MFHBT: Hybrid Binary Translation System with Multi-stage Feedback Powered by LLVM

Zhaoxin Yang[1,2], Xuehai Chen[1,2], Liangpu Wang[1,2], Weiming Guo[3], Dongru Zhao[3], Chao Yang[4], and Fuxin Zhang[1,2](✉)

[1] SKLP, Institute of Computing Technology, CAS, Beijing, China
{yangzhaoxin21s,fxzhang}@ict.ac.cn

[2] University of Chinese Academy of Sciences, Beijing, China
{chenxuehai20,wangjingpu17}@mails.ucas.ac.cn

[3] University of Science and Technology of China, Hefei, China
{ustcgwm,zhaodongru}@mail.ustc.edu.cn

[4] State Grid Liaoning Electric Power Supply Co. Ltd., Shenyang, China
yangchaoneu@sina.com

Abstract. The shortage of applications has become a major concern for new Instruction Set Architecture (ISA). Binary translation is a common solution to overcome this challenge. However, the performance of binary translation is heavily dependent on the quality of the translated code. To achieve high-quality translation, recent studies focus on integrating binary translators with compilation optimization methods. Nevertheless, such integration faces two main challenges. Firstly, it is hard to employ complex compilation optimization techniques in a dynamic binary translator (DBT) without introducing significant runtime overhead. Secondly, the task of implementing register mapping in the compiler is challenging, which can reduce expensive memory access instructions generated to maintain the guest CPU state. To resolve these challenges, we propose a hybrid binary translation system with multi-stage feedback, combining dynamic and static binary translator, named MFHBT. This system eliminates the runtime overhead caused by compilation optimization. Additionally, we introduce a mechanism to implement the register mapping through inline constraints and stack variables in the compiler. We implement a prototype of this new system powered by LLVM. Experimental results demonstrate an 81% decrease in the number of memory access instructions and a performance improvement of 3.28 times compared to QEMU.

Keywords: Hybrid binary translation · LLVM · Optimization · Register mapping

1 Introduction

Binary translation is a technique that enables cross Instruction Set Architecture (ISA) compatibility [28]. It allows applications compiled for one ISA to run on another ISA without recompilation, especially when the source code is difficult to obtain or when recompiling is costly. It also enables basic software development before the hardware can be obtained. Several factors may influence the efficiency of a binary translator, including the overhead of initialization before translation, the overhead of code translation and optimization, and the overall quality of the generated code [5,21,25]. Code quality holds particular significance.

Recent studies have focused on integrating binary translators with compilers like LLVM [11,18,22] to achieve high-quality translation, which allows for the utilization of diverse general-purpose optimization techniques provided by compilers. However, two main challenges arise when integrating binary translators with compilers.

The first challenge lies in minimizing additional runtime overhead caused by the time-consuming optimization algorithms provided by compilers in dynamic binary translators (DBT). HQEMU [12,15] tackles this challenge by profiling hot traces, taking advantage of the multicore resources and multithreading itself to mitigate the optimization overhead imposed by LLVM. However, the overhead of code optimization continues to grow due to the expanding number and complexity of LLVM's optimization passes. Consequently, the effectiveness of optimization may be undermined since a greater amount of time is spent on un-optimized code. Although CrossDBT [19] and HBT [23] offload part of the optimization work to the static binary translator (SBT) they integrated, they still rely on LLVM as code optimizer during execution, resulting in additional runtime overhead. Moreover, in both CrossDBT and HBT, the static translator lacks the capability to leverage feedback information [26] from the dynamic translator for additional optimization.

Another challenge arises regarding the effective maintenance of the virtual guest CPU state across the execution of translation units. Both HQEMU and CrossDBT use memory operations for maintenance purposes, resulting in the significant overhead of memory access. Although HQEMU optimizes maintenance by performing it only before guest memory access and jump instructions, the cost of memory access remains high. Utilizing register mapping can reduce maintenance memory access overhead by caching the guest CPU state in host registers. However, specific challenges arise when applying it to LLVM IR. Firstly, LLVM IR is designed to be architecture-independent, but register mapping requires direct interaction with architecture-dependent physical registers, leading to a contradiction. Secondly, it is crucial to ensure that LLVM remains a sufficient number of registers for its own utilization after register mapping.

To solve the above issues, we present MFHBT, a hybrid binary translation system combining both DBT and SBT with multi-stage feedback powered by LLVM. The system eliminates runtime code optimization overhead by offloading all code optimization work to SBT. Furthermore, the system proposes a register mapping mechanism realized through LLVM inline constraints and stack variables to reduce memory access overhead of guest CPU state maintenance.

The contributions of this paper include:

- We design a binary translation system based on LLVM. This system eliminates translation and optimization overhead caused by LLVM during execu-

tion. Moreover, it supports continuous optimization of the translated code by enabling feedback from DBT to SBT.
– We introduce a mechanism to reduce the cost of guest CPU state maintenance when using LLVM for code optimization. This mechanism combines the use of LLVM inline constraints and stack variables to provide a register mapping scheme.
– We implement a translation system, named MFHBT-LA, from x86-64 to LoongArch [27] and test its efficiency. Experiment results demonstrate an 81% decrease in the number of memory access instructions and a performance improvement of 3.28 times compared to QEMU [3]. The source code is available at https://github.com/ylzsx/MFHBT.

2 Background

2.1 Hybrid Binary Translation

Static binary translation (SBT) is an offline translation method that does not rely on program information during runtime [6]. It transforms the original binary code from guest architecture into new binary code for the host architecture prior to program execution. This approach allows for longer translation time, enabling the application of aggressive and time-consuming optimizations to generate highly efficient translated code. However, static binary translation suffers from certain limitations and incompleteness issues, such as self-modified code, which can hinder its practicality [9].

Dynamic binary translation (DBT) involves translating individual translation unit by following the execution flow and generating code using Just-In-Time (JIT) technology [2,4,17]. The generated code is subsequently executed. Due to its comprehensive understanding of program execution, dynamic binary translation effectively addresses various issues, such as self-modified code, indirect jumps, and indirect calls. However, it is important to note that DBT is sensitive to the overall cost of code generation and optimization. As a result, more complex optimization methods in the translation module are restricted, leading to inferior code quality compared to static binary translation.

To enhance the quality of the translated code while ensuring completeness, we combine SBT and DBT [1,20], thereby enhancing the overall performance of the entire binary translation system.

2.2 Maintain Guest CPU State

In binary translation, maintaining the guest CPU state is essential. This process involves acquiring the current guest CPU state prior to executing each translation unit and updating the new guest CPU state posterior to emulating the functionality of guest instructions. The commonly used methods include the memory storage method and the register mapping method. The memory storage method requires additional instructions for memory access, resulting in reduced performance compared to the register mapping method. In the register mapping method, guest registers (GRs) are mapped to host registers (HRs). After completing each translation unit, the most recent state of GRs in the guest CPU is stored in HRs. Subsequent translation units can retrieve the updated state without the need for memory access.

3 Design

3.1 Overview

We design a hybrid binary translation system that combines both the dynamic and static side to reduce the overhead of translating and optimizing at runtime, called MFHBT. This system is powered by LLVM compilation optimization and incorporates a multi-stage feedback mechanism. An overview of the system's execution process is presented in Fig. 1.

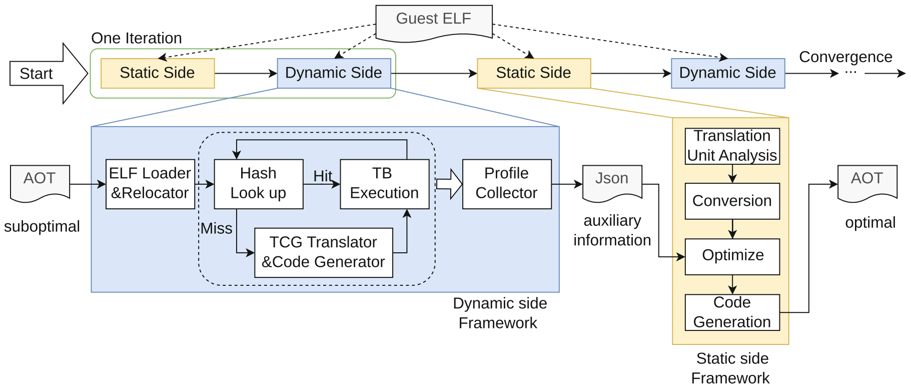


Fig. 1. Overview

During the initial iteration, the static side creates an Ahead-of-Time (AOT) file by relying solely on the translation units extracted through code mining from the guest Executable and Linkable Format (ELF) file, and no feedback information is obtained from the dynamic side. Subsequently, the dynamic side receives the AOT file and collects profiling information, which is eventually stored as a JSON file. In the second iteration, the static side examines the JSON file that was generated during the previous dynamic execution. Following that it creates superior code, which will be combined with the previous AOT file to produce a new one. The dynamic side uses this updated AOT file for execution while simultaneously collecting feedback. This iterative process continues, leading to a gradual enhancement in program performance that ultimately converges to a stable state.

The dynamic side comprises four components, functioning as ELF loading and relocation, program execution, code translation, and profile collecting. It is a lightweight binary translator that runs the high-quality generated code from the static side. Additionally, it conducts lightweight translation for basic blocks that the static side could not recognize, supplementing for the static side.

The static side is a heavyweight optimizer, built around LLVM and composed of four distinct components, functioning as translation unit analysis, instruction conversion, code optimization, and code generation. It holds two primary responsibilities: obtaining translation units and performing offline optimizations using LLVM, where the optimized code is then saved as an AOT file.

3.2 Multi-stage Feedback Mechanism

MFHBT employs a multi-stage feedback mechanism to improve the quality of generated code [7]. During each execution, MFHBT gathers information about the executed program using the profile collector on the dynamic side. This information is then stored in JSON format as profile files and utilized to aid the optimization process in the static side.

Feedback Information. This information we collect in the dynamic side can be categorized into two main aspects: code address information and instruction flow characteristics.

Code Address Information. We collect the entry address of translation units from the dynamic side and transfer them to the static side as a supplement because it is arduous to entirely identify this information through static analysis due to various factors. One challenge is determining the target addresses of indirect jumps before execution, which has been proven problematic [28]. Another challenge is the influence of parameters and execution environment on program execution paths, adding further complexity to the task. Code obfuscation techniques present additional challenges. In contrast, the dynamic side has the advantage of being able to easily identify the currently translated and executed code, which will help identify a wider range of guest code.

Instruction Flow Characteristics. We gather the instruction flow characteristics, such as hot trace paths and indirect jump target addresses [24], in our system. This information can guide further optimization in the static side, such as supplementing unrecognized translation units, expanding the range of optimization, and reordering the generated code.

Multi-stage Feedback. Our feedback mechanism operates at multiple stages, allowing each execution on the dynamic side to contribute valuable information to the static side. Factors such as program parameters, execution environment, and the program's random behavior all influence the execution path of the program. As a result, multi-stage feedback mechanism can provide more comprehensive code coverage and detailed execution flow information compared to single feedback mechanism.

Considering a program in which the execution path is influenced by the random number generated within the code. When the program is translated, it may result in different execution paths across multiple runs. During these runs, the dynamic side can capture the variations in the execution path, leading to a more thorough understanding of the program's behavior.

3.3 Register Mapping in LLVM

This paper introduces a register mapping scheme in LLVM, aiming to effectively maintain the guest CPU state. The method employs the LLVM inline constraints and stack variables, to reduce the proportion of memory access instructions in the generated code.

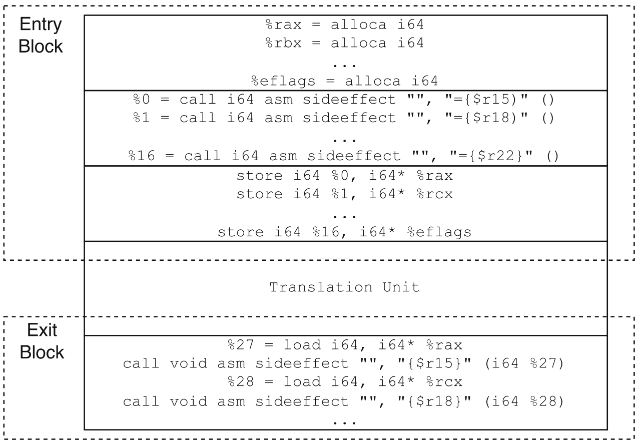
The implementation, depicted in Fig. 2, involves establishing a mapping between the guest and host registers. In the entry block, the mapping is established by three steps: 1) associating guest registers with LLVM stack variables, 2) binding host physical registers to virtual registers using the output constraint mechanism provided by LLVM IR inline assembly, 3) storing the virtual registers bound to host physical registers to LLVM IR stack variables. In the exit block, the mapping is built by two steps: 1) loading the guest registers from the stack variables into the virtual registers, 2) writing the virtual registers into the relative physical registers using the input constraint mechanism provided by LLVM IR inline assembly. In the translation unit, reading from and writing to the guest registers are translated to access the corresponding the stack variables.

Using stack variables does not result in unnecessary memory access because of LLVM's stack promotion optimization pass (mem2reg). This optimization pass elevates the operations involving stack variables to virtual registers, for which the LLVM backend will allocate physical registers. While extra register move operations may be required, the cost is significantly lower than memory access. Meanwhile, this approach restricts the utilization of physical registers solely at the entry and exit points of the translation unit, thereby preserving LLVM's exploration of physical registers during optimization.

The utilization of stack variables offers additional benefits. If stack variables are not used to cache guest registers, tracking the temporary virtual registers holding the latest value of the guest registers becomes complex, particularly when dealing with multiple levels of branching. However, stack variables facilitate efficient management of this tracking process by the compiler, thereby enhancing overall efficiency.

4 Implement

This section describes a prototype of an architecture-independent binary translation system named MFHBT-LA, which translates binary code from x86-64 to LoongArch. In the static side, it utilizes LLVM for offline optimization and in the dynamic side, it employs QEMU for handling code not covered by the static side. This system leverages LLVM and QEMU's support for multiple architectures.



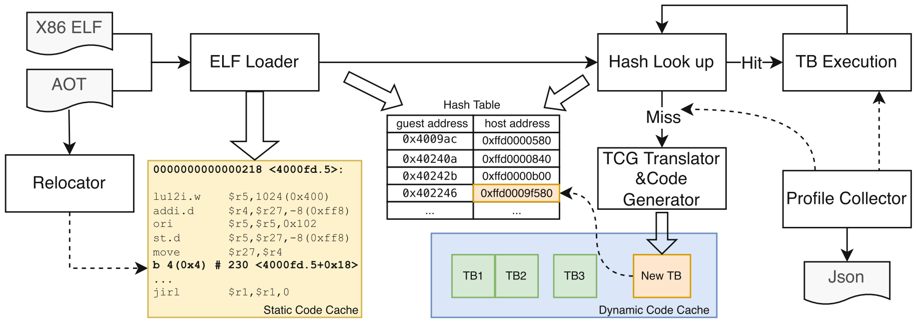
```
%rax = alloca i64
%rbx = alloca i64
...
%eflags = alloca i64
%0 = call i64 asm sideeffect "", "={$r15}" ()
%1 = call i64 asm sideeffect "", "={$r18}" ()
...
%16 = call i64 asm sideeffect "", "={$r22}" ()
store i64 %0, i64* %rax
store i64 %1, i64* %rcx
...
store i64 %16, i64* %eflags
```

```
%27 = load i64, i64* %rax
call void asm sideeffect "", "{$r15}" (i64 %27)
%28 = load i64, i64* %rcx
call void asm sideeffect "", "{$r18}" (i64 %28)
...
```

Fig. 2. An Example of Stack Translation Mode.

4.1 Dynamic Side

The dynamic side is responsible for running the pre-translated code from the static side and implementing lightweight code translation and optimization. It encompasses several tasks, including ELF loading and relocation, program execution, code translation, and profile collecting, as illustrated in the Fig. 3.



guest address	host address
0x4009ac	0xffd0000580
0x40240a	0xffd0000840
0x40242b	0xffd0000b00
0x402246	0xffd0009f580
...	...

```
0000000000000218 <4000fd.5>:

lu12i.w     $r5,1024(0x400)
addi.d      $r4,$r27,-8(0xff8)
ori         $r5,$r5,0x102
st.d        $r5,$r27,-8(0xff8)
move        $r27,$r4
b 4(0x4) # 230 <4000fd.5+0x18>
...
jirl        $r1,$r1,0
```

Fig. 3. The Design of Dynamic Side

ELF Loading and Relocation. This module comprises two components: the ELF loader and relocator. The ELF loader is responsible for loading the guest ELF file and the AOT file. Meanwhile, according to the information from the AOT file, it will establish a hash table and record link slots. The relocator fills the link slots by considering jump relationships in the guest program. This process helps to reduce the overhead of the context switch during execution.

Program Execution. Before each execution, the system will check whether a translation unit has been recorded in hash table based on the guest PC. If the unit is found, the corresponding code is executed until a context switch occurs, where control is transferred back to translator. If the unit is not found, translation begins.

Code Translation. The dynamic side performs translation using QEMU, stores the generated code into the dynamic code cache, and updates the hash table established in the ELF loading phase. It is important to distinguish between the translated code and the pre-translated AOT code in memory because direct linking is not possible when the translation protocols differ between the dynamic and static sides, such as in the case of emulating EFLAGS[1]. In such situations, the translator may need to synchronize certain states.

Profile Collecting. The profile collector keeps track of unrecognized code and the execution flow information, which allows the static side to utilize this information to generate higher-quality code in subsequent runs.

4.2 Static Side

The static side is responsible for implementing heavyweight optimizations in the system. It comprises four components, functioning as translation unit analysis, instruction conversion, code optimization, and code generation, as illustrated in the Fig. 4.

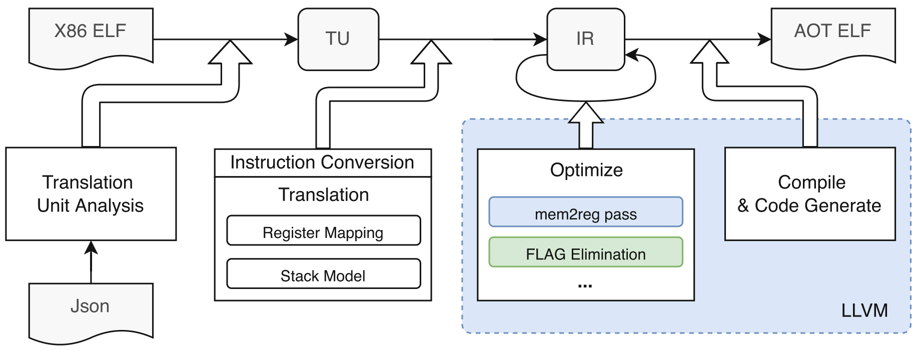


Fig. 4. The Design of Static Side

Translation Units Analysis. Translation units are obtained through two approaches: static code mining and feedback files analysis. Nonetheless, there may be cases where multiple units share the same entry address in the guest program. In such scenarios, we prioritize the unit derived from feedback files.

[1] The EFLAGS register is the status register that contains the current state of a x86 CPU.

Instruction Conversion. Each translation unit is translated to an LLVM IR function in two steps. Firstly, the translation unit is disassembled to guest instructions. Secondly, each guest instruction is lifted into LLVM IRs using a custom translation procedure. The focus is solely on ensuring the correctness of guest semantics, with an expectation of improved LLVM IR quality during code optimization.

Code Optimization. The obtained LLVM IR functions undergo optimization to enhance code quality. These optimizations involve various passes provided by LLVM, including mem2reg, function inlining, loop vectorize pass, and so on. Additionally, custom optimization passes and specific intrinsics for LoongArch architectures are implemented, such as the EFLAGS elimination pass.

Code Generation. The optimized LLVM IR functions are then transformed into host instructions using LLVM's code generation library and saved as a relocatable file following the ELF format, commonly referred to as an AOT file.

4.3 Multi-stage Feedback Mechanism

We implement a profile collector using various methods to collect feedback information in this paper, as shown in Fig. 5. Firstly, when the translation unit is missing in the hash table, we collect the entry addresses of unrecognized translation unit (①). Secondly, the NET algorithm [10] is used for hot trace paths collection (②). Finally, when dealing with the target addresses of indirect jumps, we keep a record of the guest PC and the target addresses (③).

We use the information generated to optimize the code in the static side more thoroughly. Entry addresses of unrecognized translation units are used to guide the static side to supplement the translation units in AOT file (④). Moreover, hot trace paths are used to adjust the order of basic blocks in the generated translation unit, aiming to reduce jump costs and eliminate redundant code overhead (⑤). Additionally, hot call and return instructions in the hot trace path are inlined to mitigate the overhead associated with address transformation (⑥). Furthermore, target addresses of indirect jumps are used to merge translation units that are separated by indirect jumps to expand the optimization scope (⑦).

We implement the gathering of various feedback information. And then all the collected information is stored in a standardized JSON format, enabling a consistent processing method for file handling and alleviating the associated workload.

In each iteration, a new AOT file is generated based on the feedback information received from the dynamic side. The ELF standard format ensures that all files are relocatable, allowing them to be linked with existing files through the use of GNU ld. This process decreases the overhead of re-generating AOT files in the static side.

4.4 Register Mapping in LLVM

We introduce a cache for each virtual register in the LLVM IR associated with a guest register to reduce the frequency of read and write operations on stack

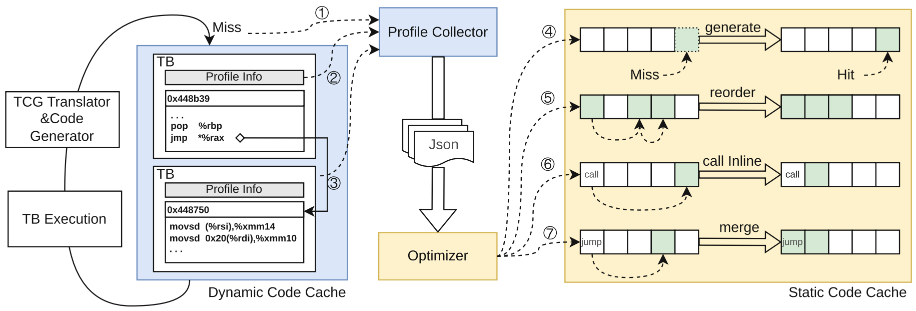


Fig. 5. The design of Multi-stage Feedback.

variables, leading to a reduced overhead of the LLVM mem2reg pass. The cache stores the most recent value of the virtual register. The value is written back to corresponding LLVM IR stack variable, only when encountering branch instructions. This approach reduces the cost of the LLVM mem2reg pass within each translation unit.

To ensure the correctness of register mapping at the entry and exit blocks of each translation unit, it is necessary to prevent the compiler from scheduling the LLVM IR instructions responsible for these mappings. To achieve this, we added priority flags to these instructions, guiding the compiler's scheduling algorithm accordingly. In MFHBT-LA, the read operations of physical registers at the entry block of a translation unit are assigned the highest priority, while the write operations of physical registers at the exit block are assigned the lowest priority. This approach effectively resolves the issue and guarantees the correctness of register mappings.

It is important to note that the register mapping mechanism does not affect the compiler's usage of physical registers or the quality of generated code, even when the number of guest registers is similar to that of host registers. Firstly, the selection of mapped registers is customizable, allowing for mapping only frequently used guest registers. Secondly, the constraints of the register mapping mechanism only apply at the entry and exit of translation units and do not interfere with the compiler's register allocation within the translation units. Therefore, compared to a purely static register mapping approach, our solution can generate high-quality code.

5 Evaluation

Benchmarks. We select the CoreMark benchmark and ten subitems from the SPEC CPU2000 INT benchmark, excluding 175.vpr and 252.eon, to evaluate the performance of our translation system. The exclusion of 175.vpr and 252.eon is due to their intensive use of floating-point operations. However we do not optimize the floating-point and vector instructions and still rely on QEMU's helper mechanism. To avoid generating AVX instructions, we compile the selected benchmarks with the options "-mno-avx -fno-tree-vectorize".

Execution Platform. We conduct testing of our translation system on a Loongson 3A5000 machine [16] running Linux kernel version 4.19.0. The machine operates at a clock frequency of 2.5 GHz. The evaluation is conducted using QEMU version v7.0.93 and LLVM version v8.0.1.

5.1 Performance

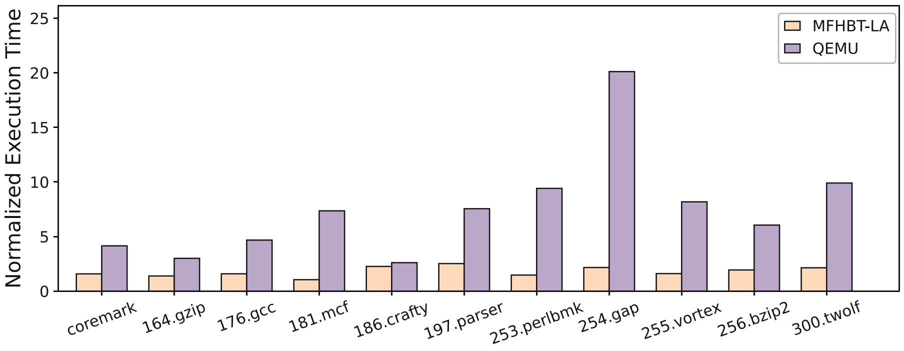


Fig. 6. Normalized execution time of MFHBT-LA and QEMU based on the native execution in CoreMark and SPEC CPU2000 INT.

We conduct a performance evaluation on three platforms: the native LA machine, QEMU, and MFHBT-LA in a stable state and calculate the normalized execution time of MFHBT-LA and QEMU based on the native program. The results, presented in Fig. 6, indicate a notable improvement in performance. MFHBT-LA exhibits a performance increase of 2.63X in the CoreMark benchmark and 3.28X in the SPEC CPU2000 INT benchmarks compared to QEMU. These findings demonstrate the superior code quality achieved through LLVM optimization compared to the translated code generated by QEMU. Furthermore, MFHBT-LA exhibite only 1.68X slower than the native execution in the SPEC CPU2000 INT.

5.2 Execution Time

Figure 7 depicts the ratio of execution time spent on the code generated in the translators. Notably, MFHBT-LA exhibits a significantly larger proportion compared to HQEMU and QEMU. The statistical data is collected using perf, which may have a slight margin of error. However, it effectively demonstrates that offloading LLVM optimization to the static side significantly reduces translation time and increases execution time spent on the code generated, consequently enhancing system performance.

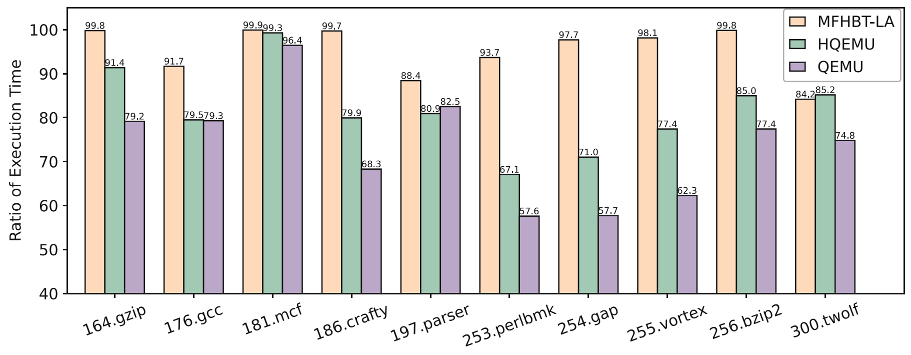


Fig. 7. Ratio of execution time to total time for the generated code of MFHBT-LA, HQEMU and QEMU.

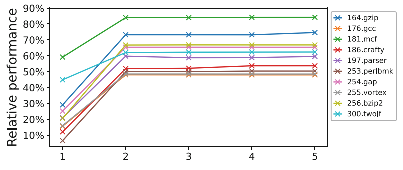
5.3 The Performance of Convergence

We demonstrate the performance of MFHBT-LA convergence no matter when the execution path is fixed or various among different executions.

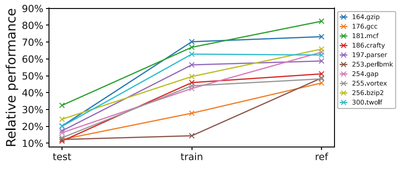
Figure 8a illustrates the relative performance of running the SPEC CPU2000 INT ref suites compared to the native program during five execution and feedback iterations. The performance reaches a stable state after two iterations, demonstrating fast convergence under a fixed execution path.

Figure 8b shows the relative performance compared to the native program of running the SPEC CPU2000 INT test, train, and ref suites in sequence. Although the three suites are various in execution path because of varying configurations and workloads, consistently improved performance is observed. This indicates that the feedback information and optimized code can be reused among different execution. This can be attributed to two main factors: (1) feedback information has a certain level of generality, resulting from factors like the limited nature of basic blocks, and (2) common execution paths exist among different runs.

Furthermore, we observe a strong resemblance between the relative performance of running ref suites in Fig. 8b and the relative performance in a stable



(a) The relative performance of running the SPEC CPU2000 INT ref suites compared to the native program during five execution and feedback iterations.



(b) The relative performance of running the SPEC CPU2000 INT test, train, and ref suites in sequence compared to the native program.

Fig. 8. The relative performance compared to the native program.

state in Fig. 8a. This finding further demonstrates that, even for programs with varying configurations, multiple executions can also lead to gradual convergence.

5.4 Memory Access Instruction Count

Figure 9 illustrates the memory access instruction count of the x86 native program, MFHBT-LA in stable state, HQEMU and QEMU. The MFHBT-LA achieves a substantial reduction in memory access, amounting to 81% and 65% when compared to QEMU and HQEMU, respectively, which is a significant contributing factor to its superior performance. This observation emphasizes the crucial role of register mapping in minimizing memory access.

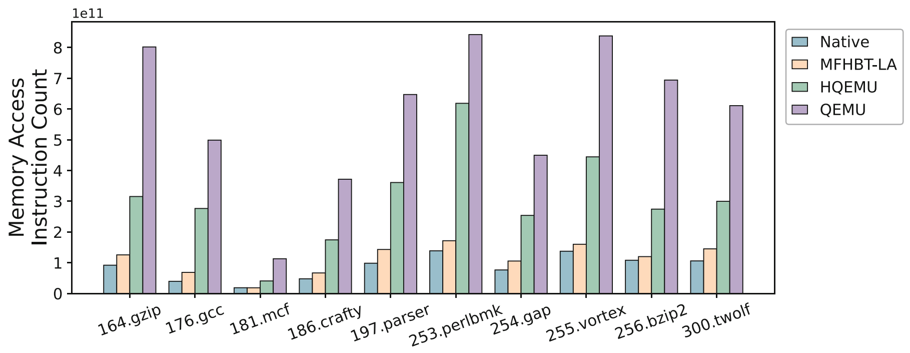


Fig. 9. The memory access instruction count for the x86 native program, MFHBT-LA, HQEMU and QEMU.

6 Discussion

Self-modifying Code. The accurate execution of self-modifying code in MFHBT is attributed to the adoption of QEMU's processing mechanism. We make slight modifications to the mechanism, resulting in the invalidation of both the dynamically generated code by QEMU and the code loaded from the AOT file when self-modification is detected. Consequently, QEMU will retranslate the code modified by the program during the subsequent execution.

Multi-architecture Support. The system is designed to be architecture independent, capitalizing on the support for multiple architectures offered by LLVM and QEMU. LLVM and QEMU both utilize Intermediate Representation (IR), TCG IR and LLVM IR, to represent program semantics, facilitating the generation of target code for various host architectures. In this work, adding a new architecture requires to implement translation procedures that convert guest instructions to LLVM IRs. Due to the optimization mechanisms provided by LLVM, the translation procedures only need to ensure correctness, rather than code quality, which accelerates the development speed of supporting a new ISA.

Real-World Applications. In addition to the benchmarks mentioned in the paper, we conduct experiments on various real-world applications, such as grep, awk, sed, and so on. Our prototype demonstrates satisfactory performance in these applications.

7 Related Work

Several conventional binary translation systems utilize a combination of binary translator and compiler. To reduce the runtime overhead of code optimization caused by compilers, HQEMU and HBT adopt different approaches. HQEMU [15], proposed by Hong et al., profiles hot traces in the execution thread, converts the TCG IR of these hot traces to LLVM IR, and implements additional optimizations in backend threads to generate superior code. This approach leverages the availability of multicore platforms to reduce the runtime overhead of code optimization. HBT [23], proposed by Shen et al., is a hybrid binary translation system based on LLVM that combines the benefits of SBT and DBT. The system offloads part of the compilation optimization cost to the SBT. Li et al. perform work to improve LLVM IR generation speed. They proposed CrossDBT [19], directly lift guest binary code to LLVM IR to avoid the additional transform overhead and local information loss compared to translate guest code to TCG IR first.

Some research works on combining static and dynamic translator to enhance the performance of the binary translation system [13,14]. Chernoff designed and implemented a binary translation system, FX!32 [8], to reduce the overhead of translation in the dynamic side. When the program execution, an AOT file generated by the static side will be loaded and executed, thus improving the performance of the system. Guan et al. proposed an approach to software cache optimization. In this approach, they rearrange the software cache layout by collecting profile information and translated code, so that the most frequently executed parts are at the top of the cache [13].

8 Conclusion

In binary translation, optimizing code quality while minimizing translation cost is crucial for improving performance. In this paper, we introduce a hybrid binary translation system with multi-stage feedback that optimizes translated code using the compiler and provides feedback to SBT based on program information from DBT. Additionally, we propose a register mapping mechanism in the compiler that reduces memory access instructions by 81% compared to QEMU in the SPEC CPU2000 INT benchmark. Our prototype, MFHBT-LA, improves performance by 3.28 times compared to QEMU in the same benchmark. As part of future work, we will optimize floating-point and vector instructions using the method proposed in this paper. Furthermore, we plan to investigate additional optimization techniques customized for specific architectures.

Acknowledgment. We would like to thank all the anonymous reviewers for their helpful comments and suggestions. This project is funded by the 2022 National Key Research and Development Program "Security Protection Technology for Distribution Network Key Information Infrastructure" Project 3 Distribution Network Computing Equipment Security Enhancement Technology Research and Localization Development (Project No. 2022YFB3105103).

References

1. Altman, E.R., Kaeli, D., Sheffer, Y.: Welcome to the opportunities of binary translation. Computer **33**(3), 40–45 (2000)
2. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: a transparent dynamic optimization system. In: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, pp. 1–12 (2000)
3. Bellard, F.: QEMU, a fast and portable dynamic translator. In: USENIX Annual Technical Conference, FREENIX Track, California, USA, vol. 41, p. 46 (2005)
4. Bezzubikov, A., Belov, N., Batuzov, K.: Automatic dynamic binary translator generation from instruction set description. In: 2017 Ivannikov ISPRAS Open Conference (ISPRAS), pp. 27–33. IEEE (2017)
5. Borin, E., Wu, Y.: Characterization of DBT overhead. In: 2009 IEEE International Symposium on Workload Characterization (IISWC), pp. 178–187. IEEE (2009)
6. Chen, J.Y., Yang, W., Hsu, W.C., Shen, B.Y., Ou, Q.H.: On static binary translation of ARM/Thumb mixed ISA binaries. ACM Trans. Embed. Comput. Syst. (TECS) **16**(3), 1–25 (2017)
7. Chen, W., Shen, L., Lu, H., Wang, Z., Xiao, N.: A light-weight code cache design for dynamic binary translation. In: 2009 15th International Conference on Parallel and Distributed Systems, pp. 120–125. IEEE (2009)
8. Chernoff, A., et al.: FX! 32: a profile-directed binary translator. IEEE Micro **18**(02), 56–64 (1998)
9. Cifuentes, Malhotra: Binary translation: static, dynamic, retargetable? In: 1996 Proceedings of International Conference on Software Maintenance, pp. 340–349. IEEE (1996)
10. Duesterwald, E., Bala, V.: Software profiling for hot path prediction: less is more. ACM SIGARCH Comput. Archit. News **28**(5), 202–211 (2000)
11. Engelke, A., Okwieka, D., Schulz, M.: Efficient LLVM-based dynamic binary translation. In: VEE 2021, pp. 165–171. Association for Computing Machinery, New York (2021)
12. Fu, S.Y., Hong, D.Y., Wu, J.J., Liu, P., Hsu, W.C.: SIMD code translation in an enhanced HQEMU. In: 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS), pp. 507–514. IEEE (2015)
13. Guan, H., et al.: A dynamic-static combined code layout reorganization approach for dynamic binary translation. J. Softw. **6**(12), 2341–2349 (2011)
14. Guan, H., Zhu, E., Wang, H., Ma, R., Yang, Y., Wang, B.: SINOF: a dynamic-static combined framework for dynamic binary translation. J. Syst. Archit. **58**(8), 305–317 (2012)
15. Hong, D.Y., et al.: HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores. In: Proceedings of the Tenth International Symposium on Code Generation and Optimization, pp. 104–113 (2012)
16. Hu, W., Wang, J., Gao, X., Chen, Y., Liu, Q., Li, G.: Godson-3: a scalable multicore RISC processor with x86 emulation. IEEE Micro **29**, 17–29 (2009)

17. Inoue, H., Hayashizaki, H., Wu, P., Nakatani, T.: A trace-based Java JIT compiler retrofitted from a method-based compiler. In: International Symposium on Code Generation and Optimization (CGO 2011), pp. 246–256. IEEE (2011)
18. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, CGO 2004, pp. 75–86. IEEE (2004)
19. Li, W., Luo, X., Zhang, Y., Meng, Q., Ren, F.: CrossDBT: an LLVM-based user-level dynamic binary translation emulator. In: Cano, J., Trinder, P. (eds.) Euro-Par 2022. LNCS, vol. 13440, pp. 3–18. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-12597-3_1
20. Liu, I.C., Wu, I.W., Shann, J.J.J.: Instruction emulation and OS supports of a hybrid binary translator for x86 instruction set architecture. In: 2015 IEEE 12th International Conference on Ubiquitous Intelligence and Computing and 2015 IEEE 12th International Conference on Autonomic and Trusted Computing and 2015 IEEE 15th International Conference on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), pp. 1070–1077. IEEE (2015)
21. Payer, M., Gross, T.R.: Generating low-overhead dynamic binary translators. In: Proceedings of the 3rd Annual Haifa Experimental Systems Conference, pp. 1–14 (2010)
22. Shen, B.Y., Chen, J.Y., Hsu, W.C., Yang, W.: LLBT: an LLVM-based static binary translator. In: Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, pp. 51–60 (2012)
23. Shen, B.Y., You, J.Y., Yang, W., Hsu, W.C.: An LLVM-based hybrid binary translation system. In: 7th IEEE International Symposium on Industrial Embedded Systems (SIES 2012), pp. 229–236. IEEE (2012)
24. Shi, H., Wang, Y., Guan, H., Liang, A.: An intermediate language level optimization framework for dynamic binary translation. ACM SIGPLAN Not. **42**(5), 3–9 (2007)
25. Spink, T., Wagstaff, H., Franke, B., Topham, N.: Efficient code generation in a region-based dynamic binary translator. In: Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems, pp. 3–12 (2014)
26. Ung, D., Cifuentes, C.: Dynamic re-engineering of binary code with run-time feedbacks. In: Proceedings Seventh Working Conference on Reverse Engineering, pp. 2–10. IEEE (2000)
27. Weiwu, H., et al.: Loongson instruction set architecture technology. J. Comput. Res. Dev. **60**, 2–16 (2023). (in Chinese)
28. Wenzl, M., Merzdovnik, G., Ullrich, J., Weippl, E.: From hack to elaborate technique-a survey on binary rewriting. ACM Comput. Surv. (CSUR) **52**(3), 1–37 (2019)