國立臺灣大學電機資訊學院資訊工程學系

博士論文

Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
Doctoral Dissertation

可重定目標且高效能之動態二元碼轉譯器框架系統
Retargetable and Efficient Dynamic Binary Translation Framework

許俊琛
Chun-Chen Hsu

指導教授：劉邦鋒博士
共同指導教授：吳真貞博士
Advisor: Pangfeng Liu, Ph.D.
Co-Advisor: Jan-Jan Wu, Ph.D.

中華民國 104 年 6 月

June, 2015

國立臺灣大學博士學位論文
口試委員會審定書

可重定目標且高效能之動態二元碼轉譯器框架系統

Retargetable and Efficient Dynamic Binary Translation Framework

本論文係許俊琛君（學號 D95922006）在國立臺灣大學資訊工程學系完成之博士學位論文，於民國 104 年 7 月 17 日承下列考試委員審查通過及口試及格，特此證明

口試委員：

[illegible]
（指導教授）

徐慰中　　楊佳玲

郭大維　　吳真貞

羅習五

系　主　任　趙坤茂

Acknowledgements

Many people have contributed to this dissertation and I would like to acknowledge their contributions here. First of all, I thank my advisor, Prof. Pangfeng Liu, for providing me with the freedom and resources to do high-quality research, for being a caring teacher, and for teaching me all the fundamentals of being a good researcher. I want to thank my co-advisor, Dr. Jan-Jan Wu, for her high standards for research, endless patience and continuous encouragement. My life as a graduate student would have been very short and unproductive without her advices.

I'd like to give my very special thanks to Prof. Wei-Chung Hsu for always providing very useful technical insights and advices for my researches and for being a caring teacher. I thank Dr. Chien-Min Wang who always asks very sharp questions to polish my thoughts and ideas. I also thank Prof. Pen-Chung Yew for providing the research direction of this dissertation.

I want to thank members of the Computer System Group in Instituted of Information Science, Academia Sinica, and Parallel Distributed Lab in National Taiwan University. They all have contributed to this dissertation and my life in the past nine years. I am very appreicated those helpful discussions with Dr. Ding-Yong Hong in developing research ideas.

Finally, I cannot express with words my indebtedness to my parents and Celia for giving my their unconditional love and support at every step I have taken in my life. Even though they will likely not understand much of it, this dissertation would be meaningless without them.

Hsu, Chun-Chen,

Aug. 2015, Taipei

誌謝

許多人對這篇論文有著無比的貢獻，我想在此致上我的感謝之意。 首先，我要感謝的，是我的博士論文指導教授劉邦鋒老師。我對電腦科學的研究興趣是由劉邦鋒老師所啓發的， 沒有他我不會來唸博士班。在與他無數的討論過程中我學習到做學問的態度與方法，他亦是我學術研究上的典範。 我再來要感謝我的共同指導老師，吳眞貞老師。沒有吳眞貞老師的督促與無比的耐心以鼓勵就不會有這篇論文的產生。 我要感謝徐慰中老師，他總是在技術問題上給我珍貴的回饋與建議。我感謝王建民老師，他建立我對問題的敏銳度。 我亦感謝游本中老師指導了這論文的研究方向。我很感激我在博士研究上有以上老師的幫忙，無比地感謝。 我也感謝中央研究院的系統實驗室的各個成員以及在臺大資訊工程所 332 平行分散實驗室的各個成員。我特別感謝洪鼎詠博士，我總是可以在我們的討論中找到新的研究方向。

還有眞慧，感謝妳的陪伴，那總是可以安撫我燥鬱的心情，謝謝妳一起陪我走過博士最後這幾年。 最後要感謝我的父母，許民先生以及余月玲女士。您們是我很大的精神支柱，不只是我的博士在我一生的其他時刻也是。

許俊琛，

2015 年 8 月在臺北

Abstract

Dynamic binary translation is one of the core technologies in virtualization to boost the performace of instruction set architecture (ISA) simulation. The key factors to the performace of dynamic binary translators are the quality of translated code and the ability to detect hot regions at runtime. This dissertation builds a retargetable dynamic binary translator framework and provides two hot region detection approaches to improve the performance of dynamic binary translators.

The quality of translated code is critical to the performance of a dynamic binary translator, which implements the semantics of the *guest ISA* instructions with the *host ISA* instructions, so the translated code is often carefully hand-optimized. However a hand-optimized translator is not retargetabile because it takes tremendous implementation efforts for software engineers to port it to a new host ISA.

This dissertation first proposes an LLVM+QEMU (LnQ) framework for building high performance and retargetable binary translators with existing compiler modules. The goal of LnQ framework is to enable the process of building high performance and retargetable dynamic binary translators with existing industry-strength compiler optimization passes and code generation backends. Compared to QEMU, the LnQ shows more than 2X speedup in CINT2006 for ARM-to-x86_64 and x86-to-ARM dynamic binary translators compared to QEMU.

Besides the quality of translated code, the ability to detect “hot regions” of guest applications also determines the performance of dynamic binary translators. Most dynamic binary translators target traces, i.e. frequently executed code paths, as code regions to be translated and optimized. The *Next-Executing-Tail (NET)* trace formation method is an important example of such techniques. Many existing trace formation schemes are variants of NET.

This dissertation examines the inefficiency of NET-like trace formation algorithms. We found the formed traces may contain a large number of early exits that could be branched out during the execution. If this happens frequently, the program execution will spend more time in the slow binary interpreter or in the unoptimized code regions than in the optimized traces in code cache. The benefit of the trace optimization is thus lost. Traces with frequently taken early-exits are called *delinquent* traces.

This dissertation proposes a light-weight region formation technique called *Early-Exit Guided Region Formation (EEG)* to improve the efficiency of traces. It iteratively identifies and merges delinquent regions into larger code regions. It is shown the EEG achieves 1.23X and 1.11X speedup in CINT2006 for ARM-to-x86_64 and x86-to-ARM DBTs compared to NET.

This dissertation also studies the procedure-based dynamic binary translator that detects hot procedures as its compilation (i.e. translation and optimization) unit. We compare the performance of our EEG region formation algorithm with procedure region.

摘要

動態二元碼翻譯是虛擬化技術的核心技術因爲它可用來加速指令集模擬。 對於動態二元碼翻譯器而言，其關鍵在於所翻譯的二元碼的質量， 以及是否可以在程式執行過程找到再優化增進效能的程式區段。 在這論文中，我們呈現一個可重置的二元碼翻譯系統，此系統可產生高效能的動態二元碼翻譯器。 我們另提出二種執行熱區偵查方法來增加動態二元碼翻譯器的效能。 對一動態二元碼翻譯器而言，其翻譯的二元碼對於效能的影響甚爲重要。故通常我們會對翻譯碼做手動優化。 然而這樣手動優化過的翻譯器是難以重置到另一系統，因爲需要同樣的實作力氣來移植翻譯器到新的平台上。

此論文首先提出一容易重置的二元碼翻譯系統，稱爲 LLVM+QEMU (LnQ)， 利用現有的編譯器技術來產生高效能的動態二元碼翻譯器。與 QEMU 相比， LnQ 在 ARM 到 x86_64 及 x86 到 ARM 的動態二元碼翻譯器上於SPEC CINT2006中有高於2倍的效能表現。 此外，能否在程式執行過程中偵測出執行程式熱區也是影響效能的一關鍵因素。 在此論文中我們將指出現今熱區偵測方法不好的地方， 亦即其所偵測的熱區會無法完全執行，如果這樣的熱區很多的話將會導至效能不佳。 我們首先提出測量此一弱點的方法來證明此弱點眞的存在。進而我們再提出一改進此弱點的方法。 我們在此論文中提出一輕量級的熱區偵測技術稱爲「Early-Exit Guided Region Formation (EEG)」。 EEG 能持續地尋找出無法完全執行的熱區並將它與其他熱區合併來改進效能。 這方法對於 ARM 到 x86_64 的動態二元碼翻譯器能有效改進之前的方法約23%。對於x86_64 到 ARM的翻譯器約有11% 我們最後提出另一種以程序爲單位的熱區偵測方法。並比較EEG與此方法的優劣。

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Retargetability of Dynamic Binary Translators

Dynamic binary translation is a just-in-time (JIT) compilation from binary code of a *guest* ISA to a *host* ISA, which is widely used for transparent performance optimization [1–4], program instrumentation, security monitoring [5], supporting legacy applications, and system virtualization [6]. For example, Dynamo [7] uses dynamic binary translation techniques to improve execution performance. DynamoRIO [8] PIN [9], Valgrind [10], Strata [11, 12], and Umbra [13] uses dynamic binary translation techniques for program instrumentation.

If the guest ISA is different from the host ISA, we refer to it as a *cross-ISA* binary translator. Cross-ISA binary translators enable application to migrate from one hardware platform to another, or to provide a virtualized platform to run an application without the specific hardware. For example, FX!32 [14, 15] enables application to migrate from IA-32 to Alpha, and IA-32EL [16] enables application to migrate from IA-32 to Itanium. Other migration examples include [17–22]. QEMU [23], VmWare [24], and other virtulization systems [25–28] use binary translation technique to provide server virtualization.

It takes tremendous efforts to build a cross-ISA translator because a cross-ISA translator must prepare a *carefully hand-optimized* translation template [14, 16] for *each* guest instruction. As a result it takes the same amount of effort to build a new cross-ISA translator because of the preparation of these carefully hand-optimized translation templates.

The difficulty in building cross-ISA translators motivates us to develop the *LLVM [29] + QEMU* (LnQ) framework. The goal of LnQ framework is to enable the process of building high performance and retargetable dynamic binary translators with *existing* optimizers and code generation backends. We address the retargetability issue by translating the guest instructions into a machine-independent intermediate representations (IR), which can be used by other compiler modules, e.g., a code optimizer.

With existing mature optimization techniques available from compiler backend, the quality of the translated code can be greatly improved. To further improve performance without sacrificing retargetability, we show that many binary translation optimizations can be implemented at the machine-independent IR level, so that they only need to be implemented once in LnQ and can be applied to every translator built within LnQ framework thereafter.

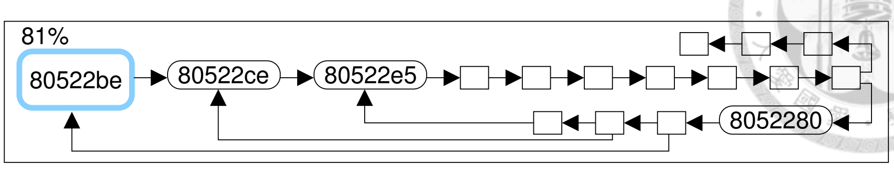
1.2 High Performance of Dynamic Binary Translators

Dynamic binary translators need to detect hot regions of guest applications at runtime in order to get high performance. The quality of the detected hot regions determines the extent and the types of optimization opportunities that can be exposed to dynamic binary translators, and thus, determines the ultimate quality of the final optimized code.

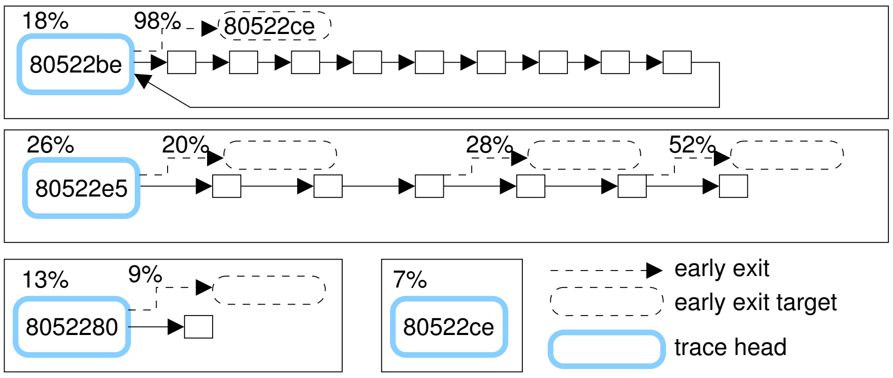
Nowadays, most dynamic binary translators target *traces*, i.e. frequently executed code paths, as code regions to be translated and optimized. Those dynamic binary translation systems [4, 8, 30] follow the well-known runtime trace formation algorithm, called Next-Executing-Tail (NET), developed in HP Dynamo [7]. The trace-based region formation approaches have also gained much attention in dynamic scripting languages [31, 32] and high-level language virtual machines [33–35] because they provide more optimization opportunities and generate higher quality codes.

Without expensive path profiling at runtime, NET forms a trace by selecting the basic blocks[1] that are most recently executed with minimal profiling overhead. The idea is that when a basic block becomes hot, it is likely that the following basic blocks are also hot.

[1]A basic block is a sequence of instructions terminated by a control transfer instruction



(a) CFG of the for–loop in 456.hmmer.



(b) Traces generated by NET

FIGURE 1.1: An example of delinquent traces of NET in 456.`hmmer`.

1.2.1 Delinquent Trace

As a hot trace is formed by cascading a sequence of hot basic blocks, there will be a conditional branch at the end of each member basic block, referred to as the *early exit* of the trace. DBTs needs to generate *compensation code* at each of such early exits to handle the case when the conditional branch is taken [6]. If early exits are frequent, then not only will such extra compensation code need to be executed, but also program execution will spend more time in the slow binary interpreter or in unoptimized code regions. The benefit of trace optimization by the DBT is thus lost. Traces with frequently taken early-exits are called *delinquent* traces.

Since NET does not use edge profiling [7] information to select next basic blocks, early exits may occur when program behavior changes in different execution phases. For example, as shown in Figure 1.1(a), the function `P7Viterbi` in `456.hmmer` (a SPEC CINT2006 CPU benchmark) contributes most of its execution time. `P7Viterbi` updates global variables according to different conditions in a performance critical `for`-loop.

NET splits the for-loop into four traces as shown in Figure 1.1(b). Each large rectangle represents a trace. The execution time of each trace, shown as the percentage of total execution time, is noted on the left top corner of the trace. The probability of an early exit being taken is also noted on each exit edge. Figure 1.1(b) shows a trace for a loop starting at 0x80522be. The probability of taking an early exit during the loop execution is 98%[2]. Such a high probability for an early exit will certainly diminish the performance benefit expected from the loop trace.

1.2.2 Solution 1: Early-Exit Guided Region Formation

To accomplish this, this dissertation proposes a light-weight technique called *Early-Exit Guided (EEG) region formation* to detect and merge delinquent regions. For example, The EEG region formation technique will merge those four traces into a large code region shown in Figure 1.1(a), which can improve its performance by 68%.

There are two key issues in EEG: (1) which regions should be merged, and (2) when to merge those regions. A simple approach for the first issue is to instrument counters into all traces. However, this approach is prohibitively expensive. Instead, we employ hardware-assisted dynamic profiling to select hot regions and to avoid monitoring and merging unimportant regions. To address the second issue, we

[2]Section 3.3 describes the experiment environment and methodology

monitor regions by instrumenting counters to detect early exits. When the counter exceeds a threshold, we merge this region with the region that begins at the branch target of the early exit. We also employ a heuristic to decide whether it is beneficial to merge the selected regions or not. We will not merge regions if it will cause too much register pressure; i.e. too many store/load operations to spill and fill values between registers and the stack (see Section 3.2.4).

We also extent LnQ to a multi-threaded DBT system. In such a system, there is a pool of compilation threads. Each thread is capable of taking one trace as a compilation (i.e. translation and optimization) unit. Similar to the work of Bohm et al. [30], we separate compilation tasks from execution tasks to hide the compilation overhead. This is especially important because compiling procedures could induce large overhead if not off loaded from the execution.

1.2.3 Solution 2: Trace-Guided Procedure-Based Region Formation

By forming a larger code region beyond traces to an entire *procedure*, we can resolve the inefficiency issues of delinquent traces since the early exits of traces will be likely within the same procedure. Furthermore, as hot traces are usually identified more quickly than hot procedures, we could form and optimize traces to improve performance before hot procedures are identified. Identifying hot procedures basically uses the same profile information collected for trace formation. No additional profiling overhead is incurred. These advantages motivate us to integrate trace-guided optimizations into a procedure-based DBT.

Unlike traces, procedures often contain hundreds of blocks. Such large compilation scopes can better utilize the power of LLVM backend optimizations, e.g. loop-based optimizations, for higher quality code. We also identify and resolve issues that are specific to procedure-level compilation in DBT systems. For instance, information on target blocks of indirect branches are not readily available in binary, as opposed to high-level language virtual machines in which richer knowledge about procedures is available to their JIT compilers.

We also identify a *Call-Return problem* of procedure-based dynamic binary translation, in which a guest call instruction could make the block at its returned address unreachable. We solve this problem by making the unreachable block an entry block of the translated procedure and make it reachable (see Section 4.2.1).

We also propose two feedback-directed optimizations on procedures (see Section **??**). The first optimization is Trace-Guided Block Reordering, in which it uses frequently-taken paths to order the basic blocks in translated procedures. The other optimization is Trace-Guided Partial Inlining, in which blocks of the taken paths in a called procedure are included using hot traces detected during hot trace-formation, as opposed to using meta-data annotated in byte code in Java [36].

1.3 Contributions

This dissertation makes the following major contributions:

- This dissertation presents the design and implementation of the LnQ framework, which can build high performance multi-threaded trace-based/procedure-based dynamic binary translators. We introduce several novel approaches to build high performance dynamic binary translators with existing compiler modules. LnQ dynamic binary translators can off-load compilation overhead to other cores and allow more aggressive and sophisticated optimizations to be done on the larger code regions during execution. We build x86-to-x86_64, ARM-to-x86_64, x86-to-ARM, ARM-to-ARM binary translators to demonstrate the performance of LnQ.

- This dissertation identifies delinquent traces, and shows that many SPEC CINT2006 benchmarks suffers from the inefficiency of traces. Our experimental results show that there is a substantial amount of delinquent traces, and that more than 100 early exits are taken for every million executed instructions in 65% of SPEC CINT2006.

- This dissertation presents an *Early-Exit-Guided* region formation algorithm (EEG) that uses hardware-assisted dynamic profiling and instrumented software counters to detect and merge delinquent traces/regions into larger regions to resolve the efficiency problem of delinquent traces.

- This dissertation also presents thread-guided procedure-based region formation algorithms in dynamic binary translators. We identify a number of important issues in such a system and propose promising solutions to these issues. Two feedback-directed optimizations for procedures are proposed to leverage runtime profile information collected during trace formation and optimization. The Trace-Guided Block Reordering approach guides the ordering of basic blocks in procedures according to frequently-taken paths detected during trace formation. And Trace-Guided Partial Inlining approach inlines hot code regions found during trace optimization.

1.4 Dissertation Organization

This dissertation has six chapters. Chapter 2 presents the design and implementation of LnQ framework including how we implement 3 retargetable runtime optimizations in LnQ. Chapter 3 shows techniques to detect delinquent traces and presents the EEG region formation algorithm for delinquent traces. Chapter 4 presents procedure-based region formation algorithms. Chapter 5 describes related works in dynamic binary translation and region formation. Chapter 6 provides concluding remarks to summarize our key results and insight presented in this dissertation, and suggestion for future researches in region formation.

Chapter 2

The LLVM+QEMU (LnQ) Framework

2.1 LnQ: Design and Implementation

This section describes the design of LnQ – a framework that builds high performance dynamic binary translators. The LnQ framework consists of two modules - an *emulation module* and a *translation module*, as shown in Figure 2.1. We use QEMU [37] as our emulation module, and build the translation module with LLVM [38] compiler infrastructure.

We first describe the design and execution flow of the translation module. Our translation module contains an *LLVM IR translator*, *IR libraries*, and an *instruction description table* of guest ISAs, as shown in Figure 2.2.

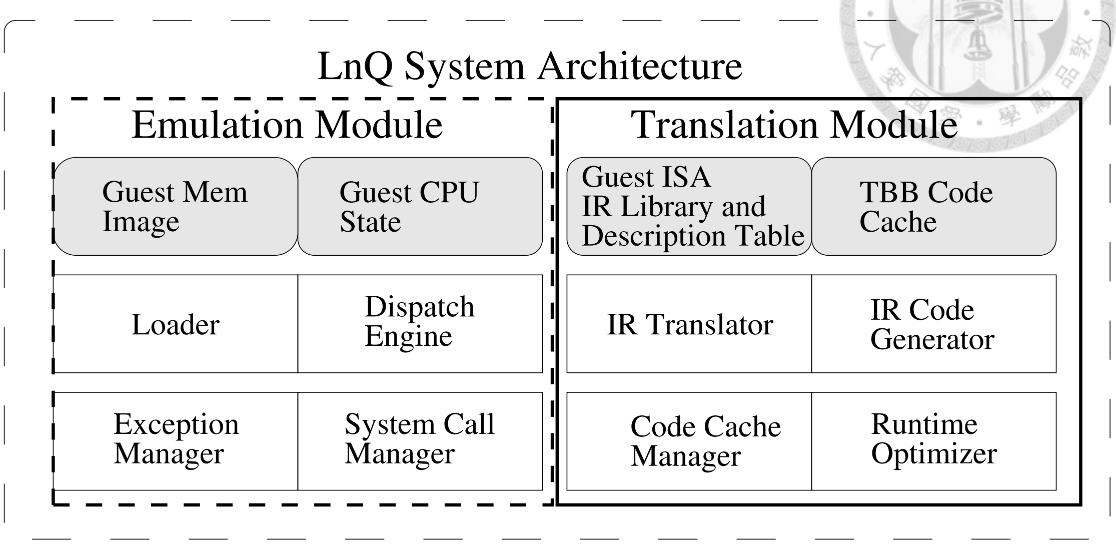


FIGURE 2.1: The architecture of LnQ framework

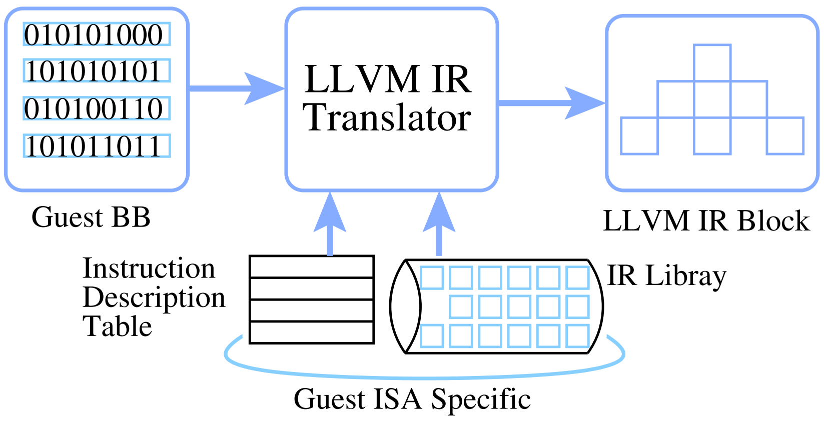


FIGURE 2.2: Illustration of the design and execution flow in LnQ translation module.

2.1.1 LLVM Intermiediate Representation

We choose *LLVM virtual instruction set* as our intermediate representation (IR) of guest instructions for the following three reasons. First, the LLVM compiler infrastructure provides modular and reusable components for building an efficient just-in-time (JIT) runtime system. The LLVM infrastructure reduces the time and cost to develop LnQ framework. Second, LLVM is an open source project and is well-documented. We believe that an open source project greatly improves component interoperability and enables future extension of our research work. Thirdly, LLVM IR is well-defined and can be easily manipulated by a rich set of API.

2.1.2 IR Library and Instruction Description Table

To support a new guest ISA in LnQ framework, we need to provide LnQ the *IR library* and the *instruction description table* of the guest ISA. The IR library consists of pre-built LLVM IR templates for every guest instruction. These templates are referred to as *translation functions.* Each translation function implements the semantics of a guest instruction in `C`, and is compiled into LLVM IR with LLVM-enabled compliers, such as *clang, llvm - gcc.*

In addition to IR library we also need to provide an instruction description table that describes the *properties* of parameters related to the translation function of each guest instruction. A property of a parameter contains the type of the parameter (e.g. a constant or a register ID), and how to obtain the parameter (e.g. from the immediate operand of the decoded guest instruction).

This property approach improves guest ISA retargetability because guest ISA specific IR libraries and instruction description tables can be "plugged into" an LLVM IR translator. To support a new guest ISA, we only need to implement the semantics of all guest instructions with LLVM IR. Since the translation functions are written in `C` rather than in LLVM IR, it becomes much easier to support a new guest ISA in our LnQ framework.

2.1.3 LLVM IR Translator

The LLVM IR translator translates guest instructions into LLVM IR one guest basic block at a time, and passes the LLVM IR to LLVM JIT to generate host instructions. Figure 2.3 Illustrates all steps of the translation flow.

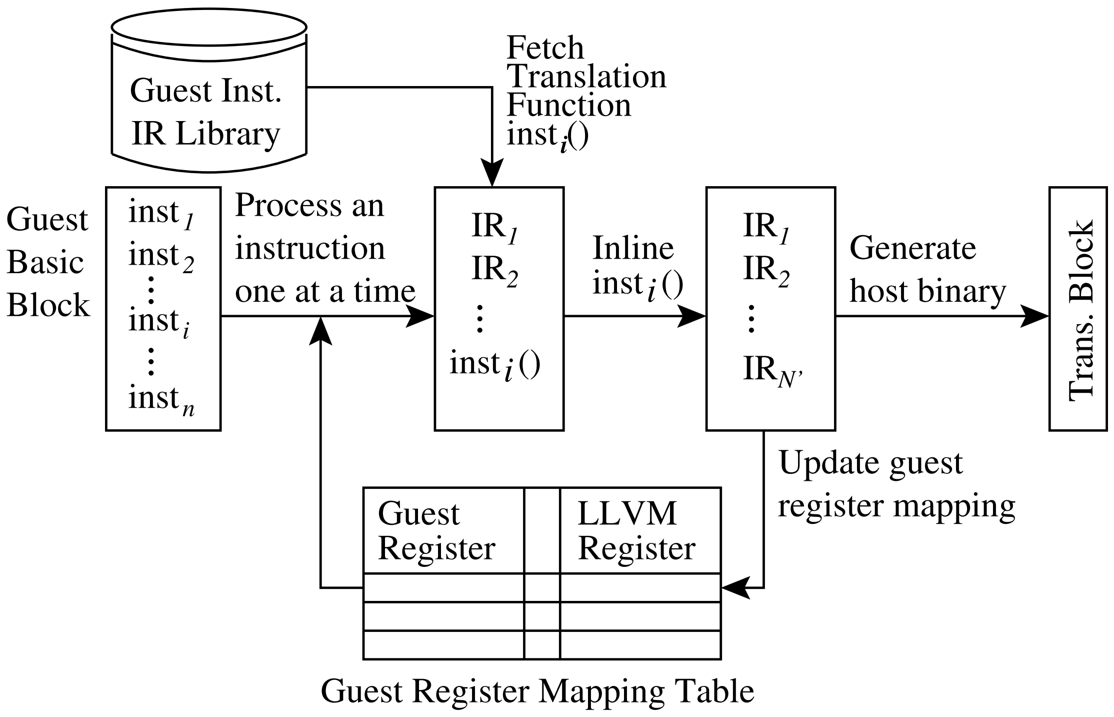

Guest Inst. IR Library
Fetch Translation Function $inst_i()$
Guest Basic Block
$inst_1$ $inst_2$ ⋮ $inst_i$ ⋮ $inst_n$
Process an instruction one at a time
IR_1 IR_2 ⋮ $inst_i()$
Inline $inst_i()$
IR_1 IR_2 ⋮ $IR_{N'}$
Generate host binary
Trans. Block
Update guest register mapping
Guest Register
LLVM Register
Guest Register Mapping Table

FIGURE 2.3: Illustration of the translation process.

First the translator creates an empty LLVM function as the container of the generated IRs. Then the translator decodes a guest basic block from the starting address until it encounters a control transfer instruction, such as calls and branches. For each decoded instruction the translator fetches its translation function from IR library and supplies the needed arguments according to the instruction description table.

The translator creates a call instruction to the translation function after generating its arguments, inlines the translation function into the function body, then generate optimized code. After inlining all translation functions of guest instructions in current guest basic block, the generated LLVM function is sent to an optimization pass manager for optimization, and then sent to the code generator to generate host instructions. The LLVM inline API provides several optimizations to the inlined LLVM IRs, such as *Constant Propagation* and *Unreachable Basic Block Elimination*.

2.1.3.1 Register Mapping

Register mapping maps guest registers to host registers in order to eliminate redundant loads and stores of guest registers. If we map guest registers to host registers, then we can retrieve guest register values from host registers, and only need to update guest registers values (kept in memory) when we leave a translated block.

We map guest registers to LLVM registers, and let LLVM register allocator to map LLVM registers to host registers. However, LLVM IR must follow Static Single Assignment (SSA) form in which each LLVM register can only be defined once. As a result we need a mapping table to map each guest register to its *current*

LLVM register, so that we can determine which LLVM register has a particular guest register from the mapping table.

The LLVM IR translator maintains the mapping table as follows. Initially the mapping table is empty. If we cannot find the corresponding LLVM register of a guest register in the mapping table, then the guest register was not loaded yet. The translator then creates a load instruction to load the guest register from memory to an LLVM register, and updates the mapping table so that future references to that guest register will be mapped to the LLVM register.

The translator looks up the mapping table mainly to determine which LLVM registers to use when constructing the arguments of a translation function. After constructing the arguments, the translator inlines the translation function.

If an inlined function, which is compiled by a LLVM-enabled complier, modifies guest registers, then according to SSA constraints on LLVM register, the LLVM-enabled compiler places the new value of the guest register into another LLVM register. Therefore we need the following information to update the mapping table correctly – the modified guest registers and the new LLVM register this guest register should be mapped to.

Our solution is to associate the information of modified guest register and new LLVM register with every translation function. When the translator inlines the translation functions, it retrieves these information and will be able to update the mapping.

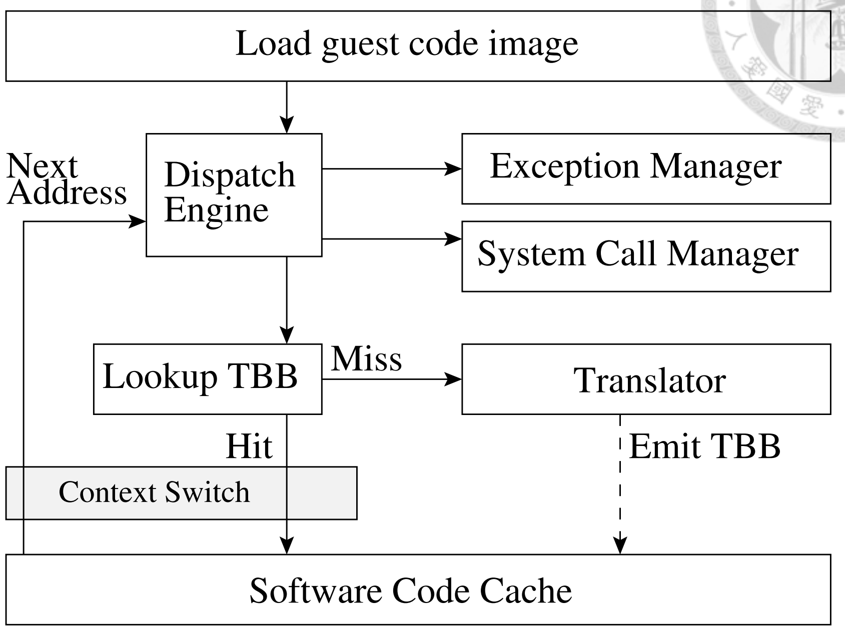


Figure 2.4: Illustration of the control flow in emulation module.

2.1.4 Emulation Module

Figure 2.4 illustrates the execution flow of a dynamic binary translator. First the loader loads the guest application image into memory. The guest code image consists of basic blocks that end with a control transfer instruction, such as calls, branches, etc. Currently LnQ uses guest basic block as the unit for translation and execution. The emulation module then initializes the guest CPU state which represents the state of the guest machine processors. For example, an x86 guest CPU state contains program counter, general purpose registers, and the eflags register, etc.

After initialization, the dispatch engine tries to locate the translated basic block of the current guest basic block, pointed by the guest program counter, with a directory to map the address of guest basic blocks to memory locations of their translated blocks in code cache. If the directory reports a hit for the current guest basic block, the control transfers to the memory location of its translated block in code cache. If we cannot find the translated block for the current guest basic block, the control transfers to the translation module, which will translate the current guest basic block and add an entry into the directory.

After the translated basic block is executed, the execution control transfers back to the emulation engine, which is referred to as *context switching*. The process is repeated until program terminates.

2.2 Runtime Optimizations

We can also integrate runtime optimizations that manipulate LLVM IRs to further improve the performance in LnQ framework. Those optimizations should be retargetable and not depend on either guest ISA or host ISA. Retargetability is an important goal of LnQ framework, and we expect those optimization techniques can be reused in all binary translators for all ISAs.

To demonstrate this ability, we implemented three classic optimizations that reduce the frequency of context switch between emulation engine and code cache. The three optimizations are *block linking*, *indirect branch target caching*, and *shadow stack*. We describe each of them in the following sections.

2.2.1 Block Linking

Block linking [7] links translated blocks in code cache so that program execution transfers directly from one code block to another so as to eliminate expensive context switching. Block linking targets at blocks that have a direct branch, a conditional direct branch, or a direct call instruction as the last instruction. We use “exit” to refer to these jump or call instruction that are at the end of a block, and each exit has an unique *exit ID*, and the destination of an exit is refer to as *branch target*.

Block linking can be either *proactive* or *lazy*. Proactive block linking links translated basic blocks whenever a new block is generated. A lazy block linking links translated basic blocks when a context switch occurs.

We choose lazy block linking for two reasons. First, lazy block linking links translated basic blocks only when the execution actually goes from one block to another. Second, when a new block is generated, proactive block linking must update the branch targets of all exits that go to the newly generated block, therefore we need a data structure that maps branch address of exits to the starting address of translated basic blocks. In contrast lazy block linking does not require this extra data structure, and will not incur extra maintenance overheads. Further comparisons between proactive and lazy block linkings can be found in [8, 39].

We implement block linking as follows. Before an execution thread leaves a block it stores the exit ID of the block in a specified thread-private memory location. Then the emulation engine locates the translated basic block pointed by the branch target, retrieves the exit ID that we just stored, then uses this exit ID to look into

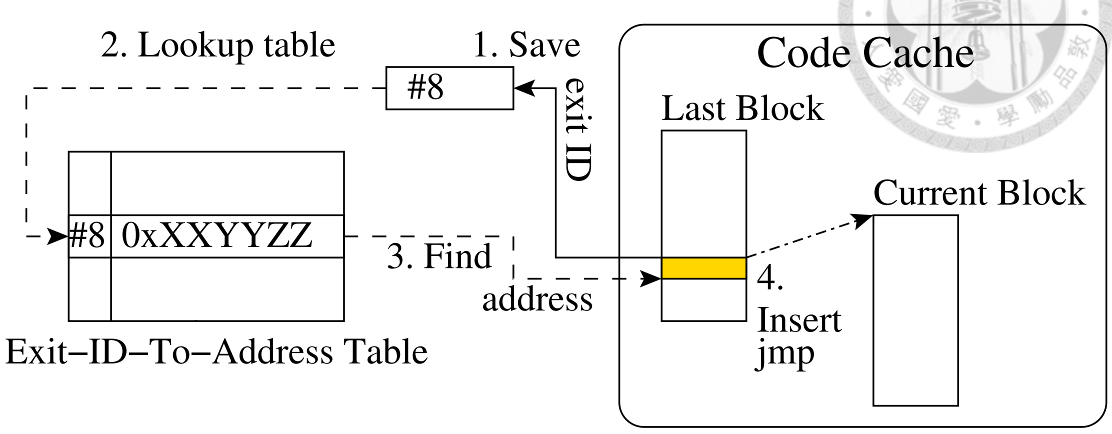


FIGURE 2.5: Illustration of execution steps of block linking optimization.

an *exit-id-to-address* mapping table that maps exit id to the address of an exit in code cache.

The entries of exit-id-to-address table is added when LLVM JIT generates host instructions for exits. Then the emulation engine will be able to patch the branch target of the exit of the block the execution thread just left. Please refer to Figure 2.5 for an illustration of our implementation.

2.2.2 Indirect Branch Target Caching

To further reduce the context switches, We use the indirect branch target caching (IBTC) to fast look up whether the target of the indirect branch has a translated basic block in code cache without returning to emulation engine. We add a IBTC lookup function calls at the ends of those blocks that end with indirect jumps and indirect calls instructions.

The IBTC contains a *shared cache* as shown in Figure 2.6. The shared cache is implemented as a direct-access hash table with 1K entry to cache all indirect branches. The hash table is indexed by the last 10 bits of the guest target address. Given a guest target address, we use the last 10 bits as the key to index the shared cache. If the comparison successes, the control then transfers to the memory location of the translated basic block. Otherwise, the control transfers back to the emulation engine, and updates the shared cache after locating the TBB.

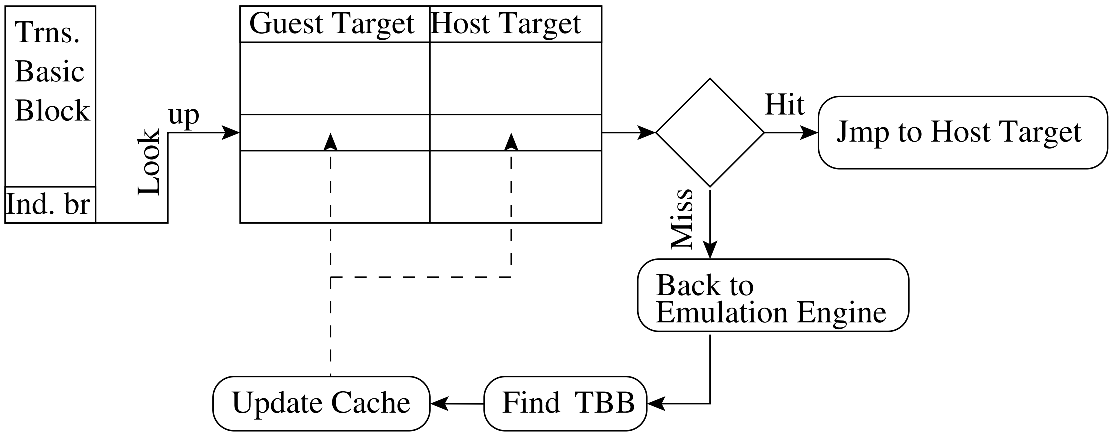


FIGURE 2.6: Illustration of execution steps of indirect branch target caching optimization.

2.2.3 Shadow Stack

Shadow stack (SS) optimizes function return mechanism in binary translation, and was first introduced in FX!32 [14]. Despite that a function return instruction can be viewed as an indirect branch, it can be optimized without indirect branch cache lookup. This is because the guest return address is known when the translator translates a guest call instruction since the call pushes the guest return address onto the stack.

If the translated basic block of the guest return address exists in code cache, the translator can push the memory location of the translated basic block onto a shadow stack, from which we can fetch the host return address when the function ends without looking up indirect branch cache or going back to emulation engine. However, if the block of the guest return address is not translated, we push the address that goes back to emulation engine.

The details of our implement of shadow stack are as follow. We assign a memory location, called *address box*, to each guest call instruction, which stores either the host return address or the return address back to emulation engine. If the guest return address does not yet have its translated basic block in code cache, we marks the address box as untranslated and stores the return address back to emulation engine in the address box. Then, when the translator generates a translated basic block, it checks whether there is a address box that should store the address of this translated block but is marked untranslated. If such address box is found, the translator marks the address box as translated and stores the address of translated basic block into the address box. For each guest call instruction, we insert instructions to push the content of its address box on top of the shadow stack. Please refer to Figure 2.7 as an illustration.

As for each guest return instruction, we insert pop shadow stack instructions in the end of the translated block. Note that we need to perform a check to see whether the guest return address is matched to the one stored on top of the shadow stack. If they are matched, the execution directly transfers to the address that is popped from the shadow stack. If the addresses are not matched, we flush the shadow stack since the shadow stack is no longer valid, and the execution transfers back to the emulation engine.

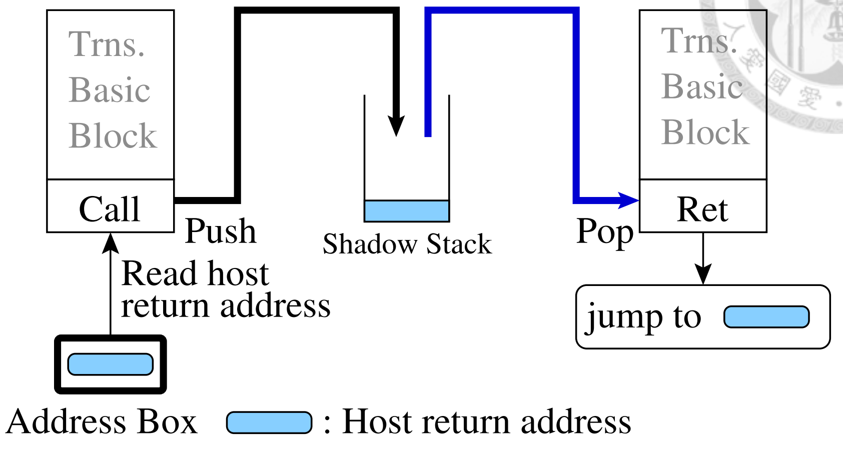


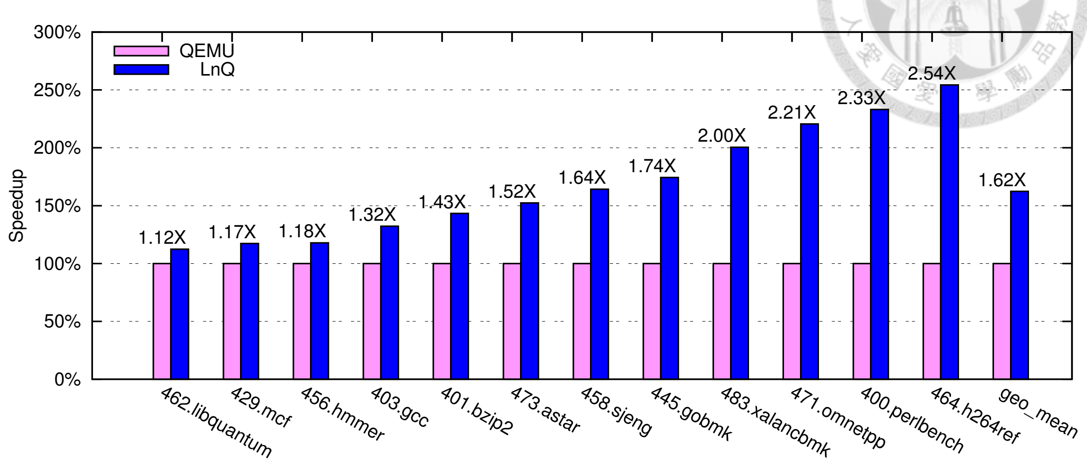
FIGURE 2.7: Illustration of execution steps of shadow stack optimization.

2.3 Performance Evaluation

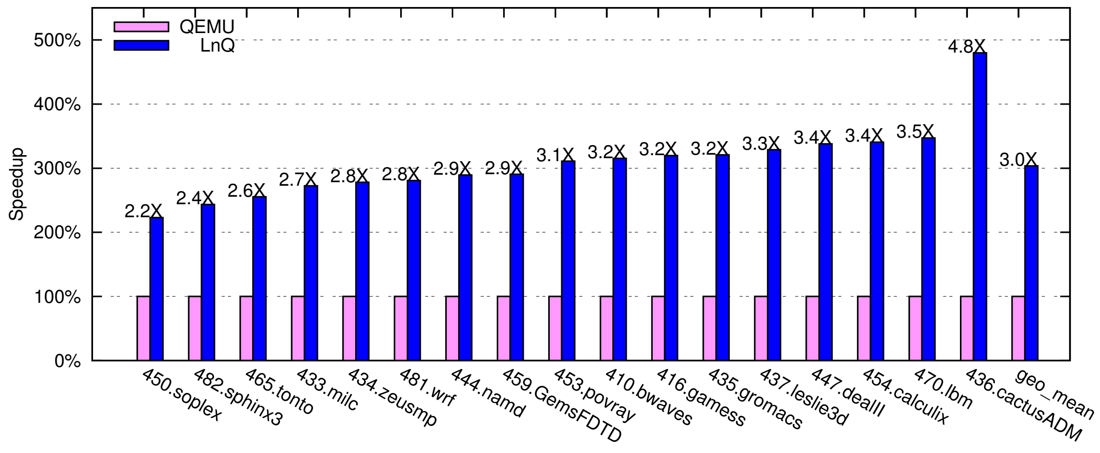
We conduct experiments to evaluate LnQ by building an x86-to-x86_64 dynamic binary translator with the LnQ framework. We use QEMU version 0.13.0 as the emulation engine module, and use LLVM version 2.8 to implement the translation module.

LnQ uses an LLVM class `MCDisassembler` to disassemble x86 machine instructions. We use `X86GenDisassemblerTables.inc` to generate the prototypes of translation functions and the instruction property table. We also use template functions to speedup the implementation of translation functions. For example, we use the `add()` template function to implement variations of `add` instructions for different types of parameters. We have implemented all instructions needed by SPEC CPU 2006.

We modified the LLVM library slightly to meet the needs of the emulation module. We instruct the register allocator not to use a specified host register which holds



2.8(A): Results of Integer Benchmarks



2.8(B): Results of Floating Point Benchmarks

FIGURE 2.8: Speedup factors of LnQ in of SPEC CPU2006 compared with QEMU.

the base memory address of the guest CPU state. This is important because a dynamic binary translator needs one host register to hold the base address of the guest CPU state during the execution. For example, QEMU uses `r14` as the CPU state register in x86_64 host machine. If the register allocator allocates `r14` register, the content of `r14` will be overwritten and the application will crash.

We use the default optimization option provided by LLVM Just-In-Time compiler. The optimization level invoked by this default option is equivalent to GCC "-O2".

2.3.1 Experiment Settings

Our experiments were conducted on an Intel Core2 CPU 975 @ 3.33GH machine with 12GB of memory. The operating system is 64 bit Gentoo distribution Linux. We use SPEC CPU 2006 as our benchmarks. All benchmarks are compiled with GCC 4.3.4 with "-O2 -m32" flags.

We run all benchmarks via the standard SPEC *runspec* script with configuration files. We run each benchmark three times with reference inputs and take the average as the experiment result. We compare the performance of our LnQ with QEMU 0.13.0.

2.3.2 Performance of LnQ

We first compare the runtime performance of LnQ with QEMU. The results are shown in Figure 2.8 where Figure 2.8(a) shows results of integer benchmarks and Figure 2.8(b) shows results of floating point benchmarks. The Y-axis is speedup factor of running time of LnQ compared with QEMU.

As shown in Figure 2.8(a), the speedup factors range from 1.12X to 2.54X, and LnQ is 1.62 times faster than QEMU on average for integer benchmarks. The speedup factors improve significantly in floating benchmarks. For SPEC CFP

2006 benchmarks, the geometric mean of speedup factors is 3.02X compared to QEMU. The speedup factors range from 2.24X to 4.8X.

The main reason of this significant improvement is due to insufficient translation ability of QEMU. The QEMU translator, called Tiny Code Generator (TCG), does not support floating point operations yet [40]. As a result TCG translates all floating point instructions into helper function calls. On the other hand, the LLVM backend used in LnQ could generate host floating point instructions directly, and hence we gains much performance improvement than QEMU.

2.3.3 Performance of LLVM Just-In-Time Compiler

In this section, we evaluate the performance of LLVM Just-In-Time (JIT) compiler used in LnQ. As described in Section 2.1.4, the total running time of a dynamic binary translator can be divided into three portions: the dispatch time, the translation time and the execution time spent in code cache. Therefore, in the following two sections, we first evaluate the execution time spent in code cache, and then we evaluate the translation time of LLVM JIT.

Note that, for the sake of simplicity of presentation, we list only results of representative subset of SPEC 2006 as suggested in [41] in following experiments. We, however, still show the geometric means derived from all benchmarks of SPEC CINT 2006 and SPEC CFP 2006.

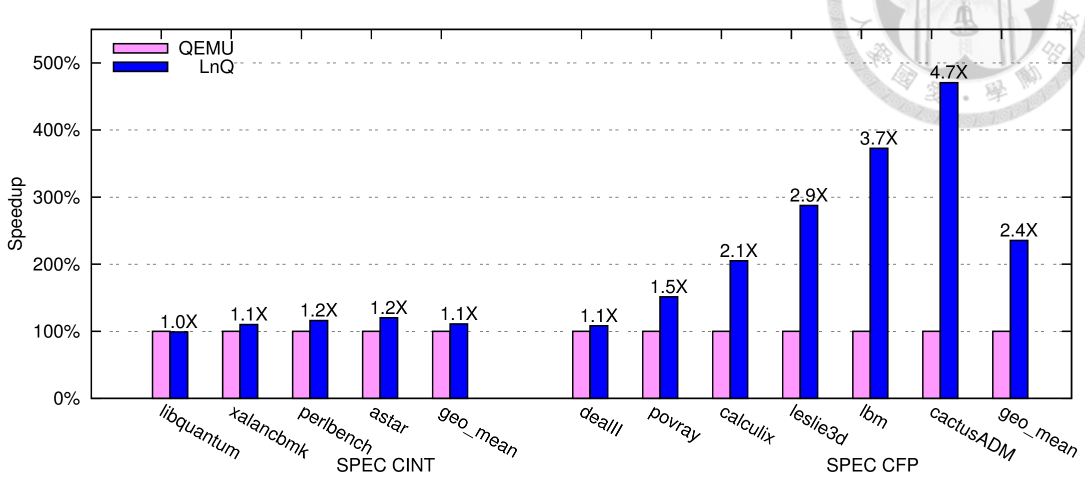


FIGURE 2.9: Speedup factors of execution time spent in code cache of LnQ compared to QEMU. The numbers above bars are

2.3.3.1 Execution Time Spent in Code Cache

We begin by evaluating the execution time of LnQ and QEMU. To be a fair comparison, we turn off all the runtime optimizations used in LnQ and QEMU. We turn off Block Linking, IBTC, and Shadow Stack in LnQ, and Block Linking in QEMU, which is only runtime optimization in QEMU. We profile the execution time by insert timing function before entering code cache and after exiting code cache. The results are shown in Figure 2.9.

From Figure 2.9, we see the execution time spent in code cache of LnQ improves about 10% in integer benchmarks and about 135% in floating point benchmarks. The significant improvement of floating point benchmarks is again contributed by translation ability of LLVM JIT as explained in previous section. The improvement of translation quality in integer benchmarks is not as expected. Thus, the major improvement of integer benchmarks shown in Figure 2.8 can be contributed

	Benchmarks	Blocks	LnQ		QEMU	
			Total	Translation	Total	Trns.
CINT	perlbench	57592	1819	54.8 (3.0%)	4240	0.22
	libquantum	2801	1825	3.0 (0.2%)	2051	0.01
	astar	7970	1180	11.8 (0.7%)	1798	0.04
	xalancbmk	29065	1208	27.6 (2.3%)	2421	0.12
	geo. mean of trns.			.16%		0%
CFP	leslie3d	6074	3147	10.6 (0.2%)	10344	0.03
	calculix	14992	9871	18.6 (0.2%)	33622	0.09
	cuctusADM	9295	5650	10.6 (0.2%)	27114	0.04
	dealII	14976	2017	16.5 (0.8%)	6814	0.07
	lbm	2594	2834	2.9 (0.1%)	9841	0.01
	povray	12434	1644	14.9 (0.9%)	5111	0.05
	geo. mean of trns.			0.35%		0%

Table 2.1: Translation overheads of LnQ and QEMU. The number of guest basic blocks, and total running time and translation time in seconds are listed for each benchmarks. The numbers in parentheses are the percentages of translation time versus total running time.

by runtime optimizations, which we further investigate the effects of runtime optimizations in Section 2.3.4.

2.3.3.2 Translation Overhead

We now evaluate the translation overhead of LnQ. Table 2.1 compares the numbers of blocks translated in all benchmarks, the translation time, and the translation time percentage (in parentheses) of LnQ and QEMU. First, the translation time of LnQ is approximate 1.16% and 0.35% for integer and floating point benchmarks, respectively. It is worth for long running guest applications although the translation time of LnQ is slower than QEMU. For example, it takes 54.8 seconds to translate `perlbench` but saves 1819 seconds in running time compared to QEMU.

However, the translation overheads of LnQ may become intolerable for small jobs. For example, the translation overhead of 403.gcc benchmark is 28% of total running time in average. This is due to that there are large number of guest basic blocks, about 60,000, in 403.gcc benchmark, and the running time is not long enough to compensate the translation overhead. We have not addressed this problem in current LnQ framework. In future work, we may adopt two phase translation approach similar to IA-32 EL [16] into LnQ framework.

2.3.4 Optimization Effects of Runtime Optimization

We further investigate the effects of each runtime optimization techniques – Block Linking (BL), IBTC, and Shaow Stack (SS). We use LnQ without any runtime optimization as our baseline. We then evaluate the effect of each optimization by adding one optimization at a time in the order of Block Linking, IBTC, and Shadow Stack. Note that it is reasonable to present the optimization effects with BL, BL+IBTC, and BL+IBTC+SS combinations because these three optimizations aim at different cases as described in Section 2.2. The results are shown in Figure 2.10.

From Figure 2.10, the improvements of runtime optimizations are significant in integer benchmarks. We calculate the improvement by subtracting the speedup factor of each optimization combination with its previous one. The average improvements are 432%, 78%, and 236% for Block Linking, IBTC, and Shadow Stack, respectively. These significant improvements of integer benchmarks are because the integer benchmarks tend to have more complicated control flow than floating point benchmarks. The average improvements of runtime optimizations in floating

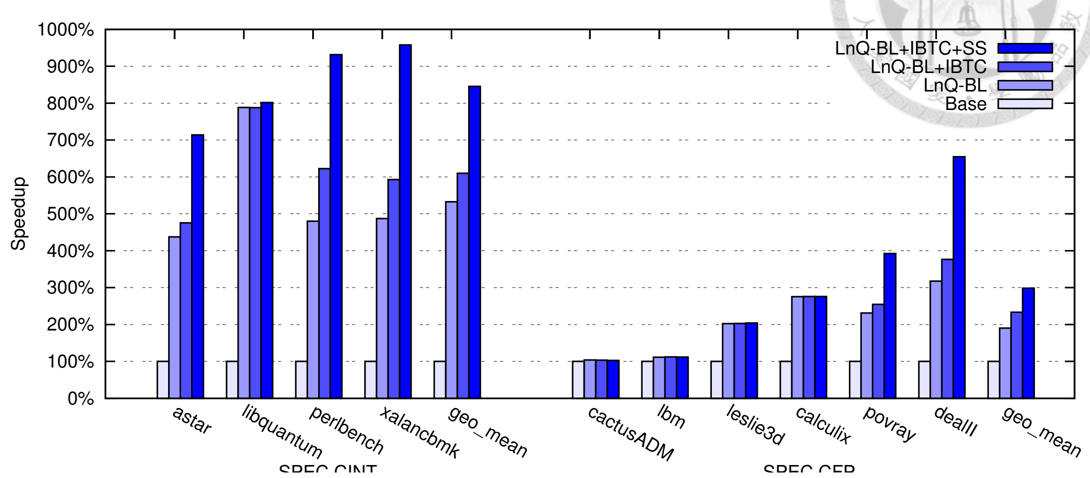


FIGURE 2.10: Performance of block linking, IBTC, and shaow stack

benchmarks are 90%, 43%, and 66% for Block Linking, IBTC, and Shadow Stack respectively.

Because the purpose of these optimizations is to reduce the frequency of going back to emulation engine, we further investigate the effects of each optimizations by showing the reduction of percentages of dispatch time. The results are shown in Figure 2.11.

In Figure 2.11, we can see most benchmarks are benefit from Block Linking optimization, which explains the 432% improvement of Block Linking. As for IBTC optimization, only benchmarks using indirect branches have performance gain from IBTC optimization. For example, `astar, libquantum, calculix, leslie3d, lbm` and `cactusADM` have less improvement from IBTC. Shadow Stack also improves most benchmarks.

Two additional notes, the first is although the dispatch time of some floating point

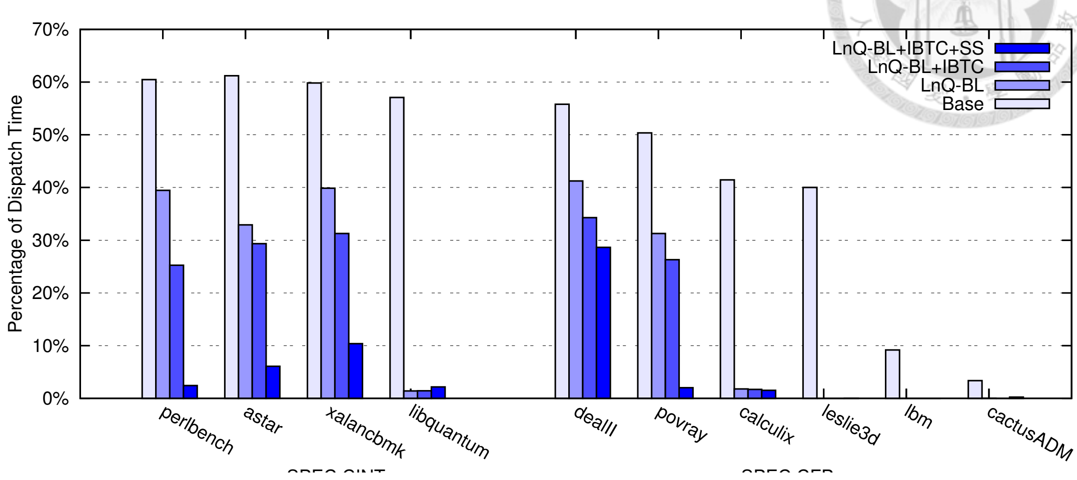


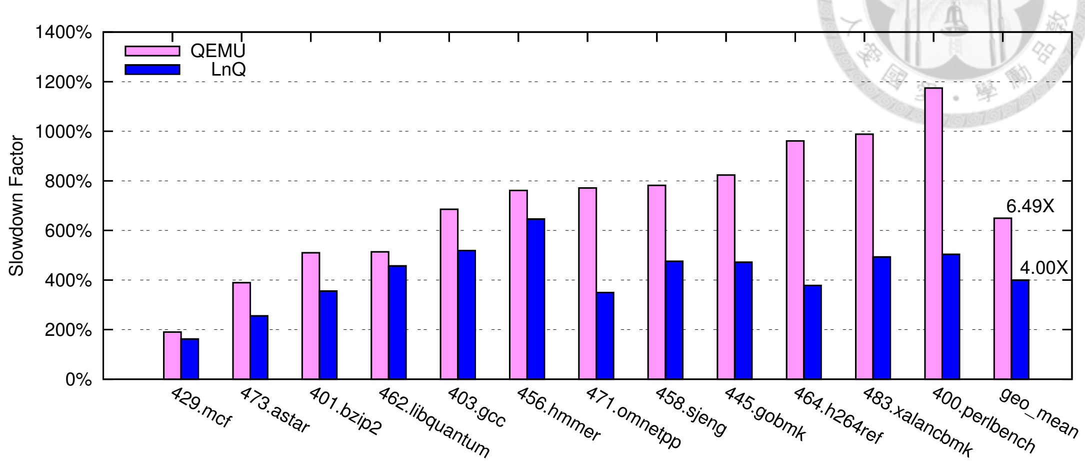
FIGURE 2.11: The reduction of percentage of dispatch time.

benchmarks are reduced dramatically, the improvements may not be that significant. This is because the dispatch time is not major part of total running time. For example, There are little dispatch time in `cactusADM` and `lbm` benchmarks. Section, the optimizations also improve the temporal locality of LnQ in that most program execution are in code cache.

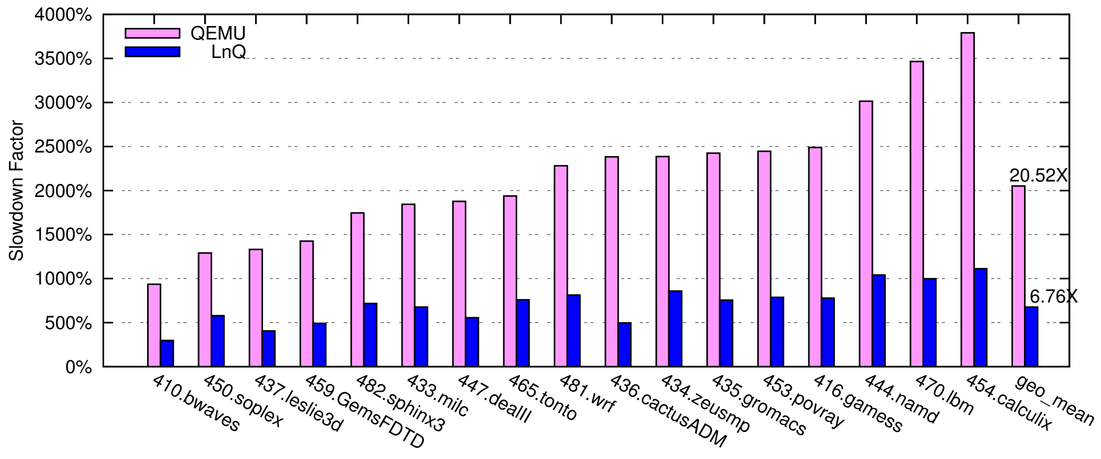
2.3.5 Slowdown of LnQ Compared to Native Run

In the last experiment, we examine the slowdown factors of LnQ compared to the native runs. For native runs, we compile SPEC benchmarks only with the "-O2" option without the "-m32" option. The results are shown in Figure 2.12.

From Figure 2.12, the geometric means of slowdown factors in SPEC CINT2006 are 4.00X and 6.49X in LnQ and QEMU, respectively. The slowdown factors increase in SPEC CFP2006, which are 6.76X and 20.52X in LnQ and QEMU. The slowdown factors of floating point benchmarks are larger than those of integer



2.12(A): Results of Integer Benchmarks



2.12(B): Results of Floating Point Benchmarks

FIGURE 2.12: The slowdown of LnQ and QEMU compared with native run.

benchmarks because the floating point operations in x86 architecture are stack-like operations, in which LnQ cannot perform register mapping for floating point registers. Results in Figure 2.12 shows that there is room for improvement fro dynamic binary translates.

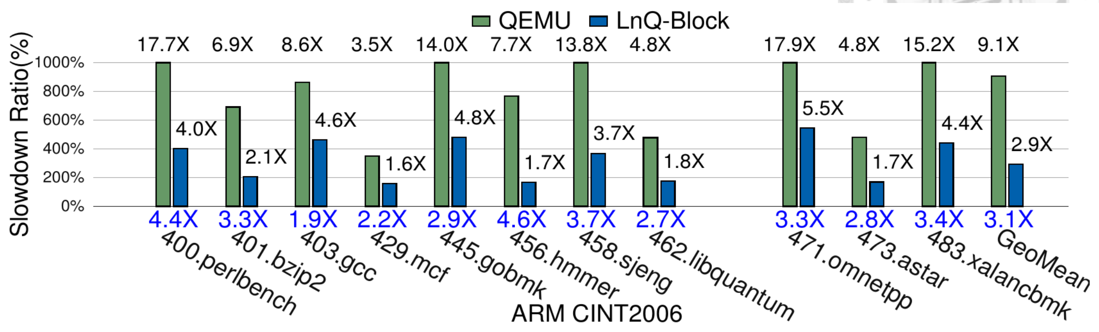


FIGURE 2.13: Performance of ARM-to-I32 LnQ

2.3.6 Performance of ARM-to-IA32 LnQ

In this section, we compare the performance of ARM-to-x86_64 LnQ to QEMU.Results are shown in Figure 2.13. On average, LnQ has 2.9X slowdown compared to native run. Compared to QEMU's 8.1X slowdown, LnQ outperforms QEMU about 3.1X.

2.3.7 Performance of IA32-to-ARM LnQ

In this section, we compare the performance of IA32-to-ARM LnQ to QEMU. We conduct experiments on the Odroid-XU board with ARM Cortex-A15 1.7GHz dual core CPU and 2GB memory. Results are shown in Figure 2.14. On average, LnQ has 3X slowdown compared to native run. Compared to QEMU's 6.7X slowdown, LnQ outperforms QEMU about 2.23X. The results also show LnQ do keep its high performance in different host CPUs.

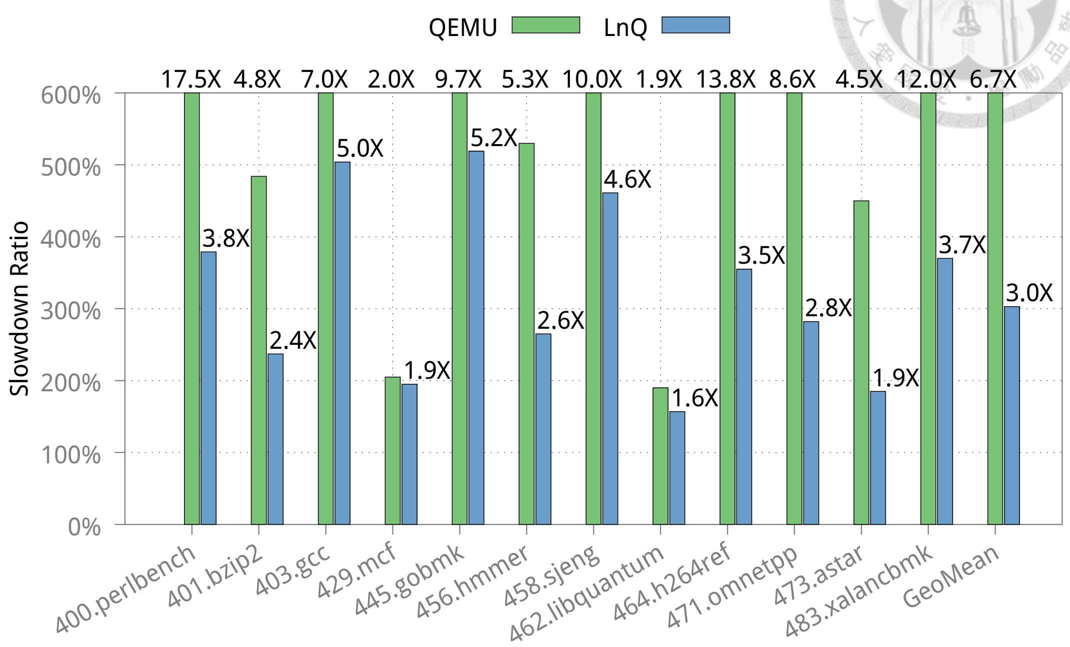


Figure 2.14: Performance of IA32-to-ARM LnQ

2.4 Concluding Remarks

This chapter introduces the LnQ (LLVM+QEMU) framework by which one can build high performance and retargetable dynamic binary translators with *existing* optimizers and code generation backends. In this chapter, we explain the design and implementation of LnQ framework. We use LLVM compiler infrastructure to design the IR library and the IR translator in the translation module. We also describe how to build IR library and the translation process of the IR translator. We also show how we perform register mapping and retargetable runtime optimizations in the IR translator.

We evaluate the performance of LnQ by building an x86-to-x86_64 dynamic binary translator with LnQ framework. The experimental results show 1.62X speedup for

SPEC CINT2006 and 3.02X speedup for SPEC CFP2006 in average compared to QEMU. The translation overhead of LnQ is 1.16% of total running time. Also, the ARM-to-x86_64 LnQ has 3.1X speedup compared to QEMU, and the IA32-to-ARM LnQ has 2.23X speedup compared to QEMU.

Chapter 3

The Early-Exit Guided Code Region Formation

3.1 Region-Based Multi-threaded Dynamic Binary Translator

In this section, we describe the design of our region-based multi-threaded dynamic binary translator, called LnQ [42]. We have implemented the EEG scheme in LnQ. LnQ uses QEMU [37] as the front-end emulation engine, and uses LLVM [38] compilation infrastructure to handle its back-end code optimization and target code generation. We implement our EEG scheme using this framework. Figure 3.1 shows the major components and the control flow of our region-based multi-threaded dynamic binary translator.

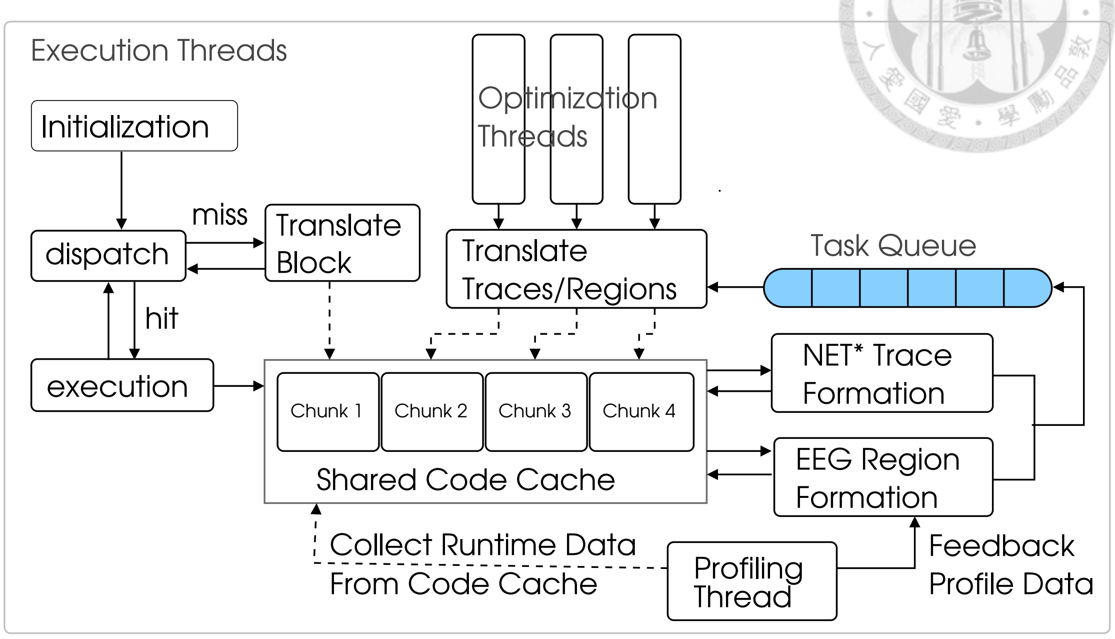


FIGURE 3.1: Control flow of execution threads and optimization threads

We use *code segments* to refer *basic blocks* and *traces/regions*, and use *code fragment* to refer a *translated* code segment by DBT. Therefore, there are *basic block fragments* and *trace/region fragments*. Each code fragment has a *prologue* to load the guest architecture states, such as the content of the guest registers, from the memory to the host registers before execution. Also, each code fragment has an epilogue to store modified machine states back to memory before leaving the code fragment. Each code fragment has its own register mapping decided by the LLVM register allocator.

LnQ uses *execution threads* and *optimization threads*. *Execution threads* are responsible for translating basic blocks and executing translated code fragments. That is, if an *execution thread* reaches a new guest basic block during execution, the execution thread generates a *basic block fragment* using LLVM. *Optimization* *threads* generate optimized traces and regions fragments also using LLVM. Execution threads compile blocks with “O0” optimization level to minimize compilation overhead. On the other hand, optimization threads compile traces and regions with “O2” to generate optimized code. All execution threads share one software code cache. As shown in Figure 3.1, we partition the code cache into *sections*, and each thread has its own section to store the translated code fragments so that threads can generate code concurrently.

The DBT system separates trace compilation from program execution. By running optimization threads concurrently on other cores, the execution threads are not disrupted. Execution threads may create region compilation tasks and send them to a *Task Queue* (see Figure 3.1) when traces or regions are formed as described in Section 3.2. We use a lock-free concurrent FIFO queue [43] to implement the *task queue* so that execution threads can insert trace/region compilation tasks into the queue while the optimization threads take those tasks from the queue without locks.

When an optimization thread generates a new trace or region, it dispatch execution threads to the newly generated code fragment by *atomically* patching jump instructions in the code cache. To do this in IA32, we need to align the patched instructions to 4-byte alignment, and use the self-branch technique mentioned in [44] to patch jumps atomically.

3.2 Early Exit Index and Early-Exit Guided Region Formation

In this section, we first describe the NET algorithm used in our system. We then define an *early exit index* to quantify how often early exits are taken in a trace. Finally we describe our early exit guided region formation technique.

3.2.1 Trace Formation Algorithm

We adopt a modified NET algorithm called *NET**, which is similar to [30], to builds traces. The difference is that NET* considers *all* basic blocks as *potential* trace head candidates, while NET only considers blocks which are targets of backward branches as *trace head* candidates in that they may form potential loops.

The NET* algorithm has two advantages. First, the NET algorithm [7] was designed for DBT systems in which a *single* DBT thread is responsible for both execution and trace building. To reduce the overhead of building traces, NET needs to be very selective in potential traces. In contrast, NET* can take advantage of modern multi-core platforms to offload the overhead of building traces. Hence, it can afford to try all basic blocks as potential trace heads.

Second, NET may not identify all loops by only considering targets of backward branches. By considering all basic blocks as possible trace heads, NET* can discover more hot traces than NET can. As reported in Section 3.3.1.1, NET* achieves 12% and 5% performance improvement on average over NET for SPEC CINT2006 and CFP2006 benchmarks, respectively.

Our NET* algorithm works as follows. We instrument software counters to record the number of times each block is executed. A block becomes a *trace head* when the number of times the block has been executed exceeds a threshold value. NET* forms a trace by appending blocks along the execution path until one of the following terminal conditions is met: (1) A branch to the trace head is taken, (2) The number of blocks exceeds a threshold, (3) The next block is the head of another trace, or (4) A guest system call instruction is encountered.

3.2.2 Early Exit Index

We first define an *early exit* of a trace. A trace can be a straight-line execution path or a cycle. If a trace is a straight-line path, then all exit edges along the path are early exits except the exit edge of the last basic block in the trace. If a trace is a cycle, all exit edges are early exit.

We define an *Early-Exit Index* (EEI) to measure the frequency of early exits taken in traces. More specifically, EEI is the number of early exits being taken for every million instructions executed in traces. It can be formally defined as in the following equation.

$$EEI = \frac{\sum_{i \in \Gamma} n_i \times \rho_i}{N}$$

where Γ is the set of traces, n_i is the number of times early exits being taken in trace i, ρ_i is the percentage of instructions executed in trace i, and N is the number of million instructions executed.

3.2.3 Early-Exit Guided Region Formation

In this section, we describe our proposed Early-Exit Guided (EEG) region formation scheme. It detects and merges regions that have frequently taken early exits. The key issues in EEG are (1) how to efficiently detect delinquent regions; and (2) when to merge them at runtime. We address them as follows.

The simplest approach to address the first issue is to instrument counters in all traces and regions. However, this approach is inefficient and may merge too many regions that are not frequently executed. Instead, we use a dynamic profiling approach with the help of on-chip hardware performance monitor (HPM) to select hot regions.

We create a profiling thread called *profiler* at the beginning of execution to perform dynamic profiling. The profiler collects program counters periodically for every million instructions retired. When a threshold number of samples are collected, the profiler accumulates the sample counts for each trace to determine the degree of *hotness* of each trace. The hotness of a trace is measured by the following equation.

$$H_T = \max\{\alpha, \beta\}$$

Here, α is the percentage of instructions executed in the trace during the *last* sampling period, and β is the percentage of instructions executed in the trace during the *entire* execution. Intuitively, α represents the hotness of the trace during the last period, and β represents the accumulated *hotness* during the entire execution. We choose the maximum of α and β as its hotness measure.

When the hotness of a trace exceeds a threshold, we start monitoring the trace by instrumenting counters to its early exits. Currently, we only monitor the early exits of conditional branches. If a counter exceeds a pre-defined threshold, it means the control leaves the region through the corresponding early exit very frequently. Then, we merge the monitored region with the target region of the early exit. We translate and optimize the merged region with our LLVM-based DBT, and replace the monitored region with the merged region.

We argue that the overhead of the instrumentation is negligible because early exits should be rarely taken. A frequently taken early exit would have triggered region formation when the counter exceeded the threshold.

3.2.4 Spill Index of a Region

The benefits of EEG region formation come from eliminating the overhead caused by frequently taken early exits, and potential optimization opportunities from a larger code region. Despite the fact that we can mostly eliminate the overhead of frequently taken early exits via region merging, we may not always have potential optimization opportunities from the merged region. In particular, if the quality of the translated code of a region is not good enough, it is not beneficial to merge such a region.

We define an index, called *Spill Index*, to assess the quality of the code generated by the LLVM compiler for a region formed by the EEG technique. A *spill instruction* is an instruction for load/store operations between registers and stack. The *Spill Index* is the percentage of spill instructions in the translated code fragment. When the Spill Index of a code fragment exceeds a threshold, that region should not be further merged because a high percentage of spill instructions often forestalls good performance due to improper register allocation of the LLVM compiler.

3.2.5 Region Versus Trace

By creating larger regions, we reduce the amount of specialization that the compiler can do for traces. As we know, the benefit of traces comes from the instruction scheduling within traces [45].

However, we need to limit the instruction scheduling optimizations when we compile traces in dynamic binary translation, because we have to rematerialize full guest state in case a hardware exception or a signal was raised.

The main advantage of EEG region formation is that it can improve DBT performance by removing transition overhead among traces, such as removing redundant loads/stores of guest state among traces.

We use Figure 3.2 as an example to illustrate our region formation strategy. Figure 3.2(a) is the control flow graph (CFG) of a hot region in a guest application. During execution, each block is first translated as shown in Figure 3.2(b). Then NET* forms three traces as in Figure 3.2(c). Trace A would be the first selected for early exit detection (see Figure 3.2(c)) since a loop is likely to become hot. Thus the early exit of A, marked by a dashed arrow from the trace started with

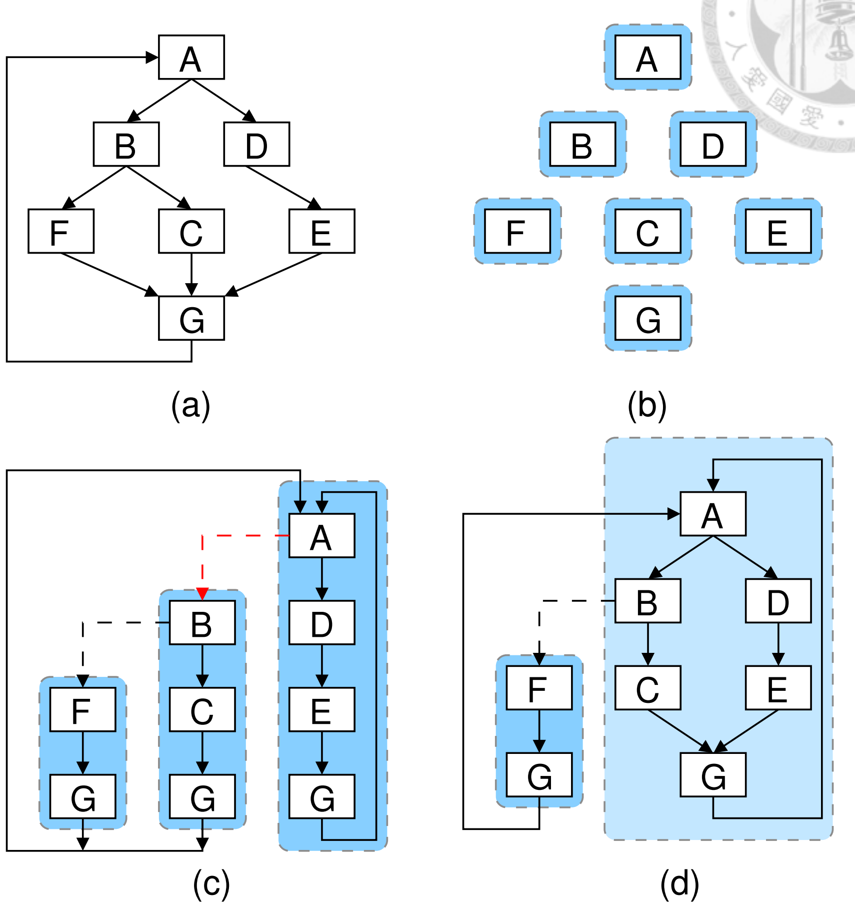


FIGURE 3.2: Illustration of region formation.

A (enclosed by the dotted rectangular) to the trace started with B, is monitored with an instrumented software counter.

We merge Trace A and Trace B to form a code region when the early exit is taken frequently. A code region, called Region A and is enclosed in the dotted rectangular in Figure 3.2(d), that consists of traces A and B is formed. After the code fragment of Region A is formed, we replace Trace A and Trace B with Region A so that Trace F now branches to Region A rather than to Trace A. Note that Region A will not be monitored because the spill index of Region A exceeds the threshold.

3.3 Performance Evaluation

In this section, we evaluate the performance of Early-Exit-Guided region formation algorithm in our LLVM-based parallel DBT systems. We start by describing our measurement methodology.

We evaluate the performance with SPEC CPU 2006 benchmarks on a 3.3GHz quad-core Intel Core i7 machine. The machine has 12 GB main memory and the operating system is 64-bit Gentoo Linux with kernel version 2.6.30. We use the *LnQ* [42] dynamic binary translation framework to build two translators which translate IA32 and ARM guest ISAs to x86_64 host ISA. For CFP2006 benchmarks, we only compile them into IA32 binaries because most CFP2006 benchmarks are written in Fortran and the ARM tool chain we use does not provide cross-compilation for Fortran. The result of ARM `464.h264ref` is not reported because the SPEC runspec tool reports a mis-match error even when it runs `464.h264ref` in a native ARM machine.

The benchmarks are compiled with GCC 4.3.4 for IA32 binaries and GCC 4.4.1 for ARM binaries. For all benchmarks, "-O2" flag is used. For IA32 benchmarks, we use "-m32" to generate IA32 binaries. For CFP2006, we use "-msse2 -mfpmath=sse" extra flags to generate SSE vector instructions. We use *runspec* script provided by SPEC to run benchmarks and report the median of 5 runs for all performance metrics.

We compare three region formation strategies in our experiments, which are *NET*, *NET** and *EEG* as described in Section 3.2. In EEG strategy, we first use NET* to select traces, and use EEG to merge traces into regions. We set block count

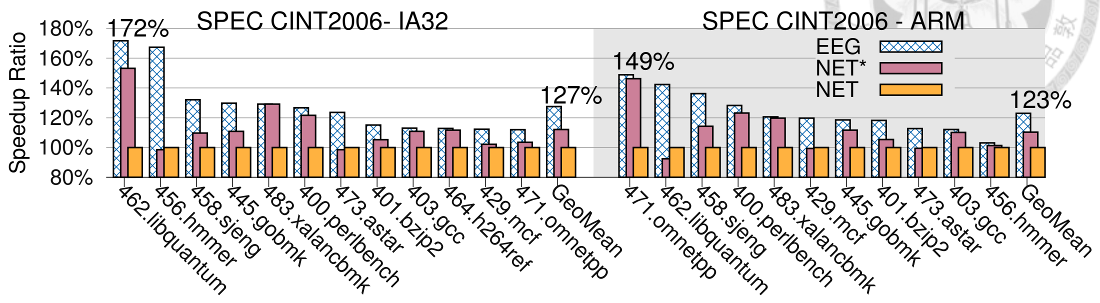


FIGURE 3.3: Performance results of NET* and EEG compared to NET in IA32 and ARM SPEC CINT2006.

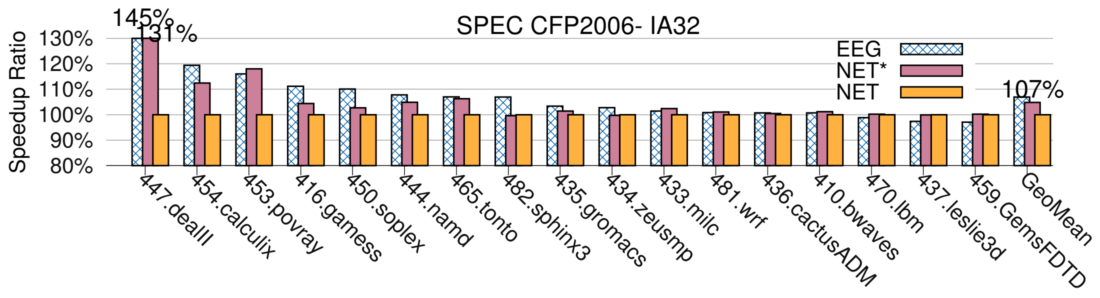


FIGURE 3.4: Performance results of NET* and EEG compared to NET in SPEC CFP2006.

threshold to 50 and allow at most 16 blocks in a trace. For EEG strategy, the threshold of spill index is set to 15%, i.e. regions cannot be further merged when the percentage of spill instructions in the translated fragment exceeds 15%.

We use Perfmon2 [46] for hardware-assisted dynamic profiling to collect runtime information for every one million retired instructions. The early exit threshold is set to 1000 and we use two optimization threads to compile traces and regions in all experiments.

3.3.1 Performance Results of SPEC CPU2006

The performance results of SPEC CPU2006 are shown in Figure 3.3 and Figure 3.4. For clearness of presentation, the benchmarks in both figures are sorted in decreasing order of speedup ratio so that it is easier to see the maximum, the minimum, and the geometric average of the results. We explain the results in the following sections.

3.3.1.1 Performance of NET*

The performance of NET* algorithm compared to NET in SPEC CINT2006 benchmarks is shown as red bars in Figure 3.3. For CINT2006 benchmarks, NET* achieves an average improvement of 12% and 10% for the IA32 and ARM benchmarks, respectively, with up to 53% and 46% for IA32 `456.hmmer` and ARM `471.omnetpp`. The results show that NET* discovers more hot traces than NET does by considering all blocks as possible trace heads, and our DBTs do not incur significant overhead because the compilation overhead is offloaded to optimization threads.

We notice that only ARM `462.libquantum` has 8% slowdown. We compare traces generated by the two algorithms and show the difference, in Figure 3.5, among traces generated by NET and NET* for a hot loop in function `quantum_toffoli` of `462.libquantum`.

As shown in Figure 3.5 (a) and 3.5 (b), both NET and NET* have the same trace `T-d10c`, but NET* splits trace `T-d094` of NET into `T-d094` and `T-d0b4` because

NET* generates `T-d0b4` before `T-d094`. The transition between traces `T-d094` and `T-d0b4` in NET* results in 8% slowdown compared to NET.

However, both NET and NET* have the delinquent trace `T-d10c` with frequently taken early exit to `T-d094` due to an unbiased branch in block `d10c`. In the next section, we show that EEG can merge the delinquent trace `T-d10c` into one region as shown in Figure 3.5(c) and improves the performance of NET* by 54%.

Figure 3.4 shows the speedup ratio of NET* algorithm with NET as baseline performance for the SPEC CFP 2006 benchmarks. NET* achieves significant improvement only in 447.dealII, 453.povray, and 454.calculix (31%, 18% and 12% respectively), and it gains 4.9% improvement on average in CFP2006 benchmarks. Most CFP2006 benchmarks spend their time in small number of hot loops, which can all be identified by NET and NET*. Thus, there is little difference between traces of NET and NET* in these benchmarks.

3.3.1.2 Performance of EEG Region Formation

The performance of EEG compared to NET in SPEC CINT2006 benchmarks is shown in Figure 3.3. For CINT 2006 benchmarks, EEG achieves an average improvement of 27.5% and 23% for the IA32 and ARM benchmarks, respectively, with up to 71.7% and 49% for IA32 `456.hmmer` and ARM `471.omnetpp`. Merging traces can reduce the prologue and epilogue code executed hence the transition overhead among different traces/regions are reduced. As we will see in Section 3.3.3, the execution with EEG has less memory and branch operations compared to NET.

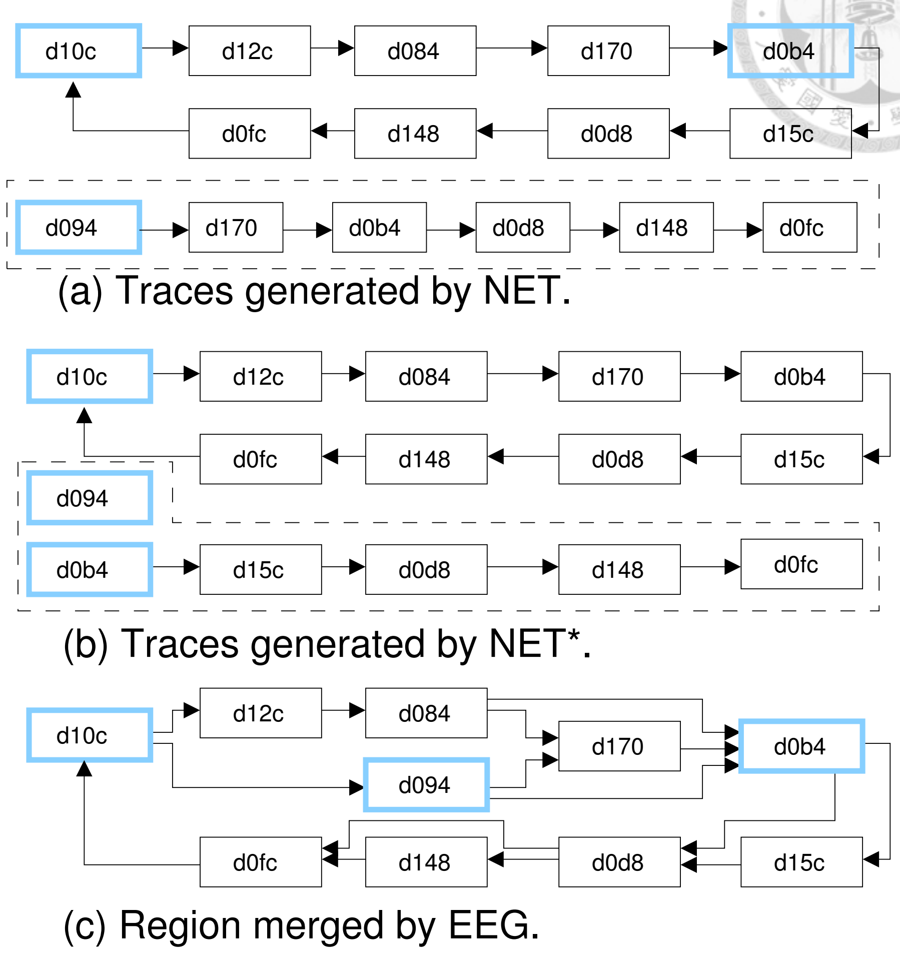


FIGURE 3.5: Traces/regions generated by NET, NET* and EEG for a loop in function `quantum_toffoli` of ARM `462.libquantum`.

We now take a closer look at IA32 `456.hmmer and ARM 462.libquantum` to give more insight of the benefit of EEG. In `456.hmmer`, the hottest function is `P7Viterbi`, which updates global variables according to different conditions in a performance critical `for`-loop. NET* splits this loop into four traces as shown in Figure 1.1(a).

Consequently, the transition among four traces results in significant overhead. Through early exit detection, EEG merges four traces into one region containing the loop as shown in Figure 1.1(b). The merged region achieves 70% performance improvement because of the elimination of the transition overhead among traces.

For `462.libquantum`, NET* splits a for-loop of function `quantum_toffoli` into three traces as shown in Figure 3.5(b). As described in the previous section, trace `T-d10c` is a delinquent trace with a frequently taken early exit to trace T-d094 due to an unbiased branch in block `d10c`. EEG improves performance by 54% by merging the two traces into one region as shown in Figure 3.5(c).

As shown in Figure 3.4, EEG improves NET* by 4.8% to 7% on CFP2006. The improvement is minor because there are few early exits in these floating point benchmarks. In the next section, we measure the early exit index and show the relation between the number of early exits and the performance improvement.

We also observe that EEG loses 2.7% and 2.9% performance compared to NET in 437.leslie3d, and 459.GemsFDTD respectively. In 437.leslie3d, the time is spent in a small number of nested loops in the procedure `EXTRAPI` of file `tml.f`. The regions generated by EEG contain nested loops while each trace generated by NET contains only the innermost loop. Therefore, in 437.leslie3d and 459.GemsFDTD, the translated code for traces is better than translated code for regions. As a result, EEG loses about 2.7% performance compared to NET.

3.3.2 Early Exit Index

In this section, we measure the Early Exit Index (EEI) of benchmarks with the NET* strategy. We insert counters at each side exit to collect the number of early exits taken in each trace, and we measure the execution frequency of traces by sampling program counters per one million retired instructions. We calculate EEI

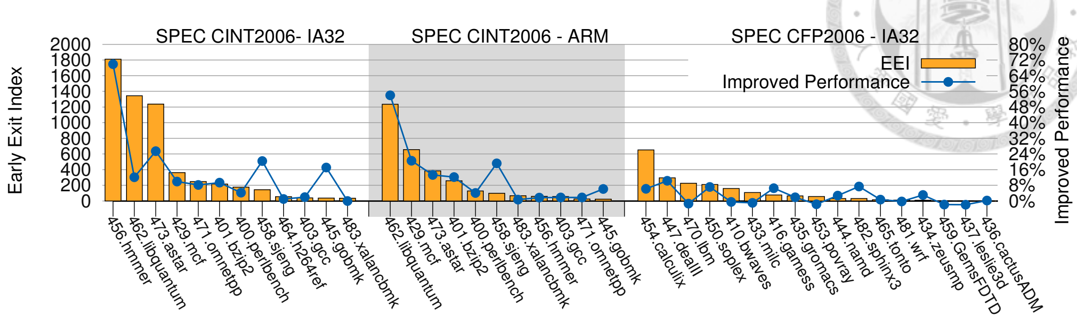


FIGURE 3.6: Measured Early Exit Index in NET* and the performance improvement of EEG.

with the collected numbers as described in Section 3.2.2. The results are shown in Figure 3.6. The Y-axis on the left side shows the measured early exit indices; the Y-axis on the right side shows the performance improvement of EEG compared to NET*.

In Figure 3.6, we observe that integer benchmarks are likely to have high EEI values. For example, 65% of CINT2006 benchmarks have EEI values larger than 100, which means there are over 100 early exits per million instructions in those benchmarks in NET*. CINT 2006 benchmarks also show positive correlation between early exit index and performance improvement. The correlation coefficient of IA32 CINT2006 and ARM CINT2006 are 0.78 and 0.93.

For CFP2006 benchmarks, all the EEI values are relatively small compared to those in integer benchmarks. Only 35% of the benchmarks have EEI values larger than 100. The correlation coefficient of early exit index is 0.43 in CFP2006. Small EEI values are due to the fact that floating point benchmarks usually spend most of their time in simple loops with fewer early exits. We also notice that some benchmarks with small EEI values achieve good performance improvements, such as `445.sjeng` and `445.gobmk`, which improve 20% and 17%, with EEI values as

IA32 CINT2006	Improved Ratio	Reduced Instructions or Misses		
		MemInst	BrInst	L1 ICache Misses
456.hmmer	69.9%	52.8%	36.9%	31.0%
473.astar	25.5%	35.4%	20.4%	3.3%
458.sjeng	20.4%	29.9%	17.0%	43.7%
445.gobmk	17.1%	18.2%	7.6%	29.2%
462.libquantum	12.1%	33.6%	9.3%	0.7%
429.mcf	9.9%	33.9%	14.7%	18.2%
401.bzip2	9.3%	18.8%	11.8%	19.0%
471.omnetpp	8.2%	17.2%	7.1%	46.2%
400.perlbench	4.2%	9.5%	4.1%	15.1%
403.gcc	1.9%	5.6%	1.8%	9.8%
464.h264ref	1.0%	1.3%	2.5%	18.6%
483.xalancbmk	0.0%	6.8%	-3.5%	3.0%

ARM CINT2006	Improved Ratio	Reduced Instructions or Misses		
		MemInst	BrInst	L1 ICache Misses
462.libquantum	54.0%	69.0%	15.8%	-1.3%
429.mcf	20.5%	45.5%	17.3%	59.4%
458.sjeng	19.2%	17.9%	11.5%	35.5%
473.astar	13.3%	21.1%	10.7%	3.7%
401.bzip2	12.2%	26.8%	13.9%	20.4%
445.gobmk	6.1%	6.5%	5.3%	17.6%
400.perlbench	4.1%	3.3%	6.3%	13.2%
471.omnetpp	1.8%	-0.8%	1.6%	7.8%
456.hmmer	1.7%	0.2%	0.7%	59.7%
483.xalancbmk	0.7%	1.5%	7.2%	2.9%
403.gcc	-0.6%	0.1%	2.4%	4.7%

TABLE 3.1: Reduced memory/branch instructions and cache misses of EEG for CINT2006 benchmarks.

low as 143 and 36 respectively. In the next section, we collect performance profiles to further analyze the sources of improvement.

3.3.3 Performance Profiles of EEG

In this section, we collect the number of *memory*, *branch* instructions and the *L1 instruction cache misses* of NET* and EEG through hardware performance monitoring. We calculate the percentage of reduced memory/branch operations and cache misses in EEG compared to NET*. We focus on the profiles of CINT2006, which are shown in Table 3.1.

As shown in Table 3.1, benchmarks with large improvement tend to have high percentage of reduced operations or L1 instruction cache misses. For example, IA32 `456.hmmer` reduces 52.8%, 36.9% and 31% of memory, branch instructions and L1 i-cache misses, and achieves 70% improvement over NET*. There are also significant percentage of reduced instructions and misses in `458.sjeng` and `445.gobmk`, which contributes to the improvement of these two benchmarks. The profiling data show that EEG can not only reduce the memory and branch instructions but also reduces L1 instruction cache misses by merging delinquent traces into regions.

3.3.4 Effect of The Threshold of Spill Index

In this section, we study the effect of the threshold of spill index, described in Section 3.2.4, on the performance of EEG. In both Figure 3.7 and Figure 3.8, the X-axis of each plot is the improvement ratio using the performance of 5% threshold as the baseline, and the Y-axis is the threshold of spill indices ranged from 5% to 25%. As shown in Figure 3.7, the performance of EEG is less sensitive to the threshold of spill index for IA32 benchmarks except `471.omnetpp`. The results show that the register pressure is not a problem in the region fragments of

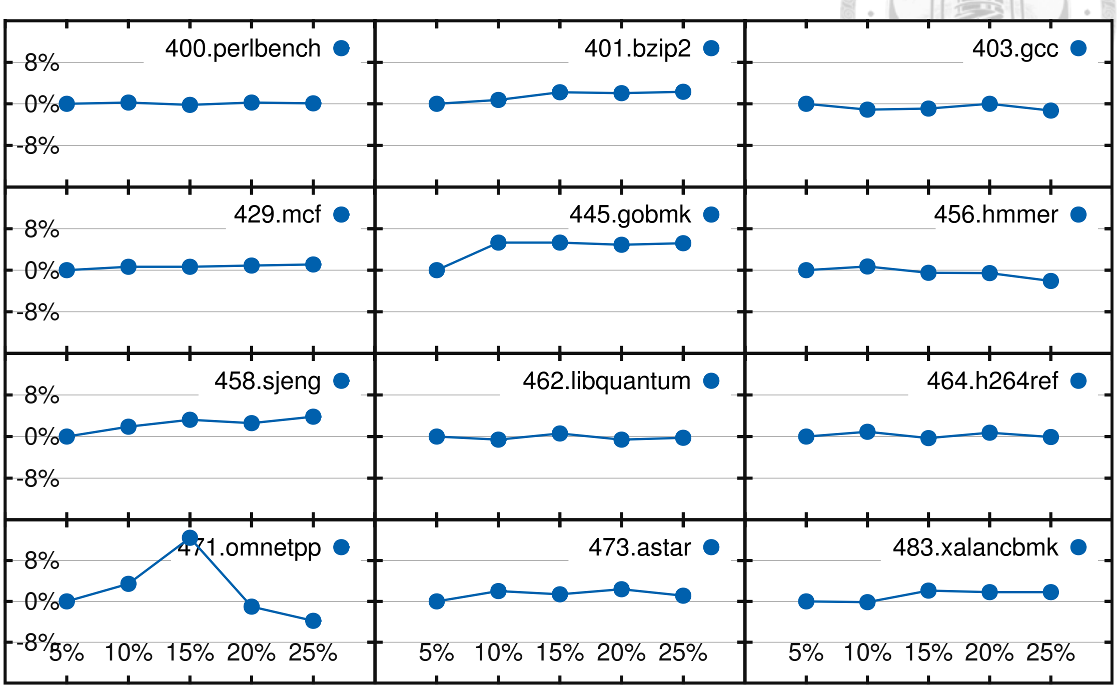


FIGURE 3.7: Effect of spill index of IA32 CINT2006.

IA32 benchmarks because the IA32 guest architecture has only 8 general purpose registers while there are 16 registers on x86_64 host architecture.

For `471.omnetpp`, the performance degrades by 13.5% when the threshold changes from 15% to 20%. The reason is that when threshold changes from 15% to 20%, the spill index of the hottest fragment changes from 18% to 36% because that fragment merges one more region and its CFG becomes complex when threshold is set to 20%. As a result, the extra spill instructions degrade the performance of `471.omnetpp`.

For ARM benchmarks, the performance of EEG is more sensitive to the threshold of spill index as shown in Figure 3.8. This is because there are 16 general purpose registers in ARM guest architecture, and register pressure becomes a problem

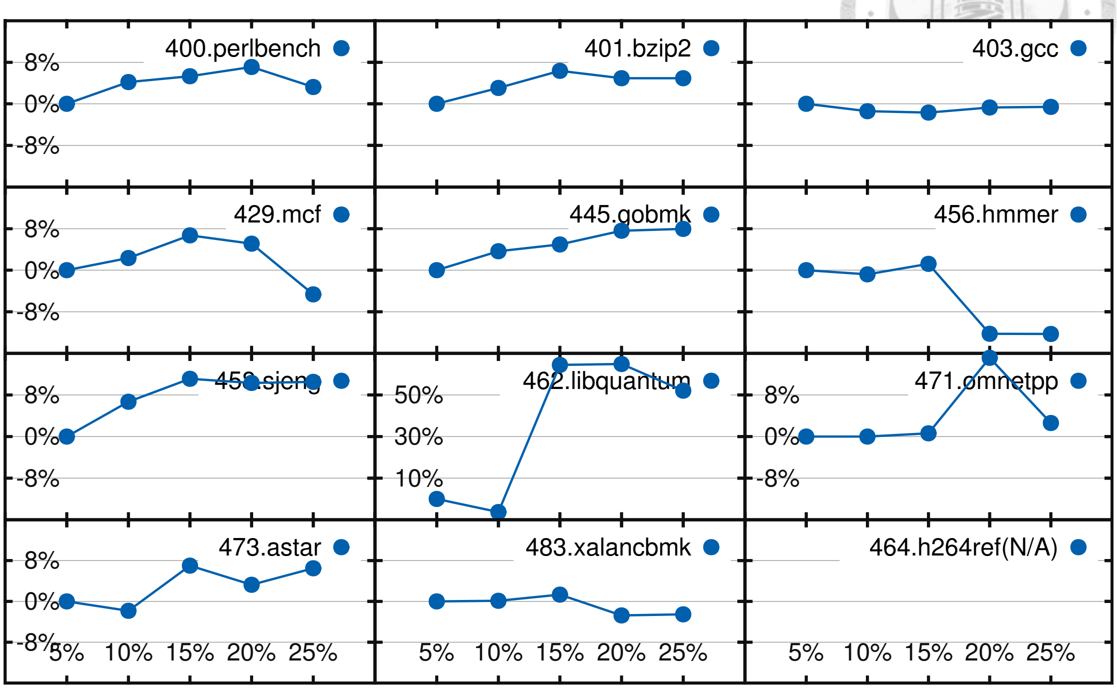


FIGURE 3.8: Effect of spill index of ARM CINT2006

when translating ARM instructions to x86_64 instructions. Consequently, if we allow regions with high spill indices, i.e., high percentage of spill code in the translated code, to be merged, the performance tends to degrade. For example, in ARM `456.hmmer`, a 12% degradation is observed when the threshold of spill index increases from 15% to 20%.

3.3.5 Statistics of Selected Traces and Regions

Table 3.2 and Table 3.4 shows the statistics of selected regions in NET, NET* and EEG for CINT2006. First, the *number of traces* in NET* increase by 54% and 59% on average compared to NET for IA32 and ARM benchmarks respectively. The *average numbers of blocks per trace* are similar in NET and NET*.

IA32 CINT2006	NET		NET*		EEG	
	#Traces	Avg.Blks	#Traces	Avg.Blks	#Regions	Avg.Blks
400.perlbench	6646	4.1	8966	5.9	13.5	1.6
401.bzip2	583	3.8	894	5.0	14.4	1.8
403.gcc	23058	4.0	34019	3.9	12.2	1.6
429.mcf	239	4.7	605	3.5	15.1	2.7
445.gobmk	9468	2.9	10961	3.9	13.3	1.7
456.hmmer	424	3.8	687	3.6	25.5	4.5
458.sjeng	1216	3.3	1749	4.7	18.9	2.5
462.libquantum	200	2.5	326	2.2	8.4	1.8
464.h264ref	2434	3.3	3974	4.3	11.9	1.8
471.omnetpp	2918	5.0	4859	5.5	12.1	1.4
473.astar	613	5.2	942	4.5	24.3	5.3
483.xalancbmk	4453	5.9	8355	4.4	11.4	1.3
Geometric Mean		3.9		4.2		2.1

TABLE 3.2: Number of traces/Regions and average blocks in NET* and EEG of IA32 CINT2006.

IA32 CINT2006	Merges		%Time Spent in	
	Avg.	Max	Trace	Region
400.perlbench	1627	16	45.3%	51.6%
401.bzip2	206	6	19.2%	79.4%
403.gcc	2728	17	27.1%	37.0%
429.mcf	83	5	1.2%	95.3%
445.gobmk	2258	20	20.9%	71.6%
456.hmmer	61	6	1.5%	97.8%
458.sjeng	764	43	15.2%	84.5%
462.libquantum	20	4	17.6%	82.2%
464.h264ref	781	11	22.0%	73.3%
471.omnetpp	345	11	29.0%	69.6%
473.astar	185	7	2.3%	96.8%
483.xalancbmk	538	12	37.0%	59.1%
Geometric Mean			12.4%	72.3%

TABLE 3.3: Number of merges and percentage of execution time of regions in EEG of IA32 CINT2006.

ARM CINT2006	NET		NET*		EEG	
	#Traces	Avg.Blks	#Traces	Avg.Blks	#Regions	Avg.Blks
400.perlbench	7839	5.1	10438	5.1	1860	12.8
401.bzip2	672	4.7	1125	4.7	205	16.4
403.gcc	24703	3.7	36710	3.7	2595	12.1
429.mcf	362	3.6	726	3.6	125	22.3
445.gobmk	14175	3.5	16587	3.5	3071	11.9
456.hmmer	847	4.8	1378	4.8	58	10.8
458.sjeng	1299	4.6	1811	4.6	760	14.7
462.libquantum	606	8.6	951	8.6	85	12.0
471.omnetpp	4584	3.5	7163	5.1	364	12.2
473.astar	959	3.9	1432	4.6	180	13.6
483.xalancbmk	4844	5.1	8690	3.9	559	11.9
Geometric Mean		4.1		4.6		13.4

TABLE 3.4: Number of traces/Regions and average blocks in NET* and EEG of ARM CINT2006.

IA32 CINT2006	Merges		%Time Spent in	
	Avg.	Max	Trace	Region
400.perlbench	1.6	12	32.9%	62.2%
401.bzip2	1.7	5	22.1%	75.0%
403.gcc	1.5	10	13.1%	32.5%
429.mcf	2.9	6	1.2 %	96.8%
445.gobmk	1.6	19	28.8%	60.9%
456.hmmer	1.4	7	52.9%	46.7%
458.sjeng	2.5	35	19.8%	79.6%
462.libquantum	2.4	11	41.0%	58.4%
471.omnetpp	1.4	12	57.0%	41.6%
473.astar	1.6	10	25.6%	72.8%
483.xalancbmk	1.3	8	49.1%	45.9%
Geometric Mean	1.7		23.0%	58.4%

TABLE 3.5: Number of merges and percentage of execution time of regions in EEG of ARM CINT2006.

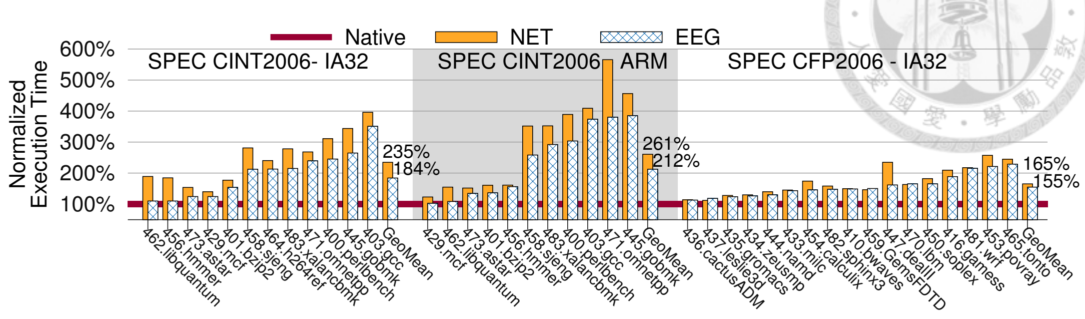


FIGURE 3.9: Normalized execution time of EEG compared to native execution.

13.6% and 11.5% of traces in NET* are merged into regions by EEG for the IA32 and ARM benchmarks respectively, which indicates that our HPM-based region formation approach described in Section 3.2.3 can effectively select hot traces to be merged. The average numbers of blocks per region are 14.4 and 13.4 for the IA32 and ARM benchmarks respectively, which are 3.4X and 2.9X larger than the traces generated by NET*.

We also compute the number of merges in EEG as shown in the first two columns of Table 3.3 and Table 3.5. There are 2.1 and 1.7 merges per region on average in IA32 and ARM benchmarks, which indicates that most regions become stable after few number of merges. The last two columns of Table 3.3 and Table 3.5 are percentage of execution time spent in traces and regions. On average, our DBTs spend 72.3% and 58.4% execution time in regions for the IA32 and ARM benchmarks respectively.

3.3.6 Performance Comparison to Native Execution

In this section, we compare the performance of EEG to native execution whose executables are compiled into x86_64 instructions. As shown in Figure 3.9, EEG achieves 1.84X and 2.12X slowdown compared to native execution in IA32 and ARM CINT2006 benchmarks, as opposed to 2.35X and 2.61X slowdown using the NET scheme. For IA32 CFP2006 benchmarks, EEG can achieve 1.55X slowdown compared to native execution. EEG has less slowdown in floating benchmarks because the SSE vector instructions can be translated into LLVM vector IRs, which can be efficiently translated into vector instructions on the host machine.

3.3.7 Performance of EEG on IA32-to-ARM LnQ

In this section, we measure the performance of EEG on IA32-to-ARM LnQ. We conduct experiments on the Odroid-XU board with ARM Cortex-A15 1.7GHz dual core CPU and 2GB memory. As shown in Figure 3.10, the Region bars are the results of EEG on IA32-to-ARM LnQ DBT. On average, EEG achieves 1.97X slowdown compared to the native runs, while the NET* gets 2.21X slowdown. Overall, EEG outperforms NET* 12% on IA32-to-ARM LnQ DBT.

3.4 Concluding Remarks

We have identified and quantified the delinquent trace problem in the popular Next-Executing-Tail (NET) trace formation algorithm. Delinquent traces contain frequently taken early exits which cause significant overhead. Motivated by this

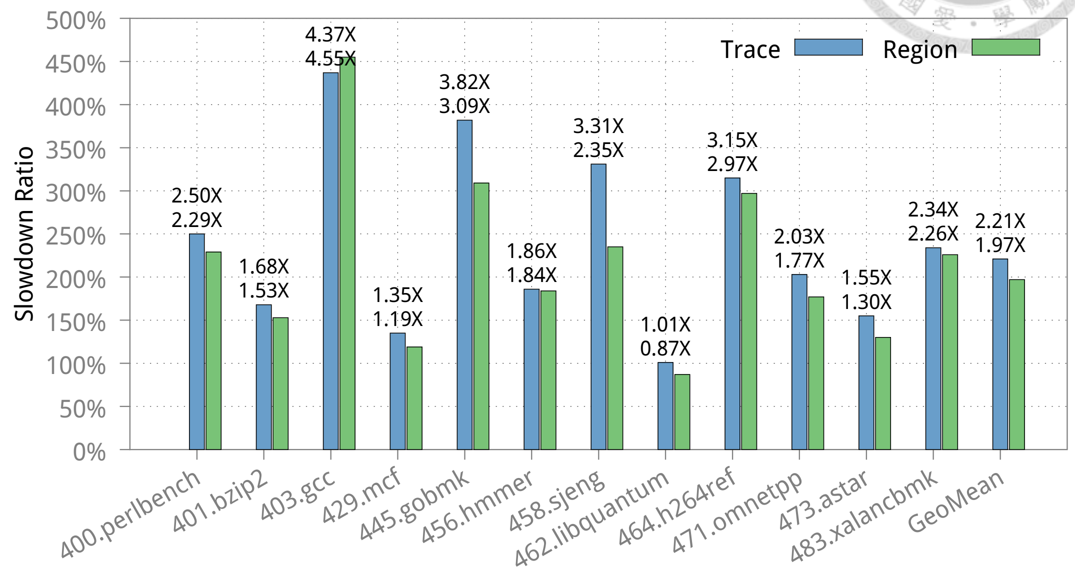


FIGURE 3.10: Normalized execution time of EEG compared to native execution on ARM host.

problem, we develop a light-weight region formation strategy called Early-Exit Guided region formation (EEG) to improve the performance of NET by merging delinquent traces into larger code regions. The EEG algorithm is implemented in two LLVM-based parallel dynamic binary translators (DBT), the IA32-to-x86_64 and ARM-to-x86_64 DBTs.

Experiment results show that EEG achieves performance improvement of up to 72% (27% on average), and up to 49% (23% on average) in IA32 and ARM SPEC CINT2006 benchmarks respectively. The profiling results show that EEG can reduce memory and branches instructions by up to 53% and 37% respectively because the transition overhead among traces is eliminated by merging delinquent traces. It also reduces the L1 instruction cache misses by up to 43.7% in CINT2006 benchmarks. We also implement EEG on IA32-to-ARM LnQ DBT, and EEG outperforms NET* 12% on ARM host.

Chapter 4

Trace-Guided Procedure-Based Code Region Formation

4.1 Architecture of Dynamic Binary Translator

Without lose of generality we assume that our guest program contains symbol table information. If the symbol table was "stripped" off from a guest program, we can always use *unstripped* [47] to recover the symbol table information. From the symbol table information we will be able to determine the boundary addresses of functions in the guest program.

This chapter uses *blocks*, *traces*, and *procedures* to refer to blocks, traces and procedures of the guest program. A *block* is a consecutive sequence of guest assembly instructions that spans from its starting address to the first control transfer instruction encountered. A *trace* is a set of blocks that form either a path or a cycle.

A *procedure* is a segment of assembly instructions that correspond to a procedure in the guest program. The address of this segment can be found in the symbol table of the guest program. We also use *call blocks* and *return blocks* as blocks which have calls and returns as their last instruction.

A *fragment* is a code segment translated by our binary translator, therefore we have *block* fragment, *trace* fragment, and *procedure* fragment, which are the translation results of a block, a trace, or a procedure.

We use *LnQ* [42] dynamic binary translation framework developed by Hsu et al. to build our procedure-based multi-threaded dynamic binary translation system. LnQ uses LLVM [38] compilation infrastructure to build the backend just-in-time compilers, and uses QEMU [37] as the emulation engine. We inherit the retargetability of LnQ, and extend the framework to accommodate a optimization thread pool. We also develop a trace-guided procedure optimizations on top of this framework.

Our dynamic binary translation system has execution threads and optimization threads. Initially there is only one execution thread. When the main execution thread encounters thread creating system call like vfork in linux, new execution threads are created so that one execution thread emulates one guest application thread.

Both execution thread and optimization thread can call LLVM JIT compiler functions. The execution thread sets JIT compilation flag to "O0" in order to enable fast instruction selection ("enable-fast-isel" option in LLVM) and minimize compilation overhead. The optimization thread sets JIT compiler flag to "O2" in order to get better code quality for traces and procedures.

A software code cache holds translated fragments. The software code cache consists of fix-sized chunks of 4MB. Both JIT compilers generate codes into their own chunks, but the execution threads can transfer among *all* chunks in the software code cache. When JIT compilers run out of chunks, they dynamically allocate more chunks from host operating system.

Each fragment has prologues to load the guest architecture states, such as guest CPU registers, to host registers before execution. Also, we have epilogues to store modified dirty states into memory before leaving fragments. Note that prologues and epilogues are necessary because, for the sake of retargetability, we do not specific register binding between guest architecture states and host architecture states. Each fragment has its own register binding scheme decided by the LLVM register allocator. We use the LLVM default linear register allocator.

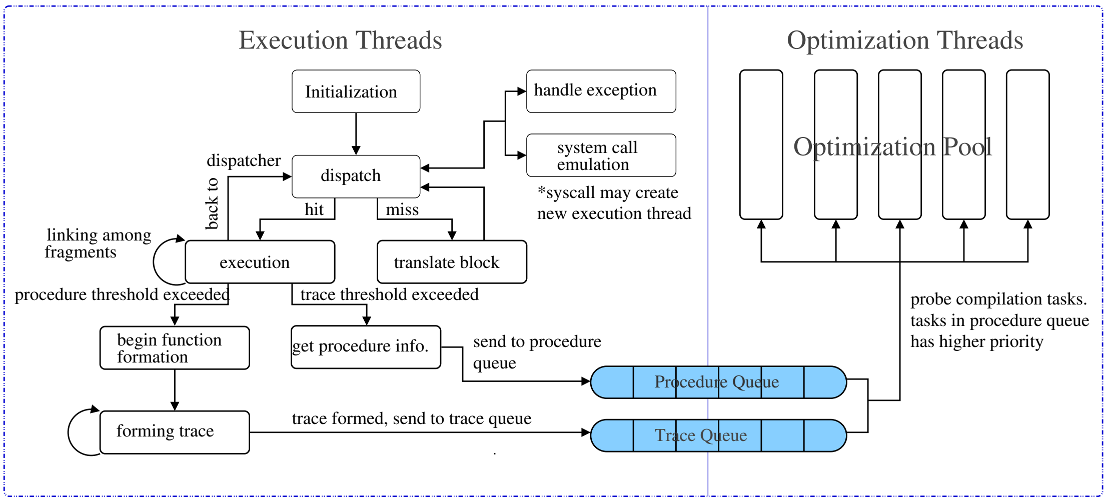


FIGURE 4.1: Control flow of main thread and optimization threads

We use two lock-free concurrent FIFO queues [43], *procedure queue* and *trace queue* as communication channels between execution threads and optimization threads. When an execution thread wants to build a trace it insert a request for trace compilation into the trace queue. Similarly if an execution thread has identified a hot procedure, it insert a request for procedure compilation into the procedure queue. Every millisecond an optimization thread probes the procedure queue then the trace queue for requests. The optimization probes the procedure queue first in order to give those procedure compilation requests a higher priority than those trace compilation requests.

We describe the control flow of execution threads and optimization threads as follow. The execution thread first looks for the translated fragment of the current guest block in the code cache. If the corresponding fragment is found, the execution thread executes the fragments in the code cache. If the execution thread cannot find the corresponding fragment, it starts translating the current guest program block. If this block happens to be a candidate of starting point of a trace or an entry block of a procedure, we insert profiling code to record the execution count of this blocks. If the execution count of the current guest block exceeds a predefined threshold, a trace/procedure compilation task is initiated and a request is inserted into the trace/procedure queue. Please refer to Figure 4.1 for an illustration.

We must make two adjustments after adding a new fragment. First, we use a *mapping table* to keep track of the mappings between guest code segments and translated fragments. When a new trace or procedure fragment is generated, we must add an entry into the mapping table to map the starting address of the guest trace/procedure to the newly generated fragment. Second, we need to update

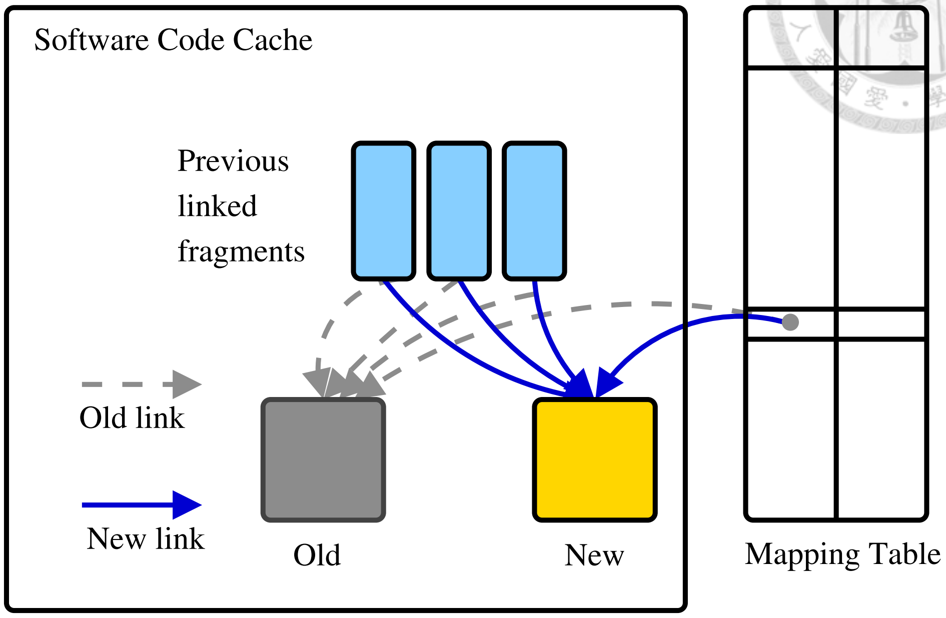


FIGURE 4.2: When a new fragment is generated, we update the mapping table and the links between the new and old fragments.

the control flow in other fragments so that they can reach the new fragment if necessary. We apply the *block linking* technique [6] to link the new fragment with other existing fragment in the code cache. Please refer to Figure 4.2 as an illustration.

4.2 Procedure Compilation in Dynamic Binary Translation

Profiling code is inserted in the entry block of a procedure to count the number of times the procedure is invoked. When the counter exceeds a predefined threshold, the execution thread inserts a request for procedure compilation into the procedure queue. We compile a procedure as follows. Since we know the boundary of the procedure, we first construct a control flow graph for the procedure. For each block in the procedure, we create a block of LLVM IRs as described in [42]. We then encapsulate these blocks with an LLVM function, and generate and optimize the procedure. Several complications arise when we compile a procedure in during dynamic binary translation – *call-return*, *code discovery*, and *targets of indirect branch*.

4.2.1 Call-Return Problem

Before describing the Call-Return problem, we first describe how the execution thread goes through code blocks in our dynamic binary translation. First, in the dispatch mode, we use a mapping table to map a starting guest address to a fragment in the code cache, so that the control can reach a fragment if it is there. Second, a fragment is transferred to another fragment via a direct branch on a host machine even in a call block (i.e. the last instruction of the guest block is a call) because the return from the call will be to a different guest block, i.e. a return block (it starts at the return address of a call block). This technique is used in QEMU [37]. Since we use a branch rather than a call on the host machine, the Call-Return problem arises.

We describe the Call-Return problem as follows.The call instruction of a call block will transfer the control outside a procedure fragment unless the call is recursive. Note that we use a branch instruction, instead of a call instruction, on the host machine to transfer execution to the called procedure. Hence, we need to redirect the execution back to the guest return block using another branch instruction after the called procedure finishes. However, we cannot jump into the middle of a procedure. It will violate the protocol of a procedure call. This is where the Call-Return problem arises.

To resolve this problem, we first need to make the return block accessible from other fragments. Recall that we use a branch instead of a call instruction for a call block, there will be no link from the call block to the return block in the control flow graph, as indicated in Figure 4.3. Since LLVM only allows one entry block per LLVM function (we cannot jump into the middle of a function as noted before), we create a *pseudo* entry block that only contains a switch statement, and make it the entry block of this LLVM function. The real entry block and all return blocks become the different cases of this switch statement. We also add a prologue block before each of these cases. This switch statement is never executed and its sole purpose is to make the return blocks *reachable* from the entry block of this LLVM function, so that they will not be eliminated by LLVM optimization. Please refer to Figure 4.3 for an illustration of the pseudo entry block.

When the generated code is placed in code cache, we can locate the addresses of these return blocks via BasicBlockAddressMap data structure in the LLVM compiler. After making the return blocks accessible from other fragments, we also need to update the mapping table and the related branches. For each return block, we obtain its address in the code cache as described above, and update the entries indexed by the guest return addresses in the mapping table. Finally we update the links of code fragments in code cache that will branch back to these return blocks.

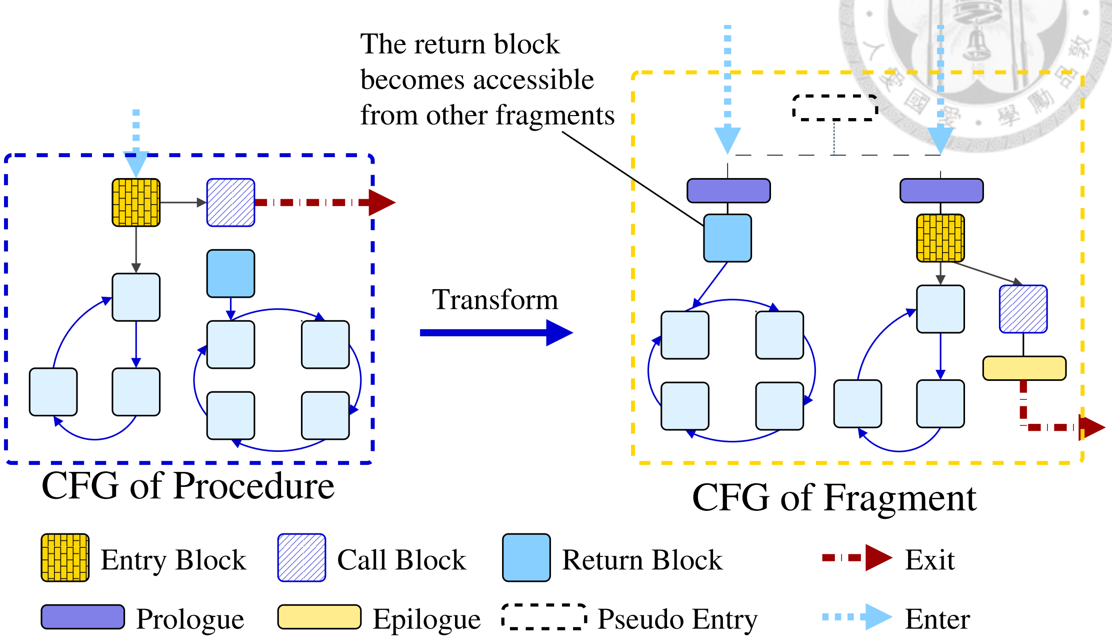


FIGURE 4.3: Illustration of an example of Call-Return problem.

4.2.2 Code/Data Distinction

When we compile a procedure in to a fragment we need to distinguish code from data in guest program. For example, if we found unreachable nodes in control flow graph, we eliminate them because they are data mistaken as instructions. It does not affect the correctness of the translated code because the execution never reaches those mis-disassembled instructions.

For ARM guest code we use the characteristic of ARM instruction set to detect data in procedures. Arm instructions use a relative addresses (to program counter) to access data. For example a load instruction `ldr r1, [pc, #1936]` the data is stored at the address of `pc + 1936`. Thus, we know the memory address `pc + 1936` is a datum, not an instruction. Detecting this kind of instructions is easy for ARM instructions set because ARM only uses load instructions to access data.

When we compile a procedure into a fragment, we need to distinguish code from data (e.g. the data on a call stack) in the guest program. For example, if we found unreachable nodes in control flow graph, we eliminate them because they are data mistaken as instructions. It does not affect the correctness of the translated code because the execution never reaches those data locations. For ARM guest code, we use the idiosyncrasies in ARM instruction set to detect data in procedures. ARM instructions use a relative addresses (to program counter) to access data. For example, for a load instruction `ldr r1, [pc, #1936]`, the data is stored at the address of `pc + #1936`. Thus, we know the memory location `pc + #1936` is a datum, not an instruction. Such detection is easy on ARM because ARM only uses load instructions to access data.

4.2.3 Targets of Indirect Branch

Indirect branches pose a serious challenge when compiling a procedure. Unlike the case of a high-level language virtual machines, such as JVM of Java, we do not have any information about the target address in a guest indirect branch. For such an indirect branch, we may have to go back to the dispatcher and determine the fragment this guest target address belongs to. However, after leaving a fragment, the execution thread could look up the mapping table to find the fragments of the guest target. The control would be directed to the block or trace fragment of the guest target address.

Fortunately, binaries generated by most compilers use *jump table* to store the target addresses of indirect branches. By recognizing and locating indirect jump tables, we can determine possible guest target addresses and add them as edges in the control flow graph. For example, x86 binary compiled by GCC 4.3.4 uses indirect branches such as `jmp *0x8065a3c(, %eax, 4)` to access a jump table for target addresses. Therefore, for an indirect branch, we first check the pattern of the memory addresses to determine whether it accesses a jump table for the target address or not. If so, we obtain those target addresses from the jump table. Otherwise, we just leave the procedure fragment and go back to the dispatcher.

4.3 Performance Evaluation

In this section, we evaluate our procedure-based optimization, and two trace guided optimizations. In the following we first describe our experiment environment and settings.

4.3.1 Experimental Environment and Settings

Experiment Environment. We conduct our experiments on an Intel Core2 CPU 975 @ 3.33GHz machine with 12GB of memory. The operating system is 64 bit Gentoo distribution Linux. We use SPEC CPU 2006 as our benchmarks. All x86 benchmarks are compiled with GCC 4.3.4 with "-static -O2 -m32" flags. All ARM benchmarks are compiled with GCC 4.4.2 with "static -O2" flags. We run all benchmarks via the standard SPEC `runspec` script with configuration files. We run each benchmark three times with reference inputs and take results reported by runspec as the experiment result.

Procedure-Based DBT System. We use LnQ [42] dynamic binary translation framework to build our binary translators and extend the framework to accommodate an optimization thread pool. We build two binary translators – one translates ARM guest ISA to x86 64 host ISA, and the other from i386 to x86 64.

We are able to run part of the benchmarks using the three translators. For ARM guest code we can run all SPEC CINT2006 benchmarks. However the SPEC runspec tool reports mis-match error when it runs h264ref benchmark. Nevertheless we notice that the runspec tool also reports mis-match error it when runs h264ref in native ARM host environment, so we will not report the results of h264ref.

We set LLVM JIT compilation flag to "O0" and enable fast instruction selection to minimize compilation overhead of execution threads. We set JIT compilers of optimization threads to "O2" option to get better code quality for traces and procedures. We observe that there is no significant performance gain when "O3" option is turned on when compared to the "O2" option. Execution count thresholds for trace and procedure are set to 30 and 40 respectively. We make the procedure threshold larger than the trace threshold so that there will be a large number of traces to use when hot procedures are identified and compiled.

We compare the procedure-based DBT with EEG region-based DBT as described in Chapter 3. We use the performance of native run as the baseline and all performance data of LnQ DBTs are normalized to the native run, that is, the results are slowdown factors compared to the native run.

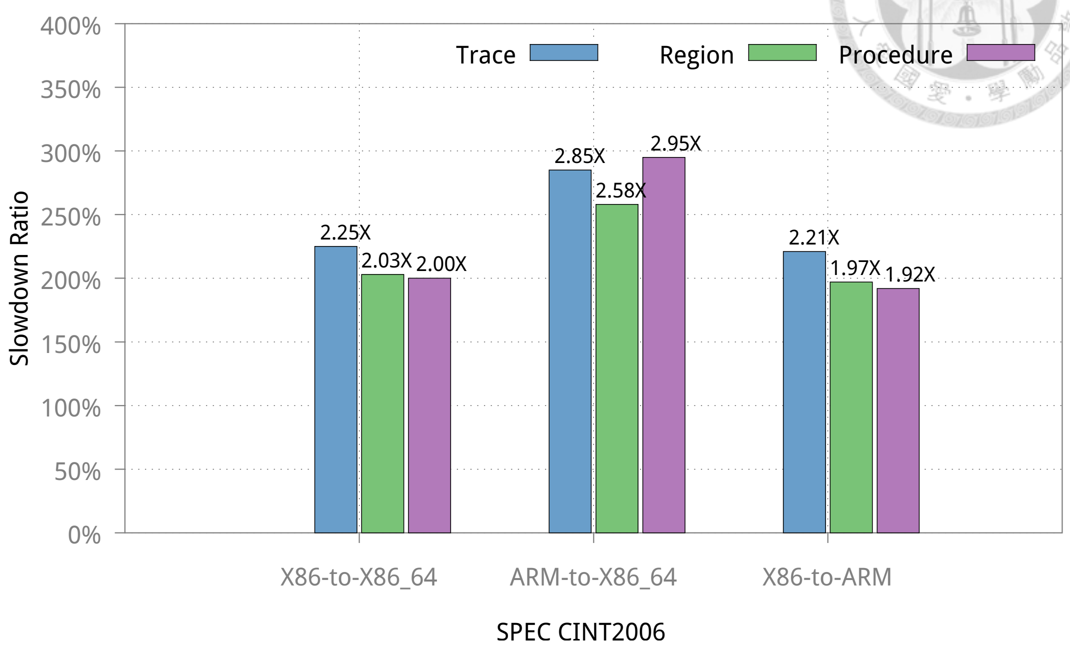


FIGURE 4.4: Overview of performance of Procedure-Based LnQ.

4.3.2 Experimental Results

4.3.2.1 Overview of the Performance

Figure 4.4 shows overall performance of 3 region formation algorithms. The Trace/Region represent NET* and EEG Region formation.

As shown in Figure 4.4, the Procedure-based approach has similar performance as EEG region formation on X86-to-X86_64 and X86-to-ARM LnQ DBTs while it has bad performance on ARM-to-X86_64 LnQ. On average, the Procedure-based approach achieves 2.0X, 2.95X and 1.92X on X86-to-X86_64, ARM-to-X86_64, and X86-to-ARM LnQ DBTs, the lower the better.

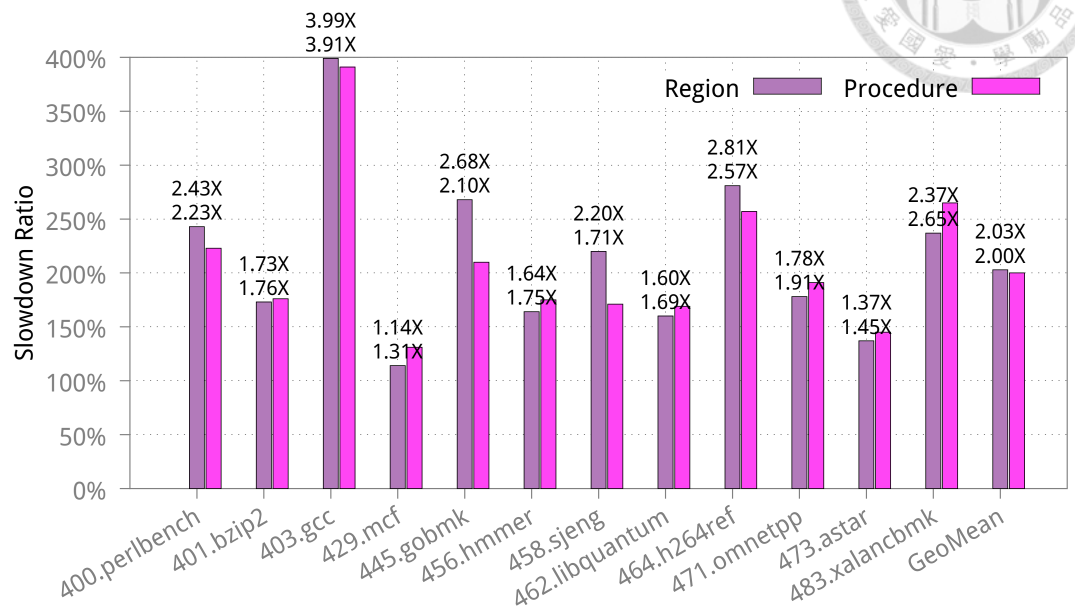


FIGURE 4.5: Detailed Performance of Procedure-Based X86-to-X86_64 LnQ.

4.3.2.2 Detailed Performance of Procedure-Based DBT

Figure 4.5, Figure 4.6 and Figure 4.7 show the performance of Procedure-based DBT.

4.4 Concluding Remarks

In this chapter, We present a retargetable procedure-based multi-threaded dynamic binary translation system. We describe how to solve Call-Return problem and other issues in procedure-based compilation.

A prototype was built to study various design issues, and the experimental results show that, in the SPEC 2006 integer benchmark, On average, the Procedure-based

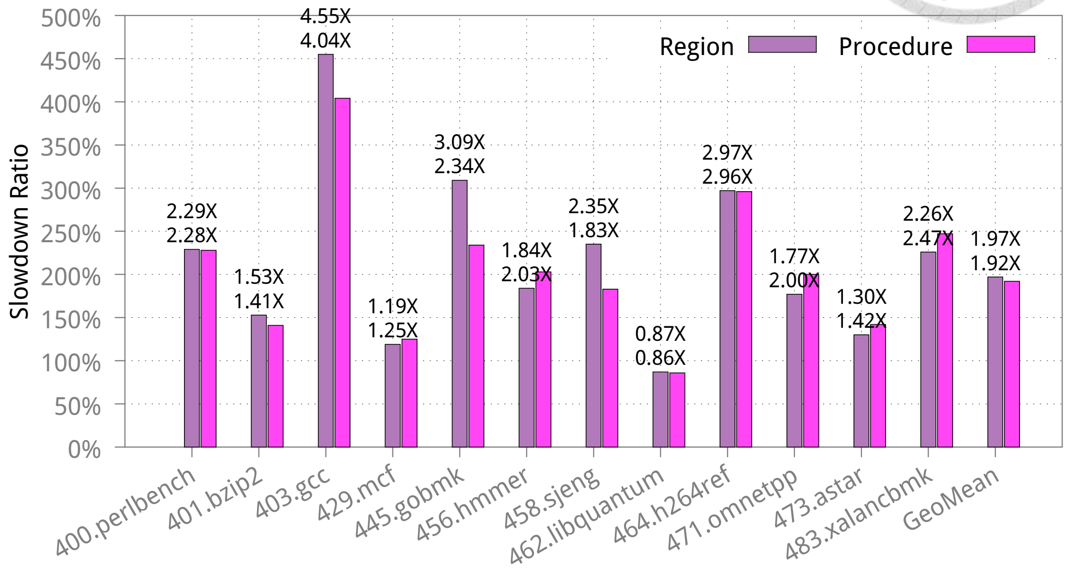


Figure 4.6: Detailed Performance of Procedure-Based X86-to-ARM LnQ.

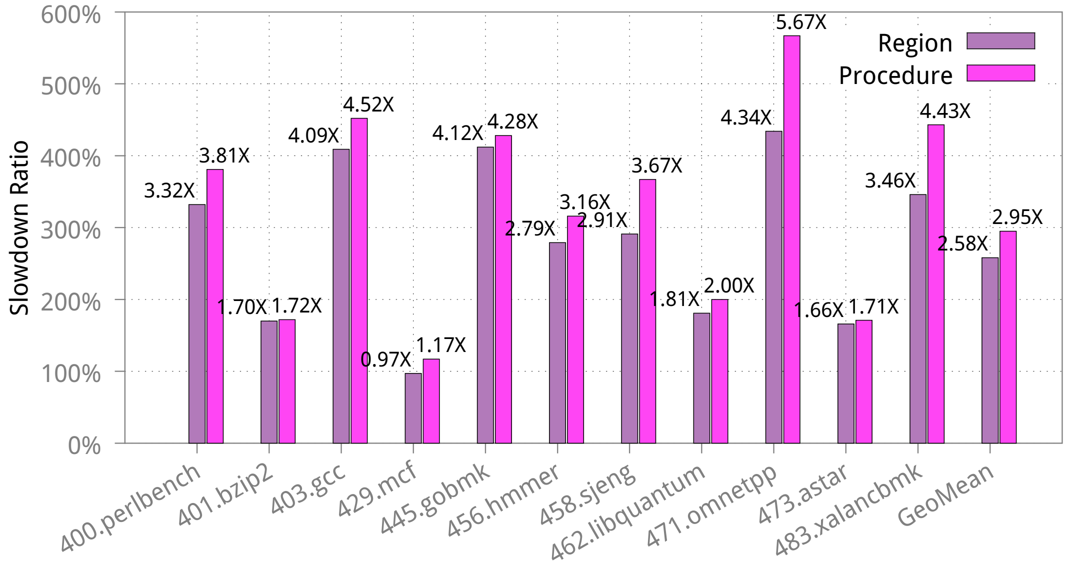


Figure 4.7: Detailed Performance of Procedure-Based ARM-to-X86_64 LnQ.

approach achieves 2.0X, 2.95X and 1.92X on X86-to-X86_64, ARM-to-X86_64, and X86-to-ARM LnQ DBTs.

Chapter 5

Related Works

5.1 Related Works of Dynamic Binary Translation (DBT)

5.1.1 QEMU

The most popular retargetable binary translator is QEMU [23]. Prior to QEMU verion 0.10, QEMU translated guest instructions into a series of micro operations. Each micro operation is implemented by `C` and is compiled into host machine instructions by gcc. There are two disadvantages of this approach. First, the generated host instructions can only be blindly pasted into TBB, and is difficult to be further optimize according to the context of guest basic block, such as registers mapping. Second, the approach is tightly bound to a specified version of gcc so that the micro operation can be compiled correctly.

LnQ framework does not have these two shortcomings. First, the Just-In-Time compiler can load LLVM IRs at runtime and further optimize them according to the context of current guest basic block, applying optimization passes such as constant propagation, unreachable basic block elimination, etc. Second, the generated IR follows LLVM semantics and can be understood by LLVM components, so it is not restricted to any specific version of LLVM-enabled compilers.

QEMU then uses its own tiny code generator (TCG) after version 0.10. As before, the guest instructions are split into several TCG operations, then the TCG parses those micro operations and generates machine code. However, TCG is designed for compilation speed and is difficult to implement sophisticated transformation in TCG. For example, it is difficult to change the layout of basic blocks in TCG. This is why TCG has only two simple optimizations: liveness analysis and simple instruction simplification. Also, the translation ability of TCG currently is still insufficient. For example, TCG does not support floating point operations yet [40], which results in poor performance in floating point benchmarks.

5.1.2 Retargetable Dynamic Binary Translators

Chipounov et al [48] use LLVM to compile micro operation functions of QEMU version 0.9. This approach is basically translating the intermediate representation generated by QEMU into LLVM IR and then generate host binary. The advantage of this approach is it can support all guest ISA supported by QEMU since it use QEMU to decode the guest instructions. The disadvantages of this approach is its inefficient as reported in [48]. Possible causes may be there are redundant IRs generated from micro functions and the insufficient translation ability of QEMU as mentioned above. LnQ translates each guest instruction *directly* into LLVM instructions, and this approach can achieve much better performance as shown in Section 2.3.

Hong et al [49] also use LLVM to optimize translated code. In HQEMU, TCG is translated into LLVM IR to achieve retargetability. While HQEMU benefits from the retargetability of TCG, it is also limited by the ability of TCG. For example, due to lack of SIMD vector and floating-point IR in TCG, HQEMU can only use helper functions to implement the semantics of those guest instructions. Of course HQEMU can inline those helper functions to improve performance, which is the translation function approaches used in LnQ to implement the semantics of guest SIMD instructions and float-point instructions.

As for region formtion, HQEMU uses NET to detect hot regions and merges hot traces within loops. That is, HQEMU uses hardware performance monitors to detect hot traces. Once HQEMU detects hot traces, it check if these traces could form a loop. If yes, those traces within the loop will be merged into a larger region.

There are two differeces between HQEMU's trace merging and LnQ's EEG region formtion. First, the merged regions in HQEMU may still suffer the inefficiency of early exists because HQEMU only analyzes the control-flow graph of traces and speculatively merges those traces as long as there is a loop found without checking if the loop will be actually executed. On the other hand, EEG merges two regions only if it detects frequent transfers between them. Second, the merged region will not be merged again. On the other hand, in EEG, regions could evolve over time. That is, regions could be merged again as long as frequent early exists are detected in that region.

Walkabout [17] is a retargetable binary translation framework developed by University of Queensland and Sun Microsystems. It uses a machine dependent intermediate representation to translate and execute binary code from a source machine on a host machine. Walkabout uses machine specifications to describe the syntax and semantics of source and host machine instructions, and how to select hot paths. The performance reported in [17] is a slowdown factor of 139. Even with PathFinder on SPARC.V9 the slowdown factor was 0.6 to 15.

5.1.3 Other Dynamic Binary Translators

The most relevant works in dynamic binary translation are Ha et al. [50] and Bohm et al. [30]. Both works propose the strategy of spawning one or multiple optimizations threads for JIT trace compilation so that concurrent interpretation and JIT trace compilation can be achieved. Their difference with our work is that our system use emulation rather than interpretation. Furthermore, we use optimization threads to compile not only traces but also procedures, and we use traces to further improve procedure compilation.

Dynamic binary translation (DBT) is widely used to support legacy binary code to run on a new architecture such as IA-32EL [16], DAISY [19], and Transmeta [51]. IA32-EL is a process virtual machine that enables IA32 applications to run on Intel Itanium. IA32-EL uses hyper-blocks as its unit of optimization in the hot code translation phase. A hyper block is a set of predicated basic blocks with a single entry and multiple exits. IA32-EL forms hyper blocks based on the execution counts of basic blocks and edge counters collected collected during the cold code execution.

DAISY and Transmeta are system virtual machines, where DAISY supports IBM PowerPC applications to run on VLIW processors and Transmeta supports IA-32 applications to run on a proprietary VLIW processor. Transmeta did not revealed details about how to find hot code regions. IBM DAISY uses *tree groups* as the translation unit. Tree groups have a single entry point and multiple exit points. No control flow joins are allowed within a tree group. Control flow joins can only occur on group transitions. Like IA32-EL, DAISY also uses profiling information collected during interpretation for tree group formation. Both hyper-blocks and tree groups have little advantage to non-VLIW machines, such as x86_64, since they are primarily designed to maximize instruction-level parallelism in VLIW architectures. Therefore we do not apply their approach in our system.

Moreover, DAISY, Transmeta, and IA32-EL handle early exits with chaining, i.e. the execution directly transfers to another code region. The transition overhead in those systems is not as high as in LnQ because most guest architecture states are mapped to the host architecture in these systems. For example, IA32-EL maps the state of IA-32 guest registers directly to Itanium registers. On the other hand LnQ, a retargetable dynamic binary translator, does not make any assumption about the guest and host ISAs. Consequently LnQ has to load guest states in the prologue of code fragments, and save them back to memory in the exit stubs, which incurs transition overheads.

5.2 Related Works of Region Formation

5.2.1 NET

ADORE [52] and Dynamo [7] are same-ISA dynamic binary optimizers, which means the input and the output instructions are from the same instruction set architecture. Both ADORE and Dynamo use traces, i.e. super-blocks, as the unit of optimization.

ADORE uses Hardware Performance Monitor (HPM) sampling approach to collect path profiles from the Branch Target Buffer (BTB) hardware performance counters in Itanium. It forms traces based on the collected path profiles.

Dynamo was the first trace-based dynamic optimizing compiler that used the Next-Executing-Tail (NET) algorithm. Dynamo pioneered many early concepts of trace formation and trace runtime management. Many DBT systems [8, 30, 49, 53] and just-in-time compilers [32, 34, 54, 55] use NET or its variants to form traces.

5.2.2 Most Recently Executed Tail^2

StarDBT [53] uses Most Recently Executed Tail (MRET)^2 [56], which improves NET by increasing the completion rate of traces. MRET^2 first uses NET to select a potential trace, then it clears block execution counters and restarts NET to select another potential trace. Both potential traces share the same starting address but may have different tails. MRET^2 then improves the completion rate by selecting the common path of both potential traces as a hot trace.

5.2.3 NETPlus

Another improvement of NET is proposed by Davis et al. [55] called NETPlus. NETPlus first forms traces just like NET, and when a NET tracce is formed, the NETPlus performs a forward search for loops back to the trace head by analyzing the control-flow graph.

5.2.4 Last-Executed Iteration (LEI)

Hiniker et al. [4] proposed Last-Executed Iteration (LEI) and a trace combination algorithm, which needs to interpret each taken branches to form trace and requires every taken branch to be added to a circular history buffer. This results in significant overhead of execution. Thus LEI has never been implemented in a real system.

The main difference between the proposed EEG and previous works is that EEG expands the existing regions and re-optimizes them during execution. The process of region expansion in EEG can be divided into three stages. The first stage is to decide how to form the initial region. The second stage is to decide when to expand the region. The third stage is to decide which blocks are to be merged. Previous trace formation algorithms, such as NETPlus and MRET^2, could be used in the first stage of EEG to build the initial regions. Therefore, the proposed EEG can be used effectively in most trace-based dynamic binary translators.

5.3 Related Works of Language Virtual Machines

5.3.1 Method-Based Language Virtual Machines

Region expansion is widely used in method-based JIT systems, e.g., HotSpot Java VM [57]. These JIT systems compile methods as follows. When a method-based JIT system 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.

Our EEG and method-based JIT systems use similar heuristics to decide when to expand regions during the second stage of region expansion, but they are very different in the first stage and the third stage of region expansion in terms of motivation and the type of blocks they merge.

The major difference between EEG and those systems in the first stage is the motivation in forming the initial regions. EEG uses traces as initial regions for two reasons. First, traces represent those frequently executed paths that may span across several methods. Second, it takes less time to optimize traces because of their simple control flow graph and small numbers of basic blocks. For example, we found only 4.2 blocks per trace in EEG. On the other hand, method-based JIT systems build initial regions by selecting blocks from hot methods, and excluding those blocks that are rarely executed. For example, HotSpot JVM excludes blocks that are never executed during interpretation.

The major difference between EEG and method-based JIT systems in the third stage is the type of blocks they merge. In the third stage EEG merges traces that contains frequently executed paths. However, in the third stage method-based JIT systems will only merge blocks that are rarely executed in the first stage, since those frequently executed blocks in the first stage have already been merged. Another difference between method-based JIT and this dissertation is that we integrate trace-guided optimization into procedure-based DBT.

Although procedure-based JIT compilation has been widely studied in language virtual machines [36, 58], it has not been studied adequately outside of the JVM community.

Suganuma et al. [36] investigate how to use region-based compilation to improve the performance of method-based Java Just-In-Time compilation. They use region-based compilation to partially inline procedures, instead of using traditional method inlining techniques. They collect execution counts of basic blocks in order to understand program runtime behavior, and they apply static code analysis on the Java bytecode to identify those rarely executed code blocks, such as those handle exception. They use these information to identify and optimize those often executed code blocks only, without optimizing the entire method.

In our case it is difficult to identify those rarely executed regions by a static code analysis, as they did for Java bytecode. Therefore we cannot apply their approach in our system.

5.3.2 Trace-Based Language Virtual Machines

Trace-based compilation has gained popularity in dynamic scripting languages [32, 59] and high level language virtual machines [34, 35, 54, 60] in recent years. Wu et al. [54] and Inoue et al. [34, 60] investigate the performance of several variations of NET on trace-based Java virtual machines.

Gal et al. [32] propose merging loop traces into a *trace-tree*. Their approach requires adding annotation while compiling JavaScript into bytecode, and thus cannot be applied in our case.

In contrast, our EEG merges delinquent traces/regions, which are not necessarily loop traces. EEG uses hardware monitoring to identify often executed code traces, then determines whether they have many side exits, and finally merges those often executed code regions that have many side exits to avoid early exits from a region, EEG also uses spill index to prevent generating regions which may degrade performance.

Chapter 6

Conclusion and Future Works

6.1 Conclusion

The performance of dynamic binary translators is determined by the quality of translated code and the ability to detect hot regions at runtime. The retargetability also reduce the effort of reimplementation dynamice binary translators with high performance. This dissertation presents the design and implementation of the LLVM+QEMU (LnQ) framework, which can build retargetabile and high performance multi-threaded trace-based/procedure-based dynamic binary translators in Chapter 2. We show how to use LLVM compiler infrastructure to design the IR library and the IR translator in the translation module on LnQ. We also show how we perform register mapping to remove redundant loads and stores of guest architecture states, and how to implement dynamic binary translation optimizations in LnQ which can be used in all translators built from LnQ.

In Chapter 3, this dissertation has identified and quantified the delinquent trace problem in the popular Next-Executing-Tail (NET) trace formation algorithm. Delinquent traces contain frequently taken early exits which cause significant overhead and limit the performacne of dynamic binary translators. Motivated by this problem, in this dissertation, we develop a light-weight region formation strategy called Early-Exit Guided region formation (EEG) to improve the performance of NET by merging delinquent traces into larger code regions. The EEG region formation does not only improve NET performance but all other trace formation approaches where traces are suffered from the inefficiency of early exits. We also show how to off-load compilation overhead to other CPUs to fully utilize the computation power of multi-core CPUs.

In this dissertation, we further study region formation in a more large granularity: a whole procedure. In Chapter 4 we present a trace-guided procedure-based region formation algorithm for dynamic binary translation. We describe how to solve Call-Return problem and other issues that only happen in dynamic binary translation.

A prototype was built to study various design issues, and the experimental results show that, in the SPEC 2006 integer benchmark, On average, the Procedure-based approach achieves 2.0X, 2.95X and 1.92X slowdown compared to the native run on X86-to-X86_64, ARM-to-X86_64, and X86-to-ARM LnQ DBTs.

6.2 Future Research Direction

Based on the evaluation in Section 4.3, procedure-based region formation shows good performance compared to NET. We believe a good future research direction will be using ahead-of-time (AOT) compilation in procedure-based region formation approaches. Similar to static binary translation, procedures can be analyzed and compiled before program execution. Traces, on the other hand, cannot be known before program execution. Hence, traces are not suitable targets for AOT compilation. There are several advantages for AOT compilation. First, the compilation overhead of procedures is completely eliminated. Second, more aggressive optimizations are possible for AOT compilation to compile procedures. For example, we may be able to collect path profiling information as the input to feedback-directive optimizations in AOT compilation. However, there are challenges for AOT compilation for procedures. The first challenge will be striped applications which have no symbol table information. The second challenge is how to integrate traces into pre-compiled procedures since traces are usually identified more quickly than procedures. Forming traces can improve performance before hot procedures are identified.

Bibliography

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proc. PLDI*, pages 1–12, 2000.

[2] J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism*, 6:1–24, 2004.

[3] S. Sridhar, J. S. Shapiro, E. Northup, and P. P. Bungale. HDTrans: an open source, low-level dynamic instrumentation system. In *Proc. VEE*, pages 175–185, 2006.

[4] David Hiniker, Kim Hazelwood, and Michael D. Smith. Improving region selection in dynamic optimization systems. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 141–154, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2440-0. doi: http://dx.doi.org/10.1109/MICRO.2005.22.

[5] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proc. Annual Microarchitecture Symposium*, pages 135–148, 2006.

[6] James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes.* Morgan Kaufman, 2005.

[7] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM*

SIGPLAN 2000 conference on Programming language design and implementation, pages 1–12, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: http://doi.acm.org/10.1145/349299.349303.

[8] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, Sep 2004.

[9] 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 *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: http://doi.acm.org/10.1145/1065010.1065034.

[10] Nicholas Nethercote. *Dynamic Binary Analysis and Instrumentation.* A dissertation submitted for the degree of doctor of philosophy, University of Cambridge, November 2004. URL `http://valgrind.org/docs/phd2004.pdf`.

[11] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 36–47, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X.

[12] K. Scott, N. Kumar, B.R. Childers, J.W. Davidson, and M.L. Soffa. Overhead reduction techniques for software dynamic translation. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 200–, April 2004. doi: 10.1109/IPDPS.2004.1303224.

[13] Qin Zhao, Derek Bruening, and Saman Amarasinghe. Umbra: efficient and scalable memory shadowing. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 22–31, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-635-9. doi: http://doi.acm.org/10.1145/1772954.1772960.

[14] 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. ISSN 0272-1732. doi: http://dx.doi.org/10.1109/40.671403.

[15] Raymond J. Hookway and Mark A. Herdeg. Digital fx!32: combining emulation and binary translation. *Digital Tech. J.*, 9(1):3–12, 1997. ISSN 0898-901X.

[16] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Yun Wang, and Y. Zemach. Ia-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium®-based systems. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 191–201, Dec. 2003. doi: 10.1109/MICRO.2003.1253195.

[17] Cristina Cifuentes, Brian Lewis, and David Ung. Walkabout - a retargetable dynamic binary translation framework. In *In Proceedings of the 2002 Workshop on Binary Translation*, 2002.

[18] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Joseph A. Fisher. Deli: a new run-time control point. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 257–268, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. ISBN 0-7695-1859-1.

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

[20] Jianjun Li, Chenggang Wu, and Wei-Chung Hsu. An evaluation of misaligned data access handling mechanisms in dynamic binary translation systems. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 180–189, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3576-0. doi: http://dx.doi.org/10.1109/CGO.2009.22.

[21] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. *SIGPLAN Not.*, 35(7):41–51, 2000. doi: http://doi.acm.org/10.1145/351403.351414.

[22] David Ung and Cristina Cifuentes. Dynamic binary translation using run-time feedbacks. *Sci. Comput. Program.*, 60(2):189–204, 2006. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/j.scico.2005.10.005.

[23] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[24] S. DEVINE, E. BUGNION, and M. ROSENBLUM. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. United States Patent 6,397,242.

[25] Chen Yu, Ren Jie, Zhu Hui, and Shi Yuan Chun. Dynamic binary translation and optimization in a whole-system emulator -skyeye. In *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, pages 8 pp.–336, 0-0 2006. doi: 10.1109/ICPPW.2006.32.

[26] Matthew Chapman, Daniel J. Magenheimer, and Parthasarathy Ranganathan. Magixen: Combining binary translation and virtualization. http://www.hpl.hp.com/techreports/2007/HPL-2007-77.html, 2007.

[27] Z. Wang, R. Liu, Y. Chen, X. Wu, H. Chen, W. Zhang, and B. Zang. COREMU: a scalable and portable parallel full-system emulator. In *Proc. PPoPP*, 2011.

[28] J.-H. Ding, Y.-C. Chung, P.-C. Chang, and W.-C. Hsu. PQEMU: A parallel system emulator based on QEMU. In *1st International QEMU Users Forum*, 2011.

[29] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9.

[30] I. Bohm, T.J.K. Edler von Koch, S.C. Kyle, B. Franke, and N. Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proc. PLDI*, 2011.

[31] J. Ha, M. R. Haghighat, S. Cong, and K. S. McKinley. A concurrent trace-based just-in-time compiler for javascript. In *Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures*, 2009.

[32] 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 *PLDI*, pages 465–478, 2009.

[33] D. Merrill and K. Hazelwood. Trace fragment selection within method-based jvms. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 41–50, 2008.

[34] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. A trace-based java jit compiler retrofitted from a method-based compiler. In *CGO'11*, pages 246–256, 2011.

[35] H. Hayashizaki, P. Wu, H. Inoue, M. J. Serrano, and T. Nakatani. Improving the performance of trace-based systems by false loop filtering. In *ASPLOS*, pages 405–418, 2011.

[36] Toshio Suganuma, Toshiaki Yasue, and Toshio Nakatani. A region-based compilation technique for a java just-in-time compiler. In *PLDI '03*, pages 312–323. ACM, 2003. ISBN 1-58113-662-5. doi: 10.1145/781131.781166. URL `http://doi.acm.org/10.1145/781131.781166`.

[37] QEMU. http://qemu.org.

[38] Low Level Virtual Machine (LLVM). http://llvm.org.

[39] Apala Guha, Kim hazelwood, and Mary Lou Soffa. Dbt path selection for holistic memory efficiency and performance. *SIGPLAN Not.*, 45(7):145–156, 2010. doi: http://doi.acm.org/10.1145/1837854.1736018.

[40] Tiny Code Generator (TCG) Documentation. http://wiki.qemu.org/Documentation/TCG.

[41] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. *SIGARCH Comput. Archit. News*, 35:412–423, June 2007. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/1273440.1250713. URL `http://doi.acm.org/10.1145/1273440.1250713`.

[42] 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 *ICPP*, pages 226–234, 2011.

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

[44] Vijay Sundaresan, Daryl Maier, Pramod Ramarao, and Mark Stoodley. Experiences with multi-threading and dynamic class loading in a java just-in-time compiler. In *CGO '06*, pages 87–97, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2499-0. doi: 10.1109/CGO.2006.16. URL `http://dx.doi.org/10.1109/CGO.2006.16`.

[45] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. The superblock: an effective technique for vliw and superscalar compilation. *J. Supercomput.*, 7(1-2):229–248, May 1993. ISSN 0920-8542. doi: 10.1007/BF01205185. URL `http://dx.doi.org/10.1007/BF01205185`.

[46] perfmon. perfmon2. http://perfmon2.sourceforge.net.

[47] unstrip tool. unstrip tool in dynamic instrumentation library. http://www.paradyn.org/html/tools/unstrip.html.

[48] Vitaly Chipounov and George Candea. Dynamically Translating x86 to LLVM using QEMU. Technical report, 2010.

[49] Ding-Yong Hong, Chun-Chen Hsu, Pangfeng Liu, Chien-Min Wang, Jan-Jan Wu, , Pen-Chung Yew, and Wei-Chung Hsu. Hqemu: A multi-threaded and retargetable dynamic binary translator on multicores. In *CGO '12: Proceedings of the 10th annual IEEE/ACM international symposium on Code generation and optimization*, 2012.

[50] J. Ha, M.R. Haghighat, S. Cong, and K.S. McKinley. A concurrent trace-based just-in-time compiler for single-threaded javascript. In *Workshop on Parallel Execution of Sequential Programs on Multicore Architectures*, 2009.

[51] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing^{TM}software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X.

[52] 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*, 6:2004, 2004.

[53] Cheng Wang, Shiliang Hu, Ho-Seop Kim, Sreekumar R. Nair, Mauricio Breternitz Jr., Zhiwei Ying, and Youfeng Wu. Stardbt: An efficient multi-platform dynamic binary translation system. In *ACSAC'07*, pages 4–15, 2007.

[54] Peng Wu, Hiroshige Hayashizaki, Hiroshi Inoue, and Toshio Nakatani. Reducing trace selection footprint for large-scale java applications without performance loss. In *OOPSLA '11*, pages 789–804, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048127. URL `http://doi.acm.org/10.1145/2048066.2048127`.

[55] Derek Davis and Kim Hazelwood. Improving region selection through loop completion. In *ASPLOS Workshop on Runtime Environments/Systems, Layering, and Virtualized Environments*, RESoLVE, Newport Beach, CA, March 2011.

[56] Chengyan Zhao, Youfeng Wu, J. Gregory Steffan, and Cristiana Amza. Lengthening traces to improve opportunities for dynamic optimization. In *Proceedings of the Workshop on Interaction between Compilers and Computer Architectures*, 2008.

[57] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspot(tm) server compiler. In *In USENIX Java Virtual Machine Research and Technology Symposium*, pages 1–12, 2001.

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

[59] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann, and Herman Venter. Spur: a trace-based jit compiler for cil. *SIGPLAN Not.*, 45:708–725, October 2010. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1932682.1869517. URL `http://doi.acm.org/10.1145/1932682.1869517`.

[60] Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. Adaptive multi-level compilation in a trace-based java jit compiler. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 179–194, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384630. URL `http://doi.acm.org/10.1145/2384616.2384630`.