LnQ: Building High Performance Dynamic Binary Translators with Existing Compiler Backends

Chun-Chen Hsu, Pangfeng Liu
Department of Computer Science and Information Engineering
National Taiwan University
Taiwan, R. O. C.
{*d95006,pangfeng*}*@csie.ntu.edu.tw*

Chien-Min Wang, Jan-Jan Wu, Ding-Yong Hong, Pen-Chung Yew
Institute of Information Science
Academia Sinica
Taiwan, R. O. C.
{*cmwang,wuj,dyhong,yew*}*@iis.sinica.edu.tw*

Wei-Chung Hsu
Department of Computer Science
National Chiao-Tung University
Taiwan, R. O. C.
hsu@cs.nctu.edu.tw

***Abstract*—This paper presents an LLVM+QEMU (LnQ) framework for building high performance and retargetable binary translators with existing compiler modules. Dynamic binary translation is a just-in-time (JIT) compilation from binary code of guest ISA to binary code of host ISA. The quality of translated code is critical to the performance of a dynamic binary translator, which translates code between different ISAs, so the translated code is often carefully hand-optimized. As a result, it takes tremendous implementation efforts for software engineers to port an existing dynamic binary translator to a new host ISA. 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. LnQ framework consists of a translation module and an emulation engine. We deisgn the translation module based on LLVM compiler infrastructure, and use QEMU as our emulation engine. We implement an x86-to-x86_64 dynamic binary translator with our LnQ framework to show that the framework is retargetable, and conduct experiments on SPEC CPU2006 benchmarks to show that the resulting binary translator has good perfromance. The experiment results indicate that the x86-to-x86_64 LnQ translator achieves an average speedup of 1.62X in integer benchmarks, and 3.02X in floating point benchmarks than QEMU.**

***Keywords*-Dynamic Binary Translation, Optimization, LLVM, QEMU**

I. Introduction

Dynamic binary translation is a just-in-time (JIT) compilation from binary code of a *guest* ISA to a *host* ISA. When the guest ISA and the host ISA are the same, we refer to it as a *same-ISA* translator. The purposes of same-ISA translator is to improve performance or to instrument binary code. For example, Dynamo [1] uses dynamic binary translation techniques to improve execution performance, and DynamoRIO [2] PIN [3], Valgrind [4], Strata [5], [6], and Umbra [7] uses dynamic binary translation techniques for program instrumentation.

When the guest ISA is different from the host ISA, we refer to it as a *cross-ISA* binary translator. Cross-ISA binary translator enables 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 [8], [9] enables application to migrate from IA-32 to Alpha, and IA-32EL [10] enables application to migrate from IA-32 to Itanium. Other migration examples include [11], [12], [13], [14], [15], [16]. QEMU [17], VmWare [18], and other virtulization systems [19], [20] use binary translation technique to provide server virtualization.

It takes tremendous efforts to build a cross-ISA translator. A same-ISA translator "translates" a binary code simply by copying it into a *code cache* [21], so there is little opportunity to improve performance by optimizing the code. On the other hand, a cross-ISA translator must prepare a *carefully hand-optimized* translation template [8], [10] for *each* guest instruction. As a result it takes tremendous efforts 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 [22]+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. We implemented three such optimizations – *block linking*, *indirect branch target caching*, and *shadow stack*. Section III will describe these optimizations in details.

The main contributions of this paper are three folds.

- We designed and implemented the LnQ framework.

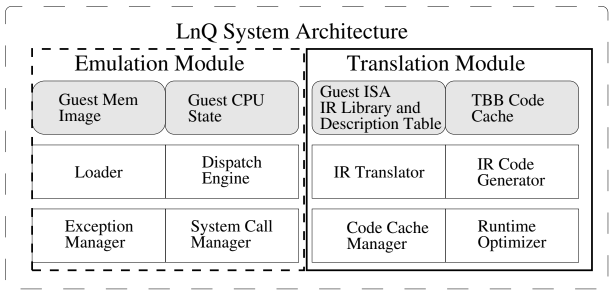


Figure 1. The architecture of LnQ framework

We introduce several novel approaches to build high performance dynamic binary translators with existing compiler modules.

- We showed how to implement binary optimization techniques in a retargetable way so that every binary translator built from LnQ can benefit from these optimizations.
- We build an x86-to-x86_64 binary translator to demonstrate the performance of LnQ. The x86-to-x86_64 LnQ binary translator is 1.62 times faster than QEMU in integer benchmarks and 3.02 times faster in floating point benchmarks.

The organization of this paper is as follows. Section II introduces the design and implementation of the LnQ framework. Section III describes how we implement retargetable runtime optimizations in LnQ framework. Section IV describes experiment results and evaluates the performance of LnQ framework. Section V introduces related works and Section VI concludes.

II. 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 1. We use QEMU [23] as our emulation module, and build the translation module with LLVM [24] 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.

A. LLVM Intermediate 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.

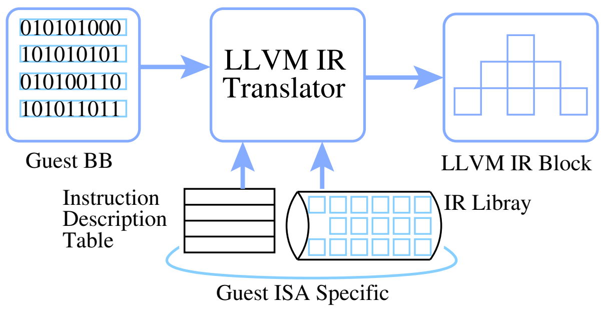


Figure 2. Illustration of the design and execution flow in LnQ translation module.

B. 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 compilers, 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.

C. 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 3 Illustrates all steps of the translation flow.

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.

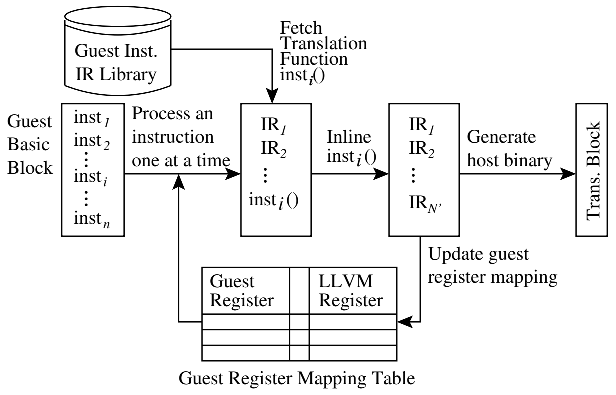

Guest Inst. IR Library
Fetch Translation Function inst$_i$()
Guest Basic Block
inst$_1$ inst$_2$ ⋮ inst$_i$ ⋮ inst$_n$
Process an instruction one at a time
IR$_1$ IR$_2$ ⋮ inst$_i$()
Inline inst$_i$()
IR$_1$ IR$_2$ ⋮ IR$_{N'}$
Generate host binary
Trans. Block
Update guest register mapping
Guest Register
LLVM Register
Guest Register Mapping Table

Figure 3. Illustration of the translation process.

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*.

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.

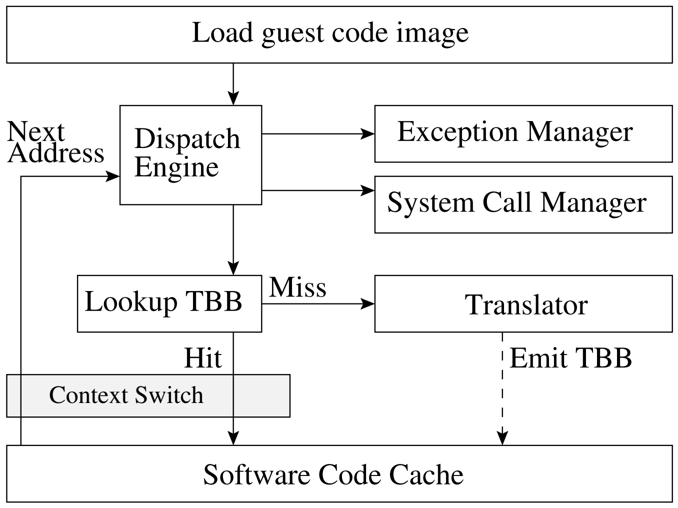


Figure 4. Illustration of the control flow in emulation module.

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.

D. Emulation Module

Figure 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.

III. 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.

A. Block Linking

Block linking [1] 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 [2], [25].

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

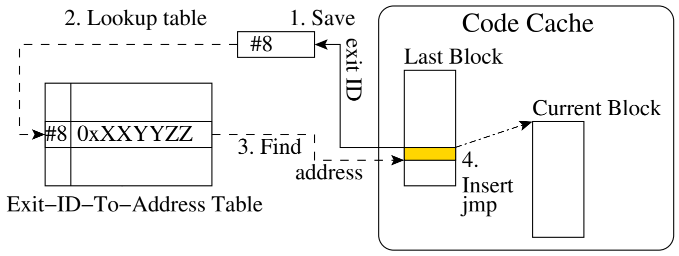


Figure 5. Illustration of execution steps of block linking optimization.

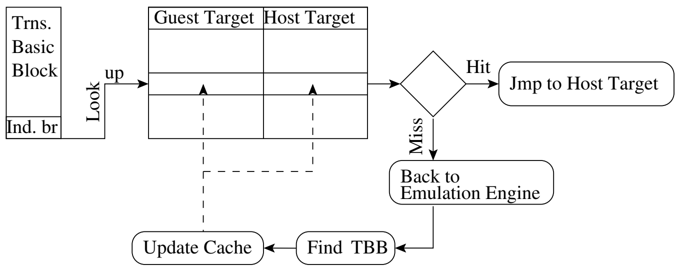


Figure 6. Illustration of execution steps of indirect branch target caching optimization.

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 5 for an illustration of our implementation.

B. 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 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.

C. Shadow Stack

Shadow stack (SS) optimizes function return mechanism in binary translation, and was first introduced in FX!32 [8]. 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.

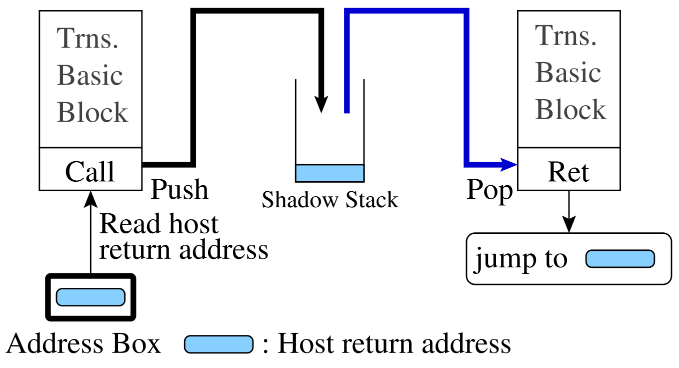


Figure 7. Illustration of execution steps of shadow stack optimization.

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 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.

IV. Experiments

1) Performance of LnQ: 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 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".

A. 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.

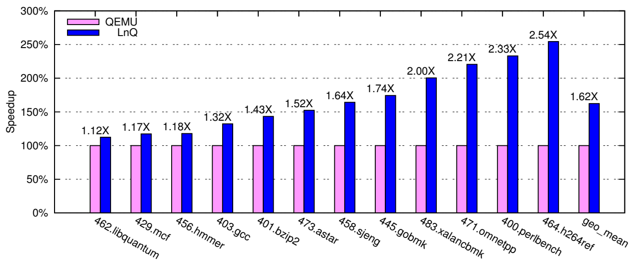
We run all benchmarks via the standard SPEC *runspec* script with configuration files. We run each benchmark three times with reference inputs and take the average as the experiment result. We compare the performance of our LnQ with QEMU 0.13.0.

B. Performance of LnQ

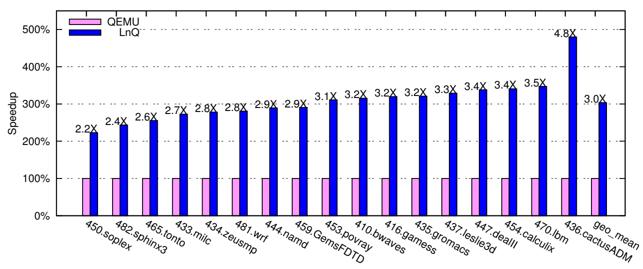
We first compare the runtime performance of LnQ with QEMU. The results are shown in Figure 8 where Figure 8(a) shows results of integer benchmarks and Figure 8(b) shows results of floating point benchmarks. The Y-axis is speedup factor of running time of LnQ compared with QEMU.

As shown in Figure 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 [26]. As a result TCG translates all floating point instructions into helper function calls. On



8(a): Results of Integer Benchmarks



8(b): Results of Floating Point Benchmarks

Figure 8. Speedup factors of LnQ in of SPEC CPU2006 compared with QEMU.

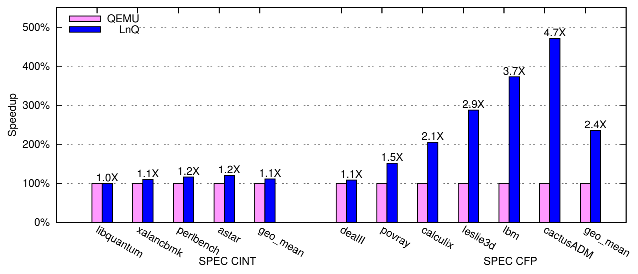


Figure 9. Speedup factors of execution time spent in code cache of LnQ compared to QEMU. The numbers above bars are

	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 I
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.

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.

C. 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 II-D, 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 [27] in following experiments. We, however, still show the geometric means derived from all benchmarks of SPEC CINT 2006 and SPEC CFP 2006.

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 9.

From Figure 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 8 can be contributed by runtime optimizations, which we further investigate the effects of runtime optimizations in Section IV-D.

2) Translation Overhead: We now evaluate the translation overhead of LnQ. Table I 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

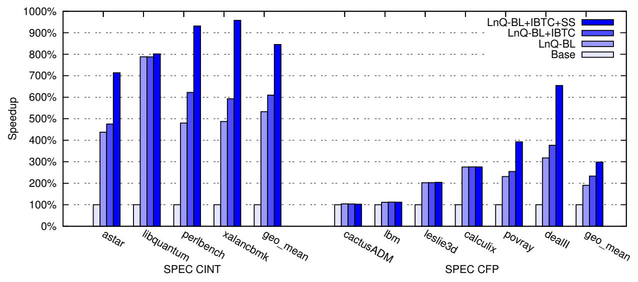


Figure 10. Performance of block linking, IBTC, and shaow stack

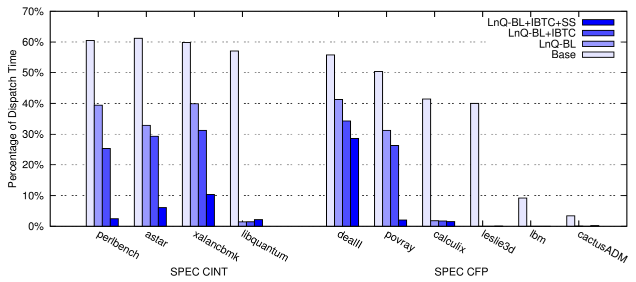


Figure 11. The reduction of percentage of dispatch time.

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 [10] into LnQ framework.

D. *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 III. The results are shown in Figure 10.

From Figure 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 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 11.

In Figure 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 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.

E. *Slowdown of LnQ Compared to Native Run*

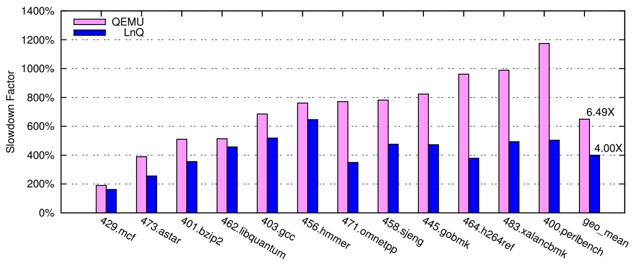
In the last experiment, we examine the slowdown factors of LnQ compared to the native runs. For native runs, we compile SPEC benchmarks only with the “-O2” option without the “-m32” option. The results are shown in Figure 12.

From Figure 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 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 12 shows that there is room for improvement fro dynamic binary translates.

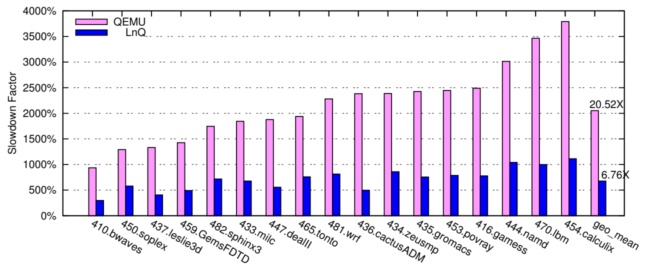
V. Related Works

The most related works to our paper is QEMU [17] and LLVM-QEMU [28]. 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.



12(a): Results of Integer Benchmarks



12(b): Results of Floating Point Benchmarks

Figure 12. The slowdown of LnQ and QEMU compared with native run.

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 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, the translation ability of TCG currently is still insufficient. For example, TCG does not support floating point operations yet [26], which results in poor performance in floating point benchmarks.

Chipounov et al [28] 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 [28]. Possible causes may be there are redundant IRs generated from micro functions and the insufficient translation ability of QEMU as mentioned above.

Our approach translates each guest instruction *directly* into LLVM instructions, and this approach can achieve much better performance as shown in Section IV.

Walkabout [11] 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 [11] is a slowdown factor of 139. Even with PathFinder on SPARC.V9 the slowdown factor was 0.6 to 15.

Pin [3] and Strata [5] are both same-ISA binary translator frameworks, which also emphasize on their retargetability. As mentioned previously same-ISA dynamic binary translators translate binary code by copying the application code into code cache, therefore they do not focus on retargetable code generation.

VI. Conclusion and Future Works

This paper 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 paper, 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.

Acknowledgment

This work is supported in part by NSC grants NSC99-2220-E-001-001, NSC99-2221-E-001-003-MY3, NSC99-2221-E-002-111-MY3, NSC99-2218-E-009-008-MY2, and is also supported in part by NSF grant CNS-0834599.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia, “Dynamo: a transparent dynamic optimization system,” in *PLDI ’00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. New York, NY, USA: ACM, 2000, pp. 1–12.

[2] D. Bruening, “Efficient, transparent, and comprehensive runtime code manipulation,” Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, Sep 2004.

[3] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. 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*. New York, NY, USA: ACM, 2005, pp. 190–200.

[4] N. Nethercote, "Dynamic binary analysis and instrumentation," A dissertation submitted for the degree of Doctor of Philosophy, University of Cambridge, November 2004. [Online]. Available: http://valgrind.org/docs/phd2004.pdf

[5] 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*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 36–47.

[6] K. Scott, N. Kumar, B. Childers, J. Davidson, and M. Soffa, "Overhead reduction techniques for software dynamic translation," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, April 2004, pp. 200–.

[7] Q. Zhao, D. Bruening, and S. Amarasinghe, "Umbra: efficient and scalable memory shadowing," in *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. New York, NY, USA: ACM, 2010, pp. 22–31.

[8] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates, "Fx!32: A profile-directed binary translator," *IEEE Micro*, vol. 18, no. 2, pp. 56–64, 1998.

[9] R. J. Hookway and M. A. Herdeg, "Digital fx!32: combining emulation and binary translation," *Digital Tech. J.*, vol. 9, no. 1, pp. 3–12, 1997.

[10] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. 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*, Dec. 2003, pp. 191–201.

[11] C. Cifuentes, B. Lewis, and D. Ung, "Walkabout - a retargetable dynamic binary translation framework," in *In Proceedings of the 2002 Workshop on Binary Translation*, 2002.

[12] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher, "Deli: a new run-time control point," in *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 257–268.

[13] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye, "Dynamic binary translation and optimization," *IEEE Trans. Comput.*, vol. 50, no. 6, pp. 529–548, 2001.

[14] J. Li, C. Wu, and W.-C. 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*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 180–189.

[15] D. Ung and C. Cifuentes, "Machine-adaptable dynamic binary translation," *SIGPLAN Not.*, vol. 35, no. 7, pp. 41–51, 2000.

[16] ——, "Dynamic binary translation using run-time feedbacks," *Sci. Comput. Program.*, vol. 60, no. 2, pp. 189–204, 2006.

[17] F. Bellard, "Qemu, a fast and portable dynamic translator," in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[18] 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.

[19] C. Yu, R. Jie, Z. Hui, and S. Y. Chun, "Dynamic binary translation and optimization in a whole-system emulator - skyeye," in *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, 0-0 2006, pp. 8 pp.–336.

[20] M. Chapman, D. J. Magenheimer, and P. Ranganathan, "Magixen: Combining binary translation and virtualization," http://www.hpl.hp.com/techreports/2007/HPL-2007-77.html, 2007.

[21] S. Sridhar, J. S. Shapiro, E. Northup, and P. P. Bungale, "Hdtrans: an open source, low-level dynamic instrumentation system," in *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*. New York, NY, USA: ACM, 2006, pp. 175–185.

[22] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO '04: Proceedings of the international symposium on Code generation and optimization*. Washington, DC, USA: IEEE Computer Society, 2004, p. 75.

[23] "QEMU," http://qemu.org.

[24] "Low Level Virtual Machine (LLVM)," http://llvm.org.

[25] A. Guha, K. hazelwood, and M. L. Soffa, "Dbt path selection for holistic memory efficiency and performance," *SIGPLAN Not.*, vol. 45, no. 7, pp. 145–156, 2010.

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

[27] A. Phansalkar, A. Joshi, and L. K. John, "Analysis of redundancy and application balance in the spec cpu2006 benchmark suite," *SIGARCH Comput. Archit. News*, vol. 35, pp. 412–423, June 2007. [Online]. Available: http://doi.acm.org/10.1145/1273440.1250713

[28] V. Chipounov and G. Candea, "Dynamically Translating x86 to LLVM using QEMU," Tech. Rep., 2010.