetpp` in Figure 6.4(b) are not listed because both QEMU and our framework failed to emulate these two benchmarks. Figure 6.4 presents the speedup of Client-Only and Client/Server over QEMU. As Figure 6.4(a) shows, three benchmarks, `gcc`, `libquantum` and `xalancbmk`, have slight performance degradation with Client-Only over QEMU. The execution thread of them also miss the best timing to utilize the optimized code, especially for `gcc` which generates considerable requests for code region optimization.

As for the Client/Server mode, the performance of `gcc`, `gobmk`, `libquantum` and `xalancbmk`, has the most significant improvement over the Client-Only mode, while other benchmarks have only slight improvement. For `gcc` and `gobmk`, these two benchmarks have much more code regions to be optimized. Therefore, it is beneficial to process such heavy optimization load on the powerful server instead of on the thin client. Client/Server achieves about 60% and 15% performance improvement over Client-Only for `gcc` and `gobmk`, respectively. For `libquantum` and `xalancbmk`, the completion time of the translated regions on the server is earlier than that processed by the client. Client/Server can thus produce the optimized code earlier in time to benefit the overall execution time, which may not be possible with the Client-Only mode. The results show that Client/Server can further improve `libquantum` and `xalancbmk` whereas Client-Only causes performance degradation compared with QEMU. The performance of Client/Server is about 2.4X faster than QEMU on average, and achieves 17% improvement over Client-Only.

In Figure 6.4(b), the results show that both Client-Only and Client/Server mode outperform QEMU for all benchmarks with reference inputs. The reason is that the optimized code is heavily used with reference inputs. It makes the translation time to account for a smaller percentage of the total execution time. Once the highly optimized codes are emitted to the optimized code cache, the thin client can immediately benefit from the better code compared to the less optimized code executed by QEMU. Since the Client/Server mode tends to eliminate the optimization overhead which is insignificant to the total execution with reference inputs, no significant performance gain is observed with Client/Server over Client-Only in this case. The only exception is `gcc` where about 624 seconds are reduced (10% improvement) with Client/Server. On average, both Client-Only and Client/Server outperform QEMU with a speedup of 2.6X and 2.7X, respectively.

The performance of our client/server-based DBT system is only 3X slower than the native execution with test inputs on average, compared to 7.1X slowdown with QEMU. As for the reference inputs, the average slowdown over native run is only 1.9X, compared to 5.1X slowdown with QEMU. Note that we have only applied the traditional optimizations in LLVM for this experiment, however, with more aggressive optimizations for performance and power consumption, the benefit of the proposed Client/Server mode could be greater.

Analysis of Emulation of Single-Thread Programs

Figure 6.5 shows the execution-time breakdown in the Client-Only and Client/Server mode according to the emulation components: *Region* (time spent in the optimized code cache), *Block* (time spent in the code cache) and *Dispatch* (time spent in the emulation module) for the MiBench and SPEC CINT2006 benchmarks. We use hardware performance counters and Linux Perf Event toolkit [62] with event `cpu-cycles` to estimate the time spent in these three components. As the

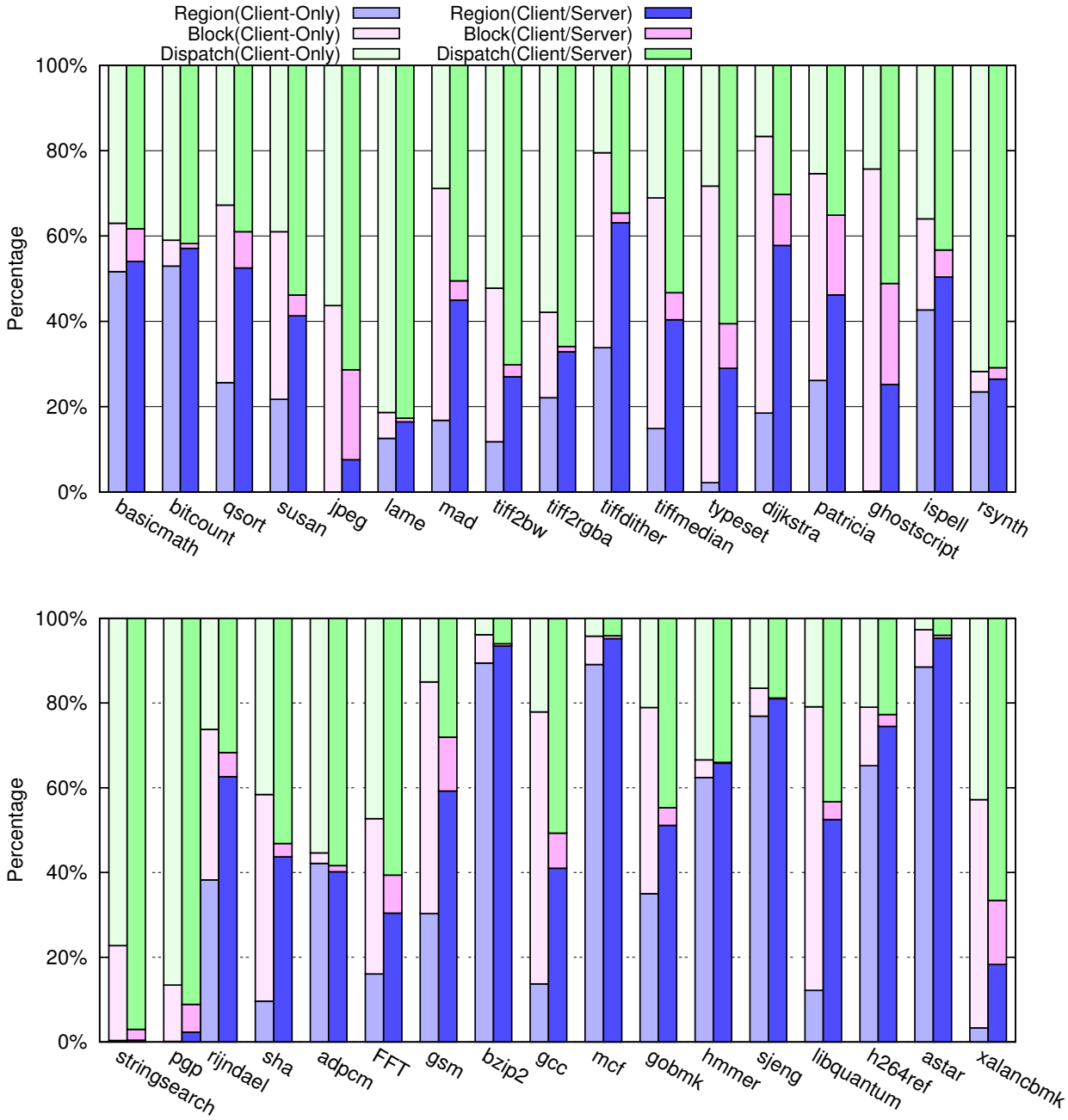


Figure 6.5: Breakdown of time of x86-32 to ARM emulation for MiBench with large inputs and CINT2006 with test inputs comparing Client-Only and Client/Server mode.

	QEMU	LLVM	Client-Only		Client/Server		
Benchmark	Expans-ion Rate	Expans-ion Rate	#Region	ICount (10^9)	#Region	ICount (10^9)	Reduc. Rate
MiBench (large input sets)							
basicmath	8.0	4.6	195	15.0	196	7.0	54 %
bitcount	8.0	3.3	22	2.2	23	1.9	16 %
qsort	8.1	4.9	59	2.7	73	1.2	56 %
susan	7.9	3.2	45	2.9	117	1.7	43 %
jpeg	7.8	4.5	71	2.5	324	1.6	36 %
lame	7.4	6.0	678	33.2	707	23.1	30 %
mad	8.2	4.1	211	5.4	233	1.4	73 %
tiff2bw	8.0	4.7	68	1.9	65	.79	59 %
typeset	7.5	4.1	167	7.2	1601	2.4	67 %
dijkstra	8.2	4.9	38	1.8	65	.63	65 %
patricia	8.1	4.5	213	5.5	220	2.0	62 %
ghostscript	7.5	4.5	456	13.9	1272	4.3	69 %
ispell	8.1	4.9	374	7.0	365	3.3	53 %
rsynth	8.0	5.1	242	22.1	243	9.3	58 %
stringsearch	8.0	2.5	5	.31	24	.27	12 %
pgp	7.9	3.6	43	1.8	393	1.2	34 %
rijndael	7.6	3.0	55	3.1	52	1.5	52 %
sha	7.7	3.1	43	1.6	48	.52	67 %
Average	7.9	4.1					46 %
CINT2006 (test input sets)							
bzip2	8.0	3.4	924	71	928	56	21 %
gcc	8.5	5.1	4678	58	16868	24	58 %
mcf	7.9	4.2	275	16	301	11	29 %
gobmk	8.7	4.9	14106	258	19993	153	41 %
hmmer	8.1	4.5	237	56	241	51	9 %
sjeng	8.3	4.8	988	54	1137	36	33 %
libquantum	7.8	4.1	112	3.5	163	1.1	68 %
h264ref	7.5	2.9	1535	210	1503	176	16 %
astar	7.7	4.0	518	56	551	40	29 %
xalancbmk	7.4	4.4	551	11	2676	4.2	63 %
Average	8.0	4.2					31 %

Table 6.2: Measures of x86-32 to ARM emulation for MiBench with large input sets and SPEC CINT2006 benchmarks with test input sets. *Expansion Rate* represents the average number of ARM instructions translated per x86-32 instruction by QEMU TCG and LLVM translator; *#Trace* represents the total amount of traces generated by the optimization thread (Client-Only) and optimization service (Client/Server) during the execution period; *ICount* represents the number of retired instructions on the client platform; *Reduction Rate* represents the ratio of reduced instruction counts.

result shows, the percentage of time spent in the optimized code cache increases and the time spent in the code cache reduced significantly with Client/Server over Client-Only mode for all benchmarks. That is, the execution thread of Client/Server spends more time running in the optimized code regions. Column 4 and 6 in Table 6.2 lists the number of regions generated with Client-Only and Client/Server modes during the whole emulation time period, respectively. The results show that much more regions are generated in the Client/Server mode. This is because the powerful server can perform more efficient LLVM JIT compilation than the ARM thin client. Take benchmark `gcc` as an example, only 4678 regions are optimized by Client-Only, but 16868 regions can be optimized by the powerful server even with less emulation time. That means there are many optimization requests pending in the optimization request queue with Client-Only. The results of the number of regions generated for Client-Only and Client/Server mode also imply that the optimized code can catch the best utilization timing for the execution thread earlier with Client/Server than Client-Only mode because its translation time for each code region is shorter. These results also match the results from the breakdown of time in Figure 6.5 that the execution thread spends more percentage of time within the optimized code with Client/Server mode.

We also use hardware performance counters with event `instructions` to count the total number of instructions executed on the thin client for MiBench and SPEC CINT2006 benchmarks. The results for Client-Only and Client/Server are listed in column 5 and 7 in Table 6.2, respectively. For Client-Only, the instruction count includes the instructions for emulating guest programs and for the aggressive optimization. Since the aggressive optimization is conducted by the server with Client/Server, only the instructions for emulation of guest programs are counted in this mode. The instruction reduction rate of Client/Server is presented in the last column, where on average about 46% and 31% of the total instructions can be reduced for MiBench and SPEC CINT2006 benchmark, respectively. This reduction of instructions results mainly from off-loading the aggressive translation/optimization to the server and also from earlier execution of better quality code from the server.

Column 2 and 3 in Table 6.2 compares the quality of the translated host code generated by QEMU TCG fast translator and by our DBT system respectively in terms of the expansion rate. The expansion rate is measured as the number of host instructions translated per guest instruction (i.e. the total number of ARM instructions generated divided by that of the x86-32 instructions). As shown in the results, QEMU translates, on average, one guest binary instruction to 8 host instructions. With the LLVM aggressive translator/optimizer, we can reduce that number to 4, and thus achieve the goal of generating higher quality code.

Emulation of the Multi-Thread Applications: SPLASH-2

Figure 6.6 illustrates the performance results of three configurations for SPLASH-2 benchmarks running one and two emulation thread(s). The Y-axis is the speedup over QEMU with one emulation thread. For QEMU, the performance scales well when increasing the number of threads to two for most benchmarks. However, the performance scalability of Client-Only is poor and even degrades for benchmarks `Cholesky` with two emulation threads, whose scalability ranges from -12%(`Cholesky`) to 24%(`FFT`). For benchmark such as `Barnes`, `Cholesky`,

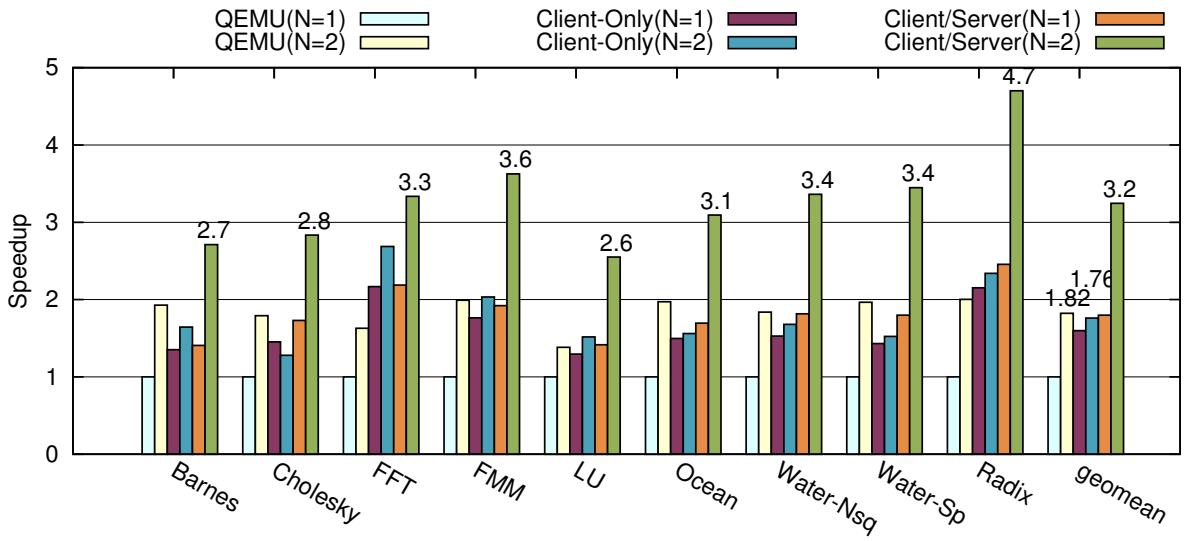


Figure 6.6: SPLASH-2 results of x86-32 to ARM emulation with large data sets. *N* represents the number of emulation thread. Code cache size: Unlimited.

`Ocean`, `Water-Nsq` and `Water-Sp`, the performance results of them are slower than those of QEMU. The poor scalability is due to two reasons: (1) the emulation threads miss the timing to utilize the optimized code, and (2) the aggressive optimization thread would compete the processor resources and interfere with the execution of the emulation threads because the testing ARM platform has only two cores (i.e. three threads are running at the same time). On average, QEMU achieves 1.82X speedup with two emulation threads, where only 10% improvement is achieved with Client-Only.

The performance of Client/Server also scales well with two emulation threads. Thanks to the powerful server for conducting the aggressive optimization, the emulation threads can fully utilize the processor resources without incurring any interference. The performance of the Client/Server mode is 1.78X and 1.84X faster than QEMU and Client-Only, respectively, when two guest threads are emulated. From the study of this result, emulating single-thread application on a single-core embedded platform can also achieve low overhead and high performance with our client/server DBT framework.

Comparison of Asynchronous and Synchronous Translation

To evaluate the effectiveness of asynchronous translation, we compare it with synchronous translation. Figure 6.7 illustrates the performance gain of the asynchronous translation mode compared to the synchronous translation mode for a subset of MiBench, SPEC CINT2006 and SPLASH-2 benchmarks. As opposed to that in the asynchronous translation, when the thin client in the synchronous mode sends an optimization request to the remote server, it will suspend and resume its execution until the optimized code is sent back from the server and emitted to the optimized code cache.

In Figure 6.7, the result shows that the many benchmarks of MiBench are sensitive to syn-

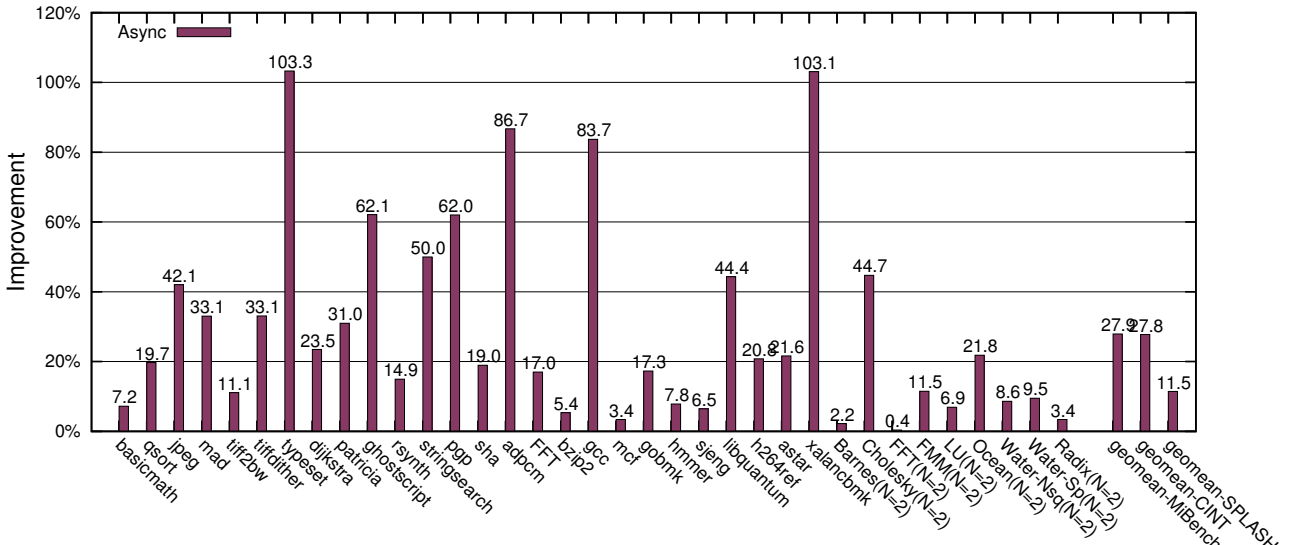


Figure 6.7: Comparison of asynchronous and synchronous translation. Code cache size: Unlimited. (*baseline = Synchronous translation*)

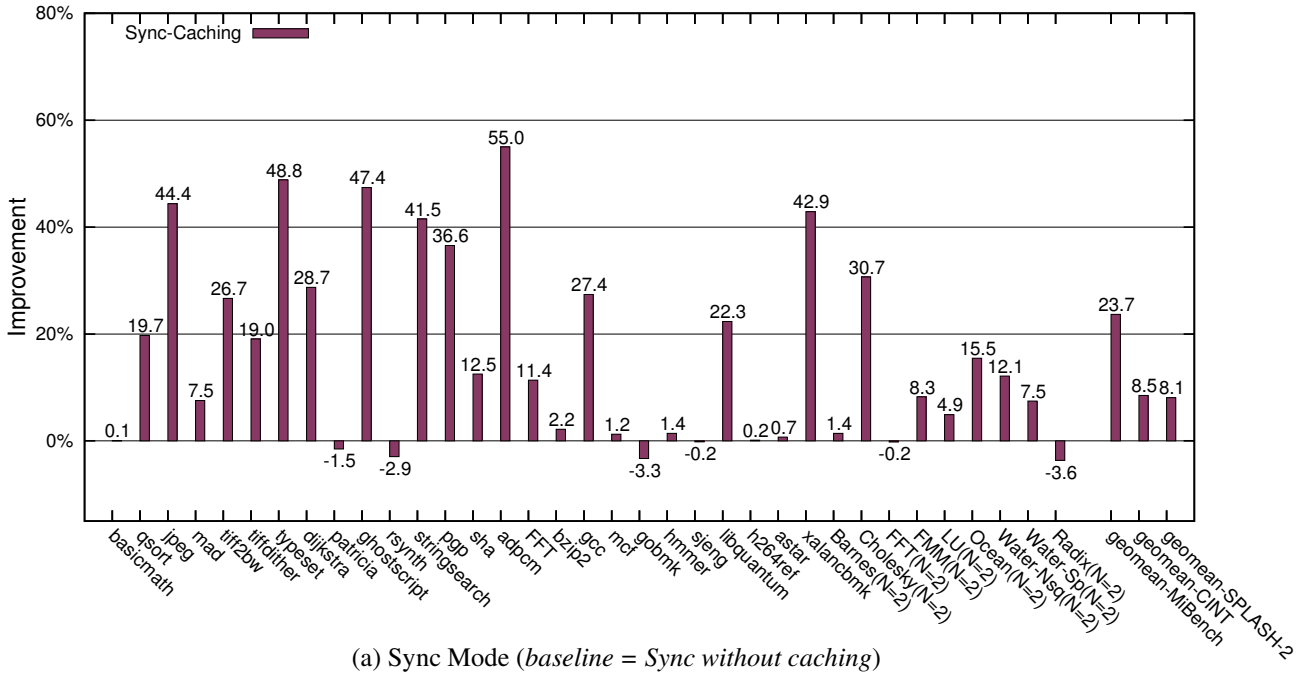
chronous translation. Even though the optimization conducted by the powerful server is efficient, the waiting time is still not tolerable for such short-running programs. For several benchmarks in MiBench, the performance with asynchronous translation can reach at least 50% improvement. The benchmarks, `libquantuam` and `xalaxncbmk` in SPEC CINT2006 and `Cholesky` in SPLASH-2, also have similar behavior because they are all short-running programs. As for `gcc`, which has a lot of code regions for optimization, asynchronous translation also shows significant performance gain (about 84%) compared to synchronous translation. The performance of the asynchronous translation achieves the geomean of 27.2%, 27.8% and 11.5% improvement over synchronous translation for MiBench, SPEC CINT2006 and SPLASH-2 benchmark suite, respectively.

Impact of Persistent Code Cache

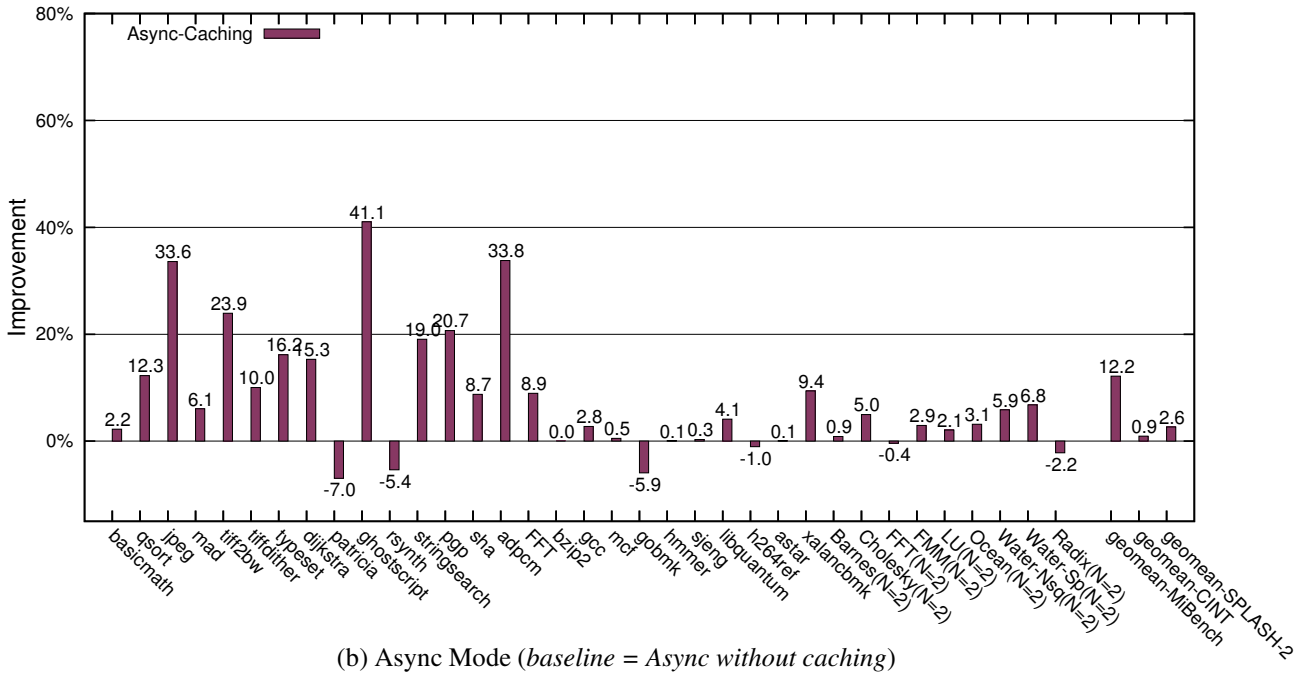
In this section, we evaluate two scenarios. First, one client emulates a program and another client emulates the same program a while later. Second, we force the client to flush its local code cache after the code cache is full. These two cases are used to evaluate the impact of the server's persistent code cache if the translated code can be re-used.

Continuous Emulation of the Same Program

Figure 6.8 presents the performance results of the first case. We also evaluate this case using both asynchronous and synchronous translation modes whose results are shown in Figure 6.8(a) and 6.8(b). As the first client starts the emulation, no pre-translated/optimized code is cached, and the server needs to perform translation/optimization for the first client. The result of this first client is set as the baseline performance. When the second client requests the same



(a) Sync Mode (*baseline = Sync without caching*)



(b) Async Mode (*baseline = Async without caching*)

Figure 6.8: Impact of persistent code cache with continuous emulation of the same program.

Benchmark	CodeSize (KBytes)	#Flush	Benchmark	CodeSize (KBytes)	#Flush
CINT2006			SPLASH-2		
bzip2	1050	12	Barnes	918	2
gcc	14984	59	Cholesky	1671	22
mcf	783	2	FFT	735	2
gobmk	5637	30	FMM	1227	6
hmmer	1155	2	LU	751	4
sjeng	1246	7	Ocean	1314	5
libquantum	724	2	Water-Nsq	1076	2
h264ref	2978	49	Water-Sp	1098	2
astar	1242	2	Radix	749	2
xalancbmk	7253	2			

Table 6.3: Measures of code cache flushing.

program that has been translated/optimized by the first client, the optimized code is reused and the improvement is illustrated in Figure 6.8. As the results show, the performance is improved with the server-side caching because the response time is shortened for all optimization requests. The impact of caching is more significant for the short-running benchmarks such as MiBench and `libquantum` and `xalancbmk`.

The impact of caching is even more noticeable with the synchronous translation mode (Figure 6.8(a)) because the waiting time for translation/optimization can be reduced with the persistent code already cached on the server. Many benchmarks of MiBench are improved by at least 30%, and benchmark `gcc`, `libquantum` and `xalancbmk`, are improved by about 27%, 22% and 43% in the synchronous mode, respectively, where an average of 24%, 8.5% and 8.1% of performance gain is achieved for MiBench, SPEC CINT2006 and SPLASH-2.

As for the asynchronous mode, the client does not wait for the server-side optimization. It also continues the execution with the help of its lightweight translator. Therefore, the benefit from persistent code caching is not significant for the client. An average of 12.2% improvement is achieved for MiBench and only 0.9% and 2.6% of improvement on average is observed for SPEC CINT2006 and SPLASH-2 benchmarks.

Several benchmarks, such as `gobmk`, show a negative impact from persistent code caching. The reason is that there are 7 different input data sets for `gobmk`. When the emulation is done with the first input data set, the server keeps some optimized code in its persistent code cache. As the emulation continues with the rest of the input sets, the server sends the cached code back to the client. However, the cached code that is good for the previous input data sets may not be suitable for the current input data. It causes many early exits from the traces [45], and results in some performance loss.

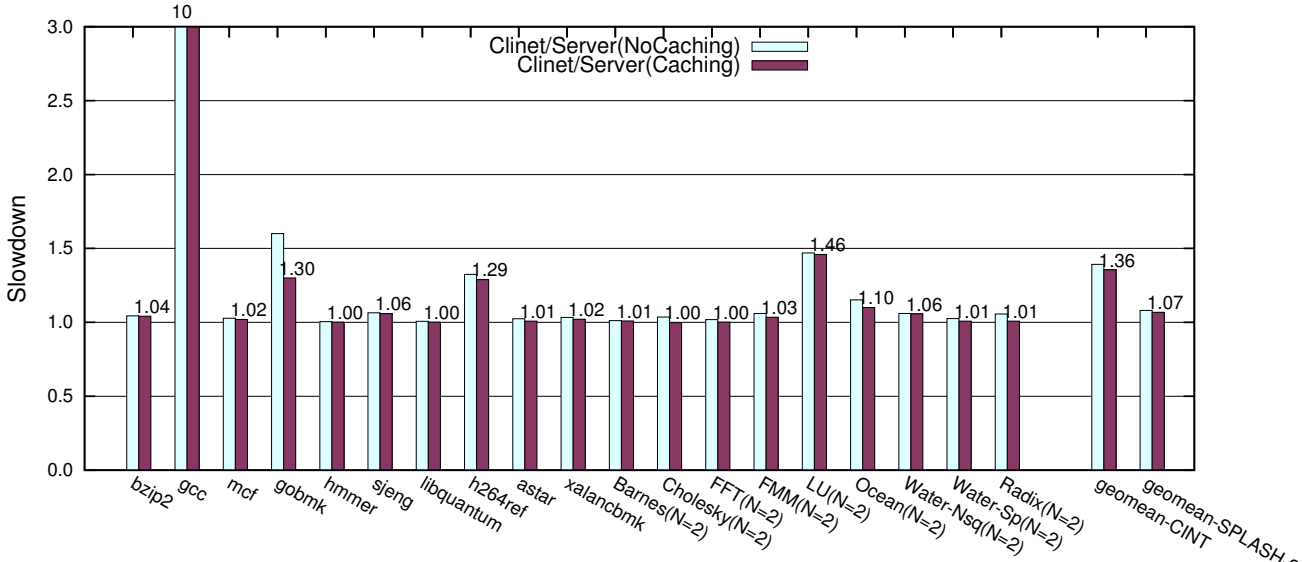


Figure 6.9: Impact of persistent code cache with code cache size as half translated code size. (*baseline = Client/Server with unlimited cache size*)

Code Cache Flushing

In the second case, the code cache size on the client is set to be smaller than the size needed for the emulated benchmarks. When the code cache is full, the client is forced to flush both the code cache and the optimized code cache. In this experiment, we use the single-thread SPEC CINT2006 and the multi-thread SPLASH-2 benchmarks for our performance studies. We set the code cache size to half of the total translated code size. For the testing benchmarks, the lightweight translator in our DBT system always fills up the code cache earlier than the optimized code cache. Thus, as the code cache is full, both code cache and the optimized code cache are flushed at the same time. The total size of the translated host code from the thin translator and the number of code cache flushes is listed in Table 6.3.

We compare the performance of Client/Server with and without server-side persistent code caching. Figure 6.9 shows the slowdown factor compared with Client/Server mode with unlimited code cache size (i.e. no flushing is required).

From column 3 and 6 in Table 6.3, most benchmarks, such as `mcf`, `hmmer` etc. in SPEC CINT2006 and `Barnes` etc. in SPLASH-2, only incur a few flushes and not much performance loss is observed in Figure 6.9 for these benchmarks with or without server-side caching. As for `gobmk` and `h264ref`, which have more code cache flushing frequency, performance degradation for these two benchmarks are observed. Without server-side caching, although it can still get help from the powerful server to re-translate the code, about 60% and 33% performance are lost compared with Client/Server mode of unlimited code cache size, respectively. With server-side caching, the slowdown of these two benchmarks reduces to 30% and 28%, respectively. As for `gcc`, its code cache size is too small to accommodate its working sets, so frequent code cache flushes would occur. The optimized code is not effectively utilized between two flushes, and sending frequent optimization requests to the server incurs a lot of overhead on the client even if

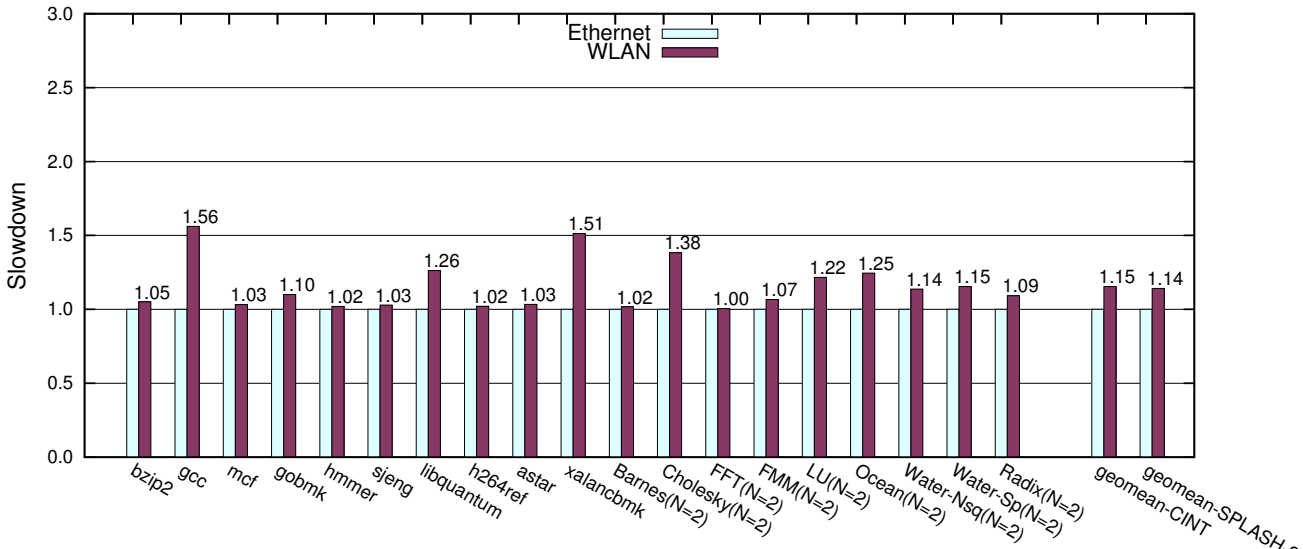


Figure 6.10: Comparison of x86-32 to ARM emulation for SPEC CINT2006 and SPLASH-2 benchmarks using Ethernet and WLAN. Code cache size: Unlimited.

the server has cached the optimized code. Thus, about 10X slowdown is observed.

Comparison of Different Network Infrastructures

Figure 6.10 presents the performance comparison of Client/Server mode with asynchronous translation using Ethernet and WLAN of 802.11g for SPEC CINT2006 and SPLASH-2 benchmarks. For the WLAN settings, the thin client connects to a wireless access point which connects to the server through the local area network. For SPEC CINT2006 benchmarks, the longer latency of WLAN only cause slight performance degradation over Ethernet, such as `bzip2`, `mcf` and `hmmer`, etc. `libquantum` and `xalancbmk` incur more performance loss because they are relatively short-running programs which are more sensitive to the network latency. `gcc` incurs about 56% performance degradation. This is because it has a large amount of code regions for optimization and the longer latency of WLAN causes the execution thread to miss the best timing to utilize the optimized code. The results also show that the benchmarks of SPLASH-2 running two emulation threads are also sensitive to the communication latency of WLAN. On average, the performance degrades about 15% and 14% with WLAN over Ethernet for SPEC CINT2006 and SPLASH-2 benchmark suite, respectively.

6.4 Summary

The two-translator approach using hybrid QEMU (client) + LLVM (server) is an effective design for building an efficient client/server-based DBT system. With the asynchronous translation scheme, the DBT can mitigate the translation/optimization overhead and network latency. It can also tolerate network disruption and outage for translation/optimization services on a server.

Experimental results show that the DBT of the client/server model can achieve 37% and 17% improvement over that of non-client/server model for x86/32-to-ARM emulation using MiBench and SPEC CINT2006 benchmarks with test inputs, respectively, and 84% improvement using SPLASH-2 benchmarks running two emulation threads on an ARMv7 dual-core platform; it is about 1.7X, 2.4X and 1.7X speedup over QEMU for these three benchmark suites, respectively.

Chapter 7

Related Work

In this chapter, we will review several dynamic binary translation systems and just-in-time compilation frameworks that are related to our work. We will first introduce the systems using template-based runtime code generator. Then, the DBT systems built with multithreaded strategies and virtual machines for embedded systems are provided. Finally, the trace selection algorithms, indirect branch optimization, and the HPM-based work that motivate this dissertation are discussed.

7.1 Dynamic Optimization Systems

7.1.1 Template-Based Compilation Systems

Cyclone [49] is a type-safe language (a dialect of C) developed by the groups from AT&T Lab and Cornell. The Cyclone compiler uses a template-based strategy for runtime code generation. At run time, pre-compiled code fragments are stitched together to create a new function, and the compiler does not perform optimizations such as register allocation, code motion, or common subexpression elimination on the templates. Hence, its strategy can keep the cost of code generation low but the resulting performance is poor. To overcome the flaw of the template-based Cyclone compiler, Smith et al. [81] tried to develop optimizations that can optimize across template boundaries. They presented the solutions to apply traditional dataflow analyses (e.g. liveness analysis) and optimizations (e.g. register allocation) to templates. In contrast, we integrated an optimizing compiler (i.e. LLVM) with QEMU, instead of trying to improve the template-based TCG compiler.

Swift [91], which is a JIT compiler on register-based DEX bytecode [27], is designed for Android platforms. Swift adopts a lightweight template-based code selector and targets for embedded systems. While translating a method, the machine code is generated by looking up a pre-defined template table indexed by DEX opcode, and the code emission is completed in a single pass. Without performing any data-flow or control-flow analysis at runtime, its translation process is very fast.

Maxine VM [68] is a Java Virtual Machine developed at Sun Microsystems. The Maxine

VM has no bytecode interpreter. Instead, it has a low overhead template-based baseline JIT [67]. The primary goal of the lightweight JIT is to produce code as fast as possible and code quality is of secondary concern. The baseline compiler generates code by translating each bytecode to a pre-defined template or sequence of machine instructions, and concatenating these sequences together to compile a method. Maxine also has an optimizing compiler which has higher compilation overhead than the lightweight JIT but produces code of better quality. The design strategy of Maxine VM is very similar to HQEMU, both having no interpreter and using multiple JIT compilers for different optimization levels. As for Swift and Maxine VM, their JIT compilers are designed for bytecode and based on the granularity of a method. In contrast, HQEMU retargets binary code and compiles traces and regions.

7.1.2 Language Virtual Machines

The Java HotSpot VM [75] is a method-based language virtual machine. When it compiles a method for the first time, it only compiles those basic blocks whose execution counts exceed a threshold during interpretation. If the execution frequently leaves a region from side exits, the JIT system expands this region to include those basic blocks that are the destinations of these side exits. Unlike Java HotSpot VM, HQEMU with the HPM-based region formation compiles traces at first and then merges separate traces which have frequent transitions with each other; HQEMU with the software-based region formation alternatively expands a trace to a bigger region at the time the trace is detected.

Ha et al. [40] proposed to spawn one dedicated helper thread for trace compilation so that concurrent interpretation and JIT trace compilation can be achieved. Their approach used trace profiling and prediction while interpreting JavaScript programs.

The Java HotSpot VM's parallel garbage collector [1] spawns multiple garbage collection threads to decrease garbage collection overhead, and hence increases application throughput. Its threading strategy does not improve the quality of code and requires the execution of guest program to stop while garbage collection is being performed.

7.1.3 Dynamic Binary Translation Systems

Dynamic binary translation is widely used for many purposes: transparent performance optimization [7, 83], security [78], runtime profiling [66, 74] and cross-ISA emulation [8, 9, 21]. With the advances of multicore architectures, several multithreaded DBT systems exploiting multicore resources for optimization have been proposed in the literatures. However, most of them have very different objectives and approaches in their designs.

A very relevant work to HQEMU is [54] which also integrates QEMU and LLVM. In their system, the authors target small programs with ARM to x86-64 emulation. It sends one block at a time to LLVM when the TCG translator determines it is worthy of optimization. Hence, the performance of the translated code was very poor. They also did not consider the retargetability issue in their DBT. It requires a different binary to LLVM translator for each different ISA, instead of using TCG as an IR as in HQEMU. Unlike their framework, HQEMU applies so-

phisticated LLVM optimization on traces. Therefore, it can benefit from the advantage of long traces. HQEMU also exposes the opportunities to eliminate redundant load/store instructions during code region transitions.

Brander et al. [15] also use LLVM for JIT compilation. They perform hot region profiling with a much higher threshold to keep the LLVM compile time low. In contrast, we can use a low profiling threshold (i.e. 50) because our LLVM translator runs on another thread, i.e., it does not interfere with the execution threads, and thus the translation overhead can be hidden.

Unlike Ha et al. [40] that used only one helper thread, Bohm [14] proposed the strategy of spawning multiple helper threads to accelerate the compilation of many traces. Their approach used trace profiling and prediction while interpreting guest programs. Instead of using interpreter, our emulation process is based on JIT compilation. We instrument trace detection routine in the block binary code, and efficiently redirect execution to trace cache as soon as the optimized code is ready. They also did not use HPM to reduce profiling overhead as in HQEMU during trace merging.

COREMU [87], a full-system emulator based on QEMU. It emulates multiple cores by creating multiple instances of sequential QEMU emulators. The system is parallelized by assigning multiple QEMU instances to multiple threads. With the same goal of COREMU, PQEMU [29] takes a different approach to have only one instance of QEMU but parallelizes it internally. Through sophisticated arrangement of critical sections, PQEMU achieves minimal overhead in locking and unlocking shared data. The advantage of their approach is that the performance of emulating multi-threaded programs can be enhanced because each guest thread is handled by a separate emulator thread. However, the emulation of single-threaded program cannot benefit as much because they did not try to optimize the target guest code in each thread. In contrast, HQEMU assigns DBT function to separate threads so very sophisticated optimizations can be applied to each guest thread without incurring overheads on the application threads. The performance of both single-threaded and multi-threaded guest programs can be improved on multicore systems.

7.2 Virtual Machines on Embedded Systems

Virtualization Tools Xen on ARM [51] and KVM for ARM [24] provided software solutions to enable full system virtualization before hardware virtualization technology becomes available on ARM platforms. Those software approaches use specialized VMM systems and need to modify the guest VMs in order to run on the ARM processors. Such approaches lose transparency. They also require the guest VMs and the host machine be the same architecture. In contrast, we focus on user-mode emulation. Our approach is based on dynamic binary translation that allows cross-ISA emulation without any modification to the emulated programs.

Dynamic Binary Translation Systems Guha et al. [38] and Baiocchi et al. [6] attempted to mitigate the memory pressure for embedded system by reducing the memory footprint in the code cache. [38] discovered that many codes in exit stubs are used to keep track of the branches of a trace. [6] also found that DBT introduces a large amount of meta-code in the code cache for its own purposes. They proposed techniques to re-arrange the meta-codes so that more space can be reclaimed for the application codes.

Baiocchi et al. [5] studied the issues of code cache management for dynamic binary translators targeting on embedded devices with a three-level cache hierarchy, ScratchPad memory, SDRAM and external Flash memories, the authors proposed policies to fetch or evict translated code among these three storage levels based on their size and access latency. Instead of using external memory, our work uses remote server's memory space as the extension to the thin client's code cache.

DistriBit [37, 61] also proposed a dynamic binary translation system in a client/server environment. The thin client in their system only consists of an execution cache and the engine. It does not have a translator. The whole translation process is sent to a remote server. The authors proposed a cache management scheme in which the decisions of a client are totally guided by the server in order to reduce the overhead on the thin client. Since only the server has the translation ability in their DBT system, the system would stop functioning if the translation service is not available. The emulation also needs to be suspended until the client receives the response from the server. Instead of using only one translator, our DBT system keeps a thin translator on the client. Thus, our system can tolerate network disruption or outage and still keep low overhead. Moreover, the performance of an emulation can be improved by asynchronous translation with our proposed two-translator approach where the communication latency can be hidden.

Zhou et al. [93] proposed a framework with a code server. Thin clients download the execution code to its code cache on-demand. In their work, the application code has to be pre-installed on the server before the clients can connect to it. In contrast, our DBT system does not require such pre-installation of application code. The server only needs to provide the translation service, and the client sends requests to the server at runtime.

7.3 Runtime Optimization Techniques

Trace Formation Dynamo [7] was the first trace-based dynamic optimizing compiler that used the Next Executing Tail (NET) algorithm. When the counter associated to a trace head reaches the threshold, NET speculatively follows the execution path during the execution of a trace head—the speculative sequence of blocks forms a NET trace. Dynamo pioneered many early concepts of trace formation and trace runtime management. Many dynamic compilation systems [14, 17, 19, 35, 52, 86] use NET or its variants to form traces.

Hiniker et al. [45] proposed the Last Executed Iteration (LEI) trace formation algorithm. Unlike NET which selects traces from the next executed blocks, LEI selects cyclic traces based on a history buffer containing the most recently interpreted taken branches.

MRET^2 [92] is a two-pass trace selection algorithm. In MRET^2, the potential hot traces are

selected twice from the same trace head. The hot trace is the common path of the two potential hot traces. The MRET^2 approach reduces the possibility of selecting a bad trace compared with other one-pass trace selection algorithms.

Hiniker et al. identified the trace separation problem in two trace selection algorithms, NET and LEI. The authors focused on the issues of code expansion and locality for same-ISA DBT systems. A software-based approach for trace merging was also proposed. Davis and Hazelwood [25] also proposed a software-based approach, called NETPlus, to solve trace separation problem by performing a search for any loop back to the trace head. Unlike their work, our work targets cross-ISA DBT systems and addresses issues of trace separation problem especially for performance and emulation overhead. We reduce redundant memory operations during region transitions and use a novel trace combination approach based on HPM sampling techniques.

Optimization for Indirect Branch Translation Optimization for indirect branch handling in DBT systems has been studied in several literatures [21, 28, 43, 46, 66]. Pin [66] uses an indirect branch chain, and for each indirect branch instruction, Pin associates it with one chain list. Unlike Pin, we use a big per-thread hash table shared by all indirect branches. Shadow stack [21], which is applied in Digital FX!32, is used to optimize the special type of indirect branch, *return*, by using a software return stack. This approach works fine for the guest architecture that has explicit call and return instructions, but it does not work for ARM because function return in ARM can be implicit. For such issue of implicit return, Hazelwood and Klauser [43] proposed using the combination of return stack and indirect branch prediction table for each indirect branch lookup. In contrast, we use IBTC for all indirect branch instructions, including returns. Our approach is retargetable for any guest architecture.

HPM-Based Feedback-Directed Optimization Kistler and Franz [57, 58] proposed an optimization framework which continually re-optimize programs based on the execution profiles collected from instrumentation and hardware counters. Their system is based on source code optimizations which re-optimize a high-level and type-safe intermediate format. In contrast, HQEMU operates on the binary images directly.

ADORE [65] is a lightweight dynamic optimization system based on HPM. It uses HPM sampling approach to collect path profiles from the Branch Target Buffer (BTB) hardware performance counters on Itanium, and forms traces based on the collected path profiles. Furthermore, performance bottlenecks of the running applications are detected and optimizations are guided according to variant performance monitors in its system.

Chen et al. [18] proposed to use multiple performance monitors to improve the accuracy of HPM sampling profiles. Based on the enhanced sample profiles, the off-line compiler can perform feedback-directed optimizations as good as using profiles from instrumentation-based profiling. These work motivate us to exploit HPM-based sampling techniques for our trace merge algorithm. However, their systems are not multi-threaded DBTs.

Chapter 8

Conclusion

Supporting retargetability in a DBT system imposes additional constraints on the structure of a DBT. Using a common IR in the translator is an effective approach to achieve retargetability, and guest instructions are translated to host instructions indirectly via the IR opcodes. The lightweight, template-based code emitter is inadequate for generating the optimal host instructions. The heavyweight aggressive optimizer produces too much translation overhead. Emulation overheads from region transitions, helper function invocation and thread synchronization, also cause the impediments to building an efficient DBT system. This dissertation explored approaches to overcome these issues, and designed an efficient and retargetable DBT system.

Tackling the dual issue of good translated code quality and low translation overhead, we presented HQEMU, a multi-threaded retargetable DBT system on multicores. HQEMU runs a fast translator (QEMU) and an optimization-intensive translator (LLVM) on different processor cores. Hence, such optimization overhead can be hidden without interfering with the execution of the guest program. With the hybrid QEMU+LLVM approach, we can benefit from the strength of both translators. We showed that this approach can be beneficial to both short-running and long-running applications.

We also recognized that considerable code region transition overhead is incurred from code region separation. We designed two kinds of region formation approaches, HPM-based and software-based trace merging, to improve existing trace selection algorithms. The novel HPM-based trace merging technique can detect traces that have trace separation problems, and merge separated traces based on the information provided by the on-chip hardware HPM. On the other hand, the software-based region formation combines the potential separate traces early in the program execution. It is helpful for emulating short-running applications, where the HPM sampling may not be able to gather sufficient sample profiles in a short run. We demonstrated that both approaches can result in the removal of redundant memory operations and improve the overall code performance significantly.

The IBTC optimization and lightweight memory transactions are also proposed to alleviate the problem of thread contention when a large number of guest threads are emulated. Our thread-private IBTC can keep the execution remaining inside the code cache, avoiding expensive context switch overhead as well as alleviating the huge contention overhead during the indirect branch

target lookups. Based on the lightweight memory transactions, the performance will not degrade much because the false protection of non-alias memory accesses and the overhead of expensive locking mechanism are eliminated as a result of the removal of global lock. We showed that these two optimizations can significantly reduce the emulation overhead of a DBT and make it more scalable.

To eliminate the emulation overhead from helper function invocations, the LLVM compiler infrastructure is leveraged so as to inline helper functions in the code cache. Moreover, we also implemented a two-level IR conversion to integrate the IR of QEMU and LLVM. Such conversion can simplify the design efforts tremendously—HQEMU supports all guest ISAs that are supported by QEMU.

Based on the hybrid two-translator architecture, we extended our retargetable DBT framework to the distributed architecture for embedded systems. We developed a distributed DBT of the client/server model: a thin translator on each thin client and an aggressive dynamic binary optimizer on the server to service the optimization requests from thin clients. With such a two-translator client/server approach, we successfully off-loaded the optimization overhead of thin clients to the server. Furthermore, the proposed asynchronous translation model can tolerate network disruption or outage for optimization services on a server, and to hide the optimization overhead and network latency.

We evaluated our retargetable DBT framework on both server and embedded platforms, and for several popular ISAs. For same-ISA emulations with single-thread benchmarks on the server platform (x86-to-x64 emulation with reference inputs), we found that using only LLVM translator (with IBTC optimization and elimination of redundant memory operations within each block) can reduce the normalized execution time to about 3.1-3.4X on average over native run. With HQEMU, trace formation and HPM-based trace merging can reduces it to 2.1-2.4X without multi-threads, and 2-2.2X with two translators running on different threads. The performance is further enhanced to 2-2.1X with software-based region formation. With multi-thread benchmarks, the normalized execution time is enhanced to about 3.3X on average, and the performance curve is very similar to that of native execution.

With the DBT of the client/server model, the performance can achieve 37% and 17% improvement over that of non-client/server model for x86-to-ARM emulation using embedded and general-purpose benchmarks, respectively, and 84% improvement using multi-threaded benchmarks running two emulation threads.

8.1 Research Impact

The base system, QEMU, is a state-of-the-art retargetable dynamic binary translator that supports many popular ISAs. QEMU has been widely used for many academic research projects, and also used in industry for software debugging and testing, validation of optimizing compilers, and early software development before hardware is available. This dissertation recognized several impediments to the performance in QEMU and explored several techniques to overcome such performance bottlenecks. We integrated QEMU with another well-known compiler frame-

work, LLVM, to build an efficient and retargetable dynamic binary translator. To our knowledge, our work is the first successful effort to integrate QEMU and LLVM to achieve significant improvement. Both same-ISA and cross-ISA emulation are getting closer to the performance of native runs. Thus, the enhancement to QEMU can make a broad impact on both industry and academic fields.

Although this research focuses only on QEMU and LLVM, we expect that the discussion of how to integrate two translators of different translation/optimization capabilities into a single framework can be beneficial to others trying to build similar systems.

Most techniques presented in this dissertation are based on high-level design methodology. The hybrid two-translator approach, including the multi-threaded and the client/server model, can be any combination of one fast translator plus aggressive optimizers. The HPM-based trace merging uses the instruction-retired event which is common on most processors. The two-level IR conversion scheme can also be applied with other IR designs. It can be easily extended or replaced with another existing IR frontend/backend so that adding a new guest/host architecture in a DBT becomes much easier. Therefore, we expect these techniques to be applicable to other DBT systems, and the optimizations to achieve performance portability as well.

8.2 Future Work

8.2.1 Full-System Virtualization

The DBT framework presented in this dissertation only focuses on the process-level emulation. In process-level emulation, the memory and I/O virtualization are mostly simplified. The virtual memory of the guest program tends to be mapped to the same address of the host space; I/O devices are not emulated and guest system calls are directly passed to the host OS. Hence, they incur almost no performance loss. On the contrary, memory and I/O virtualization play an important role in full-system virtualization. Preliminary study indicates that the software emulation of MMU and I/O devices can contribute to significant overhead and is one of the major impediments to the performance of QEMU.

Another issue in full-system virtualization is precise exception handling. Aggressive compiler, e.g. LLVM, may not retain precise states or allow state restoration after performing instruction removal or reorder with the aggressive instruction scheduling passes. Moreover, the optimization could fuse multiple guest instructions to one host instruction. Thus, the precise LLVM optimization passes need to be determined in order for HQEMU to support full-system virtualization.

8.2.2 Dynamic Optimization

Annotation Annotation data can be helpful for dynamic binary translation. An annotation-capable compiler can add annotation data into the executable, such as the CFG of a code section, loop or function boundaries, etc. Since the compiler can analyze program structure extensively during compile time, it can store those high-level information in the binary file to reduce runtime

analysis. Otherwise, these high-level information will be lost since it is impossible to recover these information from the compiled binary.

Feedback-Directed Optimization There are two directions for conducting feedback-directed optimization in a DBT system. One direction is to record the execution profiles for the first run and the profiles are used in profile-guided optimization for the next runs. This scheme is used to collect the dynamic information that is hard to get by the annotation-capable compiler. Such information includes edge probabilities, path profiles and the scopes of hot regions, etc. The scopes of hot regions are helpful for the NETPlus-based region formation to include more accurate and profitable basic blocks during the forward search.

The other direction is to seek for more optimization opportunities guided by hardware HPM-based profiling. In this dissertation, we exploited few HPM events in our dynamic binary optimizer. We plan to use variant performance monitors to detect the performance bottlenecks of the running applications, such as finding the delinquent memory operations. The potential of these two feedback-directed optimizations for dynamic optimization systems is discussed in [3, 31, 65].

8.2.3 Execution Migration

The client/server-based DBT framework in this research only off-loads the aggressive optimizations to the servers. The translated code is executed entirely on the client side. Actually, it is possible to divide the execution of the translated code between the client and the server dynamically. Thus, we could shift more tasks to the server for the goal of higher performance and low power consumption. However, system resources such as heaps and stacks, must be carefully monitored so that the division of work can be carried out correctly. Supporting such execution migration is quite challenging, and we leave this as an avenue for future work.

References

[1] Hotspot parallel collector. In *Memory Management in the Java HotSpot Virtual Machine Whitepaper*. 7.1.2

[2] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, 1989. 2.2.2

[3] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 92(2):449–466, 2005. 8.2.2

[4] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002. 2.2.1

[5] Jose Baiocchi, Bruce R. Childers, Jack W. Davidson, Jason D. Hiser, and Jonathan Misurda. Fragment cache management for dynamic binary translators in embedded systems with scratchpad. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 75–84, 2007. 7.2

[6] Jose A. Baiocchi, Bruce R. Childers, Jack W. Davidson, and Jason D. Hiser. Reducing pressure in bounded dbt code caches. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2008. 2.3.1, 7.2

[7] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12, 2000. 1, 2.3.1, 3.5, 7.1.3, 7.3

[8] Sorav Bansal and Alex Aiken. Binary translation using peephole superoptimizers. In *8th USENIX Conference on Operating Systems Design and Implementation*, 2008. 7.1.3

[9] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigel Zemach. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. In *36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003. 1, 1.2, 3.5, 6, 7.1.3

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *19th ACM Symposium on Operating Systems Principles*, 2003. 1.2

[11] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram

Schulte, Nikolai Tillmann, and Herman Venter. Spur: a trace-based jit compiler for cil. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2010. 2.1

[12] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference*, pages 41–46, 2005. 1.4

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th International Conference on Parallel Architectures and Compilation Techniques*, 2008. 5.3

[14] Igor Böhm, Tobias J.K. Edler von Koch, Stephen C. Kyle, Björn Franke, and Nigel Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *32nd ACM Conference on Programming Language Design and Implementation*, 2011. 7.1.3, 7.3

[15] Florian Brandner, Andreas Fellnhofer, Andreas Krall, and David Riegler. Fast and accurate simulation using the LLVM compiler framework. In *1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, 2009. 7.1.3

[16] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, pages 265–275, 2003. 1.2, 2.1, 2.3.4

[17] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, September 2004. 3.3, 5.1.2, 7.3

[18] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-Wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for FDO compilation. In *8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 42–52, 2010. 4.2, 7.3

[19] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, , and David Gillies. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2000. 7.3

[20] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. TaintTrace: Efficient flow tracing with dynamic binary rewriting. In *IEEE Symposium on Computers and Communications*, pages 749–754, 2006. 1

[21] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, 1998. 1, 1.2, 2.3.2, 2.3.4, 3.3, 7.1.3, 7.3

[22] CodeSourcery. GNU toolchain for ARM processors v2009q3-67. 3.7

[23] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991. 3.1

[24] Christoffer Dall and Jason Nieh. Kvm for arm. In *Annual Linux Symposium*, 2010. 7.2

[25] Derek M. Davis and Kim Hazelwood. Improving region selection through loop completion. In *ASPLOS Workshop on Runtime Environments/Systems, Layering, and Virtualized Environments*, 2011. 4.1, 7.3

[26] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, pages 15–24, 2003. 1

[27] DEX. Dalvik executable. http://en.wikipedia.org/wiki/Dalvik_(Software). 7.1.1

[28] Balaji Dhanasekaran and Kim Hazelwood. Improving indirect branch translation in dynamic binary translators. In *ASPLOS Workshop on Runtime Environments/Systems, Layering, and Virtualized Environments*, 2011. 7.3

[29] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. PQEMU: A parallel system emulator based on QEMU. In *International Conference on Parallel and Distributed Systems*, pages 276–283, 2011. 2.3.4, 7.1.3

[30] DragonEgg. A gcc plugin transforming c to llvm ir. http://dragonegg.llvm.org/. 3.6

[31] Evelyn Duesterwald. Design and engineering of a dynamic binary optimizer. *Proceedings of the IEEE*, 93(2):436–448, 2005. 8.2.2

[32] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 202–211, 2000. 3.3

[33] Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 50(6):529–548, 2001. 2.1

[34] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Ottawa Linux Symposium*, 2002. 5.2

[35] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *ACM Conference on Programming Language Design and Implementation*, 2009. 2.1, 7.3

[36] Susan L. Graham. Table-driven code generation. *Computer*, 13(8):25–34, 1980. 2.2.2

[37] Haibing Guan, Yindong Yang, Kai Chen, Yi Ge, Liang Liu, and Ying Chen. Distribit: a distributed dynamic binary translator system for thin client computing. In *19th ACM International Symposium on High Performance Distributed Computing*, pages 684–691, 2010. 7.2

[38] Apala Guha, Kim Hazelwood, and Mary Lou Soffa. Reducing exit stub memory consumption in code caches. In *2nd International Conference on High Performance Embedded Architectures and Compilers*, pages 87–101, 2007. 7.2

[39] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *4th Annual IEEE International Workshop on Workload Characterization*, pages 3–14, 2001. 6.3

[40] Jungwoo Ha, Mohammad R. Haghighat, Shengnan Cong, and Kathryn S. McKinley. A concurrent trace-based just-in-time compiler for single-threaded javascript. In *Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures*, 2009. 7.1.2, 7.1.3

[41] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In *16th International Conference on Distributed Computing*, pages 265–279, 2002. 5.2

[42] Hiroshige Hayashizaki, Peng Wu, Hiroshi Inoue, Mauricio J. Serrano, and Toshio Nakatani. Improving the performance of trace-based systems by false loop filtering. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 405–418, 2011. 2.1, 3.3

[43] Kim Hazelwood and Artur Klauser. A dynamic binary instrumentation engine for the arm architecture. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2006. 7.3

[44] Kim Hazelwood, Greg Lueck, and Robert Cohn. Scalable support for multithreaded applications on dynamic binary instrumentation systems. In *International Symposium on Memory Management*, pages 20–29, 2009. 3.3

[45] David Hiniker, Kim Hazelwood, and Michael D. Smith. Improving region selection in dynamic optimization systems. In *38th Annual IEEE/ACM International Symposium on Microarchitecture*, 2005. 3.4, 6.3, 7.3

[46] Jason D. Hiser, Daniel Williams, Wei Hu, Jack W. Davidson, Jason Mars, and Bruce R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *International Symposium on Code Generation and Optimization*, 2007. 7.3

[47] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores. In *International Symposium on Code Generation and Optimization*, pages 104–113, 2012. 6.2.4

[48] Raymond J. Hookway and Mark A. Herdeg. DIGITAL FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1):3–12, 1997. 1.2, 6

[49] Luke Hornof and Trevor Jim. Certifying compilation and run-time code generation. *Higher-Order and Symbolic Computation*, 12(4):337–375, 1999. 7.1.1

[50] Chun-Chen Hsu, Pangfeng Liu, Chien-Min Wang, Jan-Jan Wu, Ding-Yong Hong, Pen-Chung Yew, and Wei-Chung Hsu. LnQ: Building high performance dynamic binary translators with existing compiler backends. In *International Conference on Parallel Processing*, 2011. 2.1, 3.5

[51] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on arm: System virtualization using xen hypervisor for arm-based secure mobile phones. In *5th IEEE Consumer Communications and Networking Conference*, 2008. 7.2

[52] Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. A trace-based java jit compiler retrofitted from a method-based compiler. In *9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 246–256, 2011. 7.3

[53] Aamer Jaleel, Robert S. Cohn, Chi keung Luk, and Bruce Jacob. Cmp$im: A pin-based on-the-fly multi-core cache simulator. In *4th Annual Workshop on Modeling, Benchmarking and Simulation*, 2008. 2.3.3

[54] Andrew Jeffery. Using the LLVM compiler infrastructure for optimised, asynchronous dynamic translation in QEMU. Master's thesis, University of Adelaide, Australia, 2009. 3.5, 7.1.3

[55] Ho-Seop Kim and James E. Smith. Hardware support for control transfers in code caches. In *36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003. 5.1.3

[56] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, pages 191–206, 2002. 1

[57] Thomas Kistler and Michael Franz. Continuous program optimization: Design and evaluation. *IEEE Transactions on Computers*, 50(6):549–566, 2000. 7.3

[58] Thomas Kistler and Michael Franz. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems*, 25(4):500–548, 2003. 7.3

[59] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the linux virtual machine monitor. In *Ottawa Linux Symposium*, 2007. 1.2

[60] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004. 1.4, 3

[61] Lin Ling, Chu Chao, Sun Tingtao, Liang Alei, and Guan Haibing. Distribit: A distributed dynamic binary execution engine. In *3rd Asia International Conference on Modelling & Simulation*, 2009. 7.2

[62] Linux Perf Events. The performance monitoring interface for linux. https://perf.wiki.kernel.org/. 6.3

[63] llvm-gcc. A gcc driver transforming c to llvm ir. http://llvm.org/. 3.6

[64] Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Bobbie Othmer, Pen-Chung Yew, and Dong-Yuan Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 180–190, 2003. 1

[65] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei-Chung Hsu. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism*, 6:

1–24, 2004. 7.3, 8.2.2

[66] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM Conference on Programming Language Design and Implementation*, 2005. 1, 1.2, 2.3.3, 5.1.3, 5.1.3, 7.1.3, 7.3

[67] Maxine JIT. A template-based baseline compiler. https://wikis.oracle.com/display/MaxineVM/JIT. 7.1.1

[68] Maxine VM. Maxine virtual machine. https://wikis.oracle.com/display/MaxineVM/Home. 7.1.1

[69] Michael E. Maxwell, Patricia J. Teller, and Leonardo Salay. Accuracy of performance monitoring hardware. In *LACSI Symposium*, 2002. 4.2

[70] Duane Merrill and Kim Hazelwood. Trace fragment selection within method-based JVMs. In *4th ACM International Conference on Virtual Execution Environments*, pages 41–50, 2008. 3.3

[71] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *15th Annual ACM Symposium on Principles of Distributed Computing*, 1996. 3.2

[72] Ingo Molnar. The native POSIX thread library for linux. Technical report, Technical Report, RedHat, Inc, 2003. 5.2

[73] Steven S. Muchnick. *Advanced compiler design and implementation*. 1997. 3.1

[74] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Conference on Programming Language Design and Implementation*, 2007. 1, 7.1.3

[75] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspot(tm) server compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, 2001. 2.1, 7.1.2

[76] PandaBoard. http://pandaboard.org/node/300/#Panda. 6.3

[77] Perfmon2. The hardware performance monitoring interface for Linux. 3.7

[78] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, 2006. 1, 7.1.3

[79] Kevin Scott, Naveen Kumar, Sivakumar Velusamy, Bruce R. Childers, Jack W. Davidson, and Mary Lou Soffa. Retargetable and reconfigurable software dynamic translation. In *International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, pages 36–47, 2003. 1, 2.3.1

[80] Kevin Scott, Naveen Kumar, Bruce R. Childers, Jack W. Davidson, and Mary Lou Soffa. Overhead reduction techniques for software dynamic translation. In *18th International Parallel and Distributed Processing Symposium*, pages 200–207, 2004. 2.3.2, 2.3.4, 3.3,

5.1.1

[81] Frederick Smith, Dan Grossman, Greg Morrisett, Luke Hornof, and Trevor Jim. Compiling for template-based run-time code generation. *Journal of Functional Programming*, 13(3): 677–708, 2003. 7.1.1

[82] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. 2005. 1

[83] Swaroop Sridhar, Jonathan S. Shapiro, Eric Northup, and Prashanth P. Bungale. HDTrans: an open source, low-level dynamic instrumentation system. In *2nd International Conference on Virtual Execution Environments*, pages 175–185, 2006. 1, 7.1.3

[84] TCG. Tiny code generator. http://wiki.qemu.org/Documentation/TCG. 2.2.2

[85] Transitive corporation. Quicktransit software. http://www.transitive.com/. 1.4

[86] Cheng Wang, Shiliang Hu, Ho-Seop Kim, Sreekumar R. Nair, Mauricio Breternitz, Zhiwei Ying, and Youfeng Wu. StarDBT: An efficient multi-platform dynamic binary translation system. In *Asia-Pacific Computer Systems Architecture Conference*, 2007. 1.2, 7.3

[87] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. COREMU: a scalable and portable parallel full-system emulator. In *16th ACM Symposium on Principles and Practice of Parallel Programming*, pages 213–222, 2011. 2.3.4, 5.2, 7.1.3

[88] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995. 6.3

[89] Matt T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *International Symposium on Performance Analysis of Systems and Software*, 2007. 2.2.1

[90] Chen Yu, Ren Jie, Zhu Hui, and Shi Yuan Chun. Dynamic binary translation and optimization in a whole-system emulator -SkyEye. In *International Conference on Parallel Processing*, 2006. 1.4

[91] Yuan Zhang, Min Yang, Bo Zhou, Zhemin Yang, Weihua Zhang, and Binyu Zang. Swift: a register-based JIT compiler for embedded JVMs. In *8th ACM Conference on Virtual Execution Environments*, 2012. 7.1.1

[92] Chuck Zhao, Youfeng Wu, J. Gregory Steffan, and Cristiana Amza. Lengthening traces to improve opportunities for dynamic optimization. In *Workshop on Interaction between Compilers and Computer Architectures*, 2008. 7.3

[93] Shukang Zhou, Bruce R. Childers, and Mary Lou Soffa. Planning for code buffer management in distributed virtual execution environments. In *1st ACM/USENIX International Conference on Virtual Execution Environments*, pages 100–109, 2005. 7.2