LAST: An Efficient In-place Static Binary Translator for RISC Architectures

Yanzhi Lan[1,2], Qi Hu[1,2], Gen Niu[1,2], Xinyu Li[1,2], Liangpu Wang[1,2], and Fuxin Zhang[1,2](✉)

[1] State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
{lanyanzhi22b,huqi20s,niugen18z,lixinyu20s,wangliangpu21s, fxzhang}@ict.ac.cn

[2] University of Chinese Academy of Sciences, Beijing, China

Abstract. The lack of software has been a persistent issue for emerging instruction set architecture (ISA). To overcome this challenge, binary translation has emerged as a widely adopted solution, enabling programs written for older ISA to run on new ones. In the past, dynamic binary translation (DBT) was commonly utilized for software migration, but this technique required dynamic translation and often suffered from suboptimal efficiency. In contrast, Static binary translation (SBT) is an offline technique for translating binary code without runtime translation overhead. Existing SBT systems always employ address mapping tables to handle the address relocation problem, but this approach introduces performance overhead and leads to issues with indirect jump correctness. To address these limitations, we propose a novel static in-place instruction translation method for reduced-instruction set computing (RISC) architectures. This method ensures that the address of the guest program remains unchanged after translation, leveraging the regular length of various RISC instructions. We have implemented this method in a portable SBT tool called LAST, specifically designed to run MIPS or RISCV programs on the LoongArch platform. Based on the SPEC CPU2000 benchmark results, LAST achieves over 80% performance compared to the native LoongArch program, demonstrating its effectiveness and efficiency.

Keywords: Static Binary Translation · In-place instruction translation · instrumentation

1 Introduction

Binary translation enable software of one architecture to execute on a hardware platform of another architecture. This technology has a wide range of application scenarios, such as fast software simulation [4,9,13,22], program runtime analysis [6,10,20], debugging [11,14,21], and dynamic optimization [5].

Supported by The National Key Research and Development Program of China under grant number 2022YFB3105103.

Dynamic binary translation has played a prominent role in software migration endeavors in the past decades. As diverse instruction sets continue to evolve, a multitude of exceptional dynamic binary translation systems have emerged, showcasing the advancements in this field. Notable examples include IA-32 EL, which enables the execution of IA-32 applications on IA-64 processor family systems [2]. And Rosetta2, which facilitates the migration of x86 executables to the ARM platform [1]. Additionally, QEMU is a fast and portable dynamic translator, which support multiple guest and host ISAs [4]. However, it is important to note that dynamic binary translation often incurs additional performance and memory overhead.

Static binary translation not only does not need to translate at execution time but also can use larger-scale optimization methods, thus it can often achieve higher execution efficiency. SBT can often achieve higher execution efficiency because no real-time translation is required, which can be used to complete software migrations efficiently.

The Address relocation problem is caused by instruction expansion during translation, breaking the original indirect jump relationship, critically affecting the efficiency of static binary translators. The correctness of the jump relationship in the guest program is guaranteed by the compiler of the guest platform, but some address information will be lost during the compilation process, which makes it particularly difficult for the binary translator to reconstruct the jump relationship of the translated program. For direct jumps, the translator can easily calculate the new jump target through the offset value in the instruction. But for indirect jumps, their targets are unknown during translation. So address mapping tables is used to look up the targets by guest address at the runtime. However, this lookup table can introduce extra overhead, we will discuss the overhead in Sect. 2.2.

Instruction instrumentation technology is widely employed in various binary analysis tools, enabling the modification of the execution flow of the original program. These tools incorporate instrumentations into the program to facilitate statistical analysis and program debugging [23]. However, it should be noted that these tools are typically limited to programs within the same ISA. One factor contributing to this limitation is the variability in instruction lengths across different ISAs. For instance, the X86 instruction set utilizes variable-length instructions, whereas the MIPS instruction set adheres to a fixed-length format.

Over the past few decades, RISC architecture has witnessed remarkable advancements, giving rise to prominent instruction sets such as ARM, MIPS, RISCV, and LoongArch. These instruction sets have regular instruction encoding, which facilitates efficient addressing and decoding of instructions. Introduced by Loongson Technology in 2020, LoongArch is a new RISC instruction set that incorporates state-of-the-art advancements in instruction system design [26]. It offers a favorable environment for the development of low-power, high-performance CPUs [17–19, 27]. With a fixed instruction length of 32 bits and a regular encoding format, LoongArch ensures simplicity and ease of instruction handling. The design of LoongArch also prioritized software compatibility, its basic software, such as the Linux kernel, GCC compiler, and QEMU simulator, has already been successfully integrated into the community.

In this paper, we propose a novel approach called in-place instruction translation, which enables the preservation of address relationships in the translated guest program, eliminating the need for address relocation. This innovative technique is implemented in a new SBT tool named LAST, which currently supports the translation from MIPS or RISCV to LoongArch, called LASTM and LASTR. Through extensive evaluation using the SPEC CPU2000 benchmark, LASTM demonstrates remarkable performance. It achieves over 80% of the program efficiency of the native LoongArch platform. These results showcase the effectiveness and efficiency of our proposed in-place instruction translation approach in the context of static binary translation.

The main contributions of this paper are as follows:

- We propose an in-place instruction translation method in binary translation systems that can efficiently solve the address relocation problem encountered in the mutual translation of RISC architectures. This method can effectively reduce runtime translation overhead and improve system performance.
- We implement and evaluate LAST, demonstrating the effectiveness of the in-place instruction translation method. Our experiments show that in-place instruction translation can solve the address relocation problem in binary translation with minimal overhead.
- This paper discuss the potential of this instruction translation method to be efficiently applied to the mutual translation of other RISC architectures, further expanding its applications beyond LoongArch.

The organization of the rest of this paper is as follows. Section 2 introduces static binary translation, address relocation problems, and overhead in SBT. Section 3 describes the design of LAST, and Sect. 4 shows the implementation details of LAST. Section 5 analyses some experimental results. Section 6 concludes this paper.

2 Background

This section provides a concise overview of static binary translation and highlights its key challenges, focusing on the main performance overheads associated with this technique.

2.1 Static Binary Translation

Binary translation systems fall into two general categories: static binary translator and dynamic binary translator. Figure 1 shows the difference between dynamic binary translation and static binary translation. The dynamic binary translator dynamically translates instructions during the execution of the native program. While the static binary translator converts the original program to the target program offline, and the translated program executes without the assistance of the binary translator. Compared to DBT, SBT is more convenient and efficient.

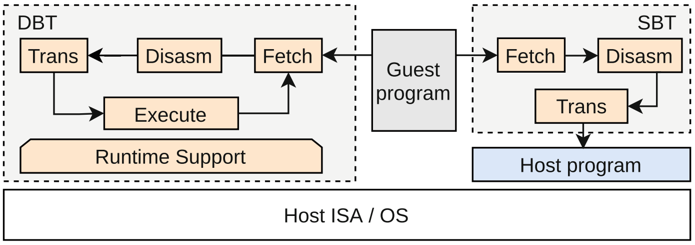


Fig. 1. The Difference between DBT and SBT

UQBT [8] is a versatile SBT tool that utilizes an intermediate language called HRTL to translate source binaries. HRTL can be further transformed into target binary assemblers, enabling compatibility with multiple platforms. However, UQBT still relies on a runtime interpreter to handle indirect register calls that cannot be determined during static translation.

LLBT [24,25] is a portable SBT tool specifically designed for translating ARM binaries to various target platforms. It employs LLVM IR (Intermediate Representation) as an intermediate representation and leverages the LLVM compiler infrastructure to retarget the LLVM IR to different ISAs. This approach significantly enhances the quality of the generated code. Nonetheless, LLBT still requires an address mapping table to effectively handle indirect jumps.

SBT can cause the address of the guest program to change due to instruction expansion during translation, breaking the original indirect jump relationship. This leads to the address relocation problem, which is described in detail in Sect. 2.2.

2.2 Address Relocation Problem in SBT

The best situation is that SBT can complete the instruction translation without any code expansion, thus avoiding updating the branch target address [28]. Nevertheless, it is obvious that one-to-many translations always exist, which requires extra space to store the extra instructions. So the branch instructions, especially indirect branch instructions, need to change their target address to avoid the incorrect execution flow.

Figure 2 show the Address Relocation Problem. Because the range of immediate numbers that can be used in the LoongArch is smaller than that in the MIPS, two additional instructions are required in Fragment L1. But the address of LABEL L3 is not changed, which may misleads some branch instructions into jumping to the wrong address, like instruction jr ra.

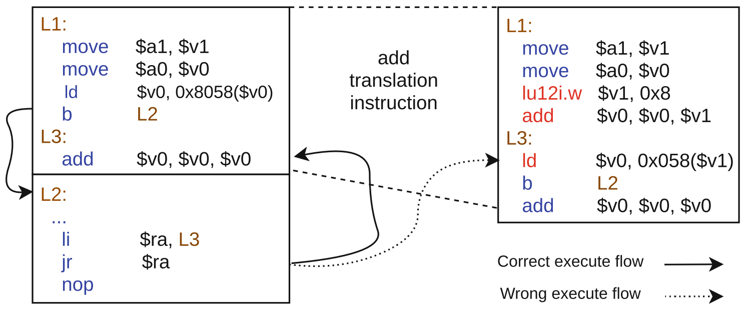


Fig. 2. The Address Relocation Problem

These problems can be easily solved for direct branches, but updating the target address for indirect branches sometimes is difficult [16]. To tackle this problem, Killian's Pixie uses a translation table in which all indirect branches need to find their target addresses [7].

Currently, the predominant approach for addressing this issue in static binary translation systems is to employ an address hash table [25]. However, challenges related to code discovery and performance persist. Please see Sect. 2.3 for further details on these challenges.

2.3 Performance Overhead in SBT

The performance of binary translation is significantly influenced by the disparities between the Guest ISA and the Host ISA. To achieve optimal performance in SBT tools, two critical issues must be addressed, as they have a substantial impact on the translation process.

- Instruction expansion. The semantics of instructions on different architectures directly leads to the inevitable instruction expansion during binary translation, which increases the number of dynamic instructions and affects efficiency.
- Indirect branch handling overhead. Caused by binary code expansion, one-to-many translations can affect the address of the original instruction. Therefore, the translator needs to correctly handle the modified jump relationships, especially indirect branches.

Instruction expansion is a common outcome of disparities between instruction sets, but the overhead associated with indirect jumps can be minimized. Numerous remarkable studies have been conducted to mitigate the impact of indirect jumps. For instance, Amanieu extensively examined various types of indirect jumps and implemented optimizations tailored to specific contextual scenarios in dynamic binary translation [12].

However, static binary translators do not actively participate in program execution and, as a result, lack the capability to dynamically handle indirect jumps during runtime. To address this limitation, a common approach is to employ a address mapping table within static binary translation systems [3]. However, this method necessitates hash table queries, thereby introducing additional overhead.

Algorithm 1: Lookup Indirect JMP Target

```
input : GPC
hash = HASH($GPC$);
HPC = Address_Mapping_Table[hash];
if $HPC$ != $NULL$ then
    jmp to HPC                       // hit the target;
else
    return error                     // Lookup error;
end
```

The address mapping table serves as a repository where the guest program counter address (GPC) is stored as the key, and the corresponding host program counter address (HPC) is stored as the value. The lookup process is outlined in Algorithm 2. During runtime, the GPC is hashed to generate an index that corresponds to the HPC stored in the address mapping table. This HPC was previously stored during the static binary translation (SBT) process. Consequently, the HPC associated with the given GPC can be swiftly retrieved based on the hash. Additionally, it is crucial to compare the guest addresses within the lookup table to ensure that the hash algorithm does not generate any conflicts.

There is no doubt that Algorithm 2 is reliable and efficient. However, the production of instructions required to calculate the address hash, fetch data from the lookup table and determine if a hit has taken place still imposes a considerable performance cost. In SPEC2000, as shown in Fig. 3, indirect jumps account for approximately 1.27% of the total number of instructions, resulting in approximately 15% performance loss due to its `1:12` instruction inflation.

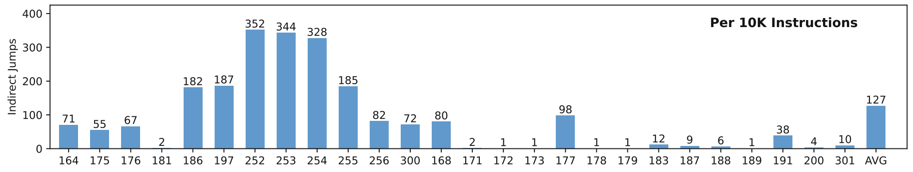


Fig. 3. Number of indirect jumps per 10K instructions

Furthermore, indirect jumps pose another huge challenge for static binary translation systems, as those jump entries can be difficult to fully identify. If the address entries, such as the switch-case jump table, cannot be recognized fully, some parts of the program are not being executed correctly because the address mapping table will lose some entry mappings.

3 Design

3.1 Overview

The design of the LAST architecture is shown in Fig. 4, which mainly includes the following modules:

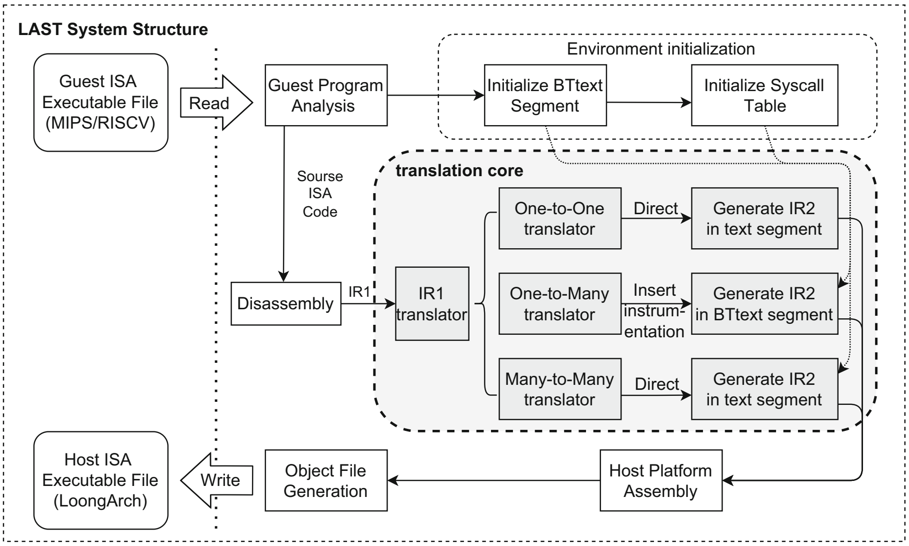


Fig. 4. The overall design of the LAST

Guest Program Analysis Module. This module accepts the origin binary executable file, parses the ELF format and maps each program segment. Then the segments will be recorded and reordered, making it easier for translator to analyze, manage, and provide essential information to other modules.

Environment Initialization Module. This module configures the basic structure related to translation, including initialization translated code segment and system call table. The translated code segment is the location for the one-to-many translated code, which will be written to the translated executable file. The system call table is used to convert different system call numbers between original and translated programs.

Disassembly Module. This module disassembles the original code segments and converts each instruction into internal IR-GUEST data.

Translation Module. This module will use the IR-GUEST data to classify different types of instructions, and then the corresponding translation function will translate them to the IR-HOST data.

In the translator core, the instructions are divided into three categories: *one-to-one*, *one-to-many* and *many-to-many* translation. Each translator puts the translated instruction in the original position, and in addition, one-to-many translator will place the extra instructions in the translated code segment.

Host Platform Assembly Module. This module integrates and assembles the IR-HOST data to generate the binary code and store it in the memory.

Object File Generation. This module will reorganize the segments (segment of the original file and the new segments generated by the translator), fill in the necessary ELF file header and write to the target file.

3.2 In-place Instruction Translation Design

In this study, we propose an innovative in-place instruction translation method that preserves the original addresses of each instruction in the program while minimizing the overhead associated with branches. This method draws inspiration from the principles employed in instruction instrumentation binary transformation tools but is tailored specifically for Cross-ISA static binary translation.

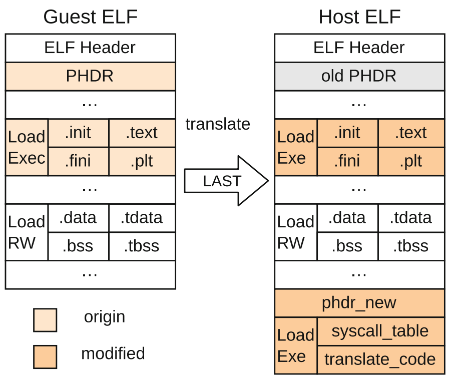


Fig. 5. Modifications of LAST to ELF Sections

LAST will generate an additional code segment to store translated binary codes and system call table. Figure 5 shows the difference between original and translated ELF file.

For the translation of each instruction, we need to consider two cases, the *one-to-one* and *one-to-many* translation. We also consider some optimization by using *many-to-many* translation.

– **One-to-one translation.** If the guest platform instruction can be converted into host platform instructions one-to-one, it is simple to place the translated instruction at the original address.
– **One-to-many translation.** Some instructions are complex and need to use multiple instructions to translate them. We put these translated binary codes at the additional code segment and then replace the original instruction with a direct jump instruction which can jump to the translated fragment.
– **Many-to-many translation.** Some instructions are used as a pattern on the guest platform to achieve a certain function. Usually, the host platform also has an instruction pattern with equal length to achieve the same function. In this case, the host instruction pattern can equivalently replace the guest instruction pattern.

The proposed tool, LAST, effectively addresses the challenges posed by indirect jumps through its in-place instruction translation mechanism. Section 4.2 of the paper will provide comprehensive implementation details, offering a deeper understanding of how LAST successfully mitigates the issues related to indirect jumps.

3.3 System Call Design

To solve the issue of system call incompatibility, LAST employs a system call table for system call conversion.

Unlike binary translators that "wrap" system calls by modifying call numbers and handling different structures [15], LAST uses the system call table to handle these issues, which is particularly useful in static in-place binary translation where dynamic interception and processing of system calls is challenging.

Algorithm 2: Handle Syscall By Syscall Table

```
input  : Guest Syscall Number
hash = HASH(Guest Syscall Number);
Host Syscall entry = Address_Mapping_Table[hash];
if Host Syscall entry != NULL then
    jmp to Host Syscall entry                // hit the target;
    covert Guest Arguments to Host Arguments;
    do Host Syscall;
    covert Host return Arguments to Guest return Arguments;
    return
else
    return error                             // Unsupport Syscall;
end
```

During the translation process, the initial step of LAST involves the insertion of the system call table into the translation code segment, as depicted in Fig. 5. The role and functionality of the system call table are further elucidated in

Algorithm 2. It facilitates several key functions: locating the appropriate table item based on the Host Syscall number, performing the necessary conversion from Guest ABI to Host ABI to enable system call invocation and kernel entry, and addressing any variations in return values that may arise between different architectures, ensuring a seamless transition back to the normal execution flow. For a more comprehensive understanding of the implementation specifics, please refer to Sect. 4.3.

3.4 ISA-Level Support

LAST utilizes some of the binary translation optimizations provided by the LoongArch instruction set, which are optional. These optimizations are provided as optional enhancements, and their detailed descriptions are presented below.

To address two efficiency issues in translation, LoongArch has designed binary translation support into its ISA. The first issue is register shortage, which may occur due to register mapping during binary translation. To solve this, LoongArch adds four new scratch registers (SCRs) alongside the 32 general-purpose registers (GPRs). These SCRs can interact with GPRs through data move instructions and are used as temporary registers. The second issue is the jump range limit. In-place instruction translation, as used in LAST, requires jumping to the translated code block. To address the problem of the limited jump range of the direct jump instruction, a jump-and-link instruction using SCRs is also added. By setting the value in one of the SCRs as the address of the translated code block, it is possible to jump to the translated code block effectively and return to the original instruction conveniently.

In LAST, the SCR is primarily used in the following situations: first, when the number of registers is insufficient during instruction translation, the SCR register is used as a temporary register instead of using a virtual register in memory. Second, when LAST needs to interrupt the current execution flow, but cannot jump for a long distance, the SCR is used as the address register for a long-distance indirect jump, thus reducing the storage and recovery of the source register.

If the host system does not support SCRs, LAST employs a strategy to identify the least frequently used registers in the guest program. These registers are then transformed into memory accesses, allowing the freed registers to be utilized for the same functionalities as the SCR registers mentioned earlier. As a result, even in the absence of SCR support on the host, the aforementioned challenges can still be addressed using additional instructions. However, it is important to note that this approach may lead to a potential loss of efficiency.

4 Implementation

4.1 Register Mapping

Register mapping, which binds the guest platform's registers to the host platform's registers, is an important part of binary translation, directly affecting the execution efficiency of binary translators.

Most binary translators, such as QEMU, use translation blocks (TBs) as base units and perform dynamic register allocation in each TB. This dynamic register allocation is convenient for design but tends to cause data transfer overhead. Regarding the implementation of LAST, it adopts a global static register mapping approach. Specifically, we illustrate this with LASTM as an example, and the corresponding mapping rules are presented in Table 1.

Table 1. Register mapping in LASTM

MIPS		LoongArch		LASTM
num	name	num	name	
0	zero	0	zero	zero
1	at	19	t7	tr_at
2, 3	v0, v1	17, 18	t5, t6	tr_v0, tr_v1
4–11	a0–a7	4–11	a0–a7	tr_a0–tr_a7
12–15	t0–t3	12–15	t0–t3	tr_t0–tr_t3
16–23	s0–s7	23–30	s0–s8	tr_s0–tr_s7
24, 25	t8, t9	16, 20	t4, t8	tr_t8, tr_t9
26, 27	k0, k1	21, 22	tp, fp	tmp
28	gp	2	gp	tr_gp
29	sp	3	sp	tr_sp
30	s8/fp	31	s8	tr_s8
31	ra	1	ra	tr_ra
hi, lo	hi, lo	-	scr2, scr3	tr_hi, tr_lo
-	-	-	scr0, scr1	tmp

The main reason for using these mapping rules is the difference in the definition of the ABI and the differences in hardware. For example, register 31 is used as the Return Address (RA) in MIPS, while the RA in LoongArch is register 1. In addition, when the branch predictor supports RAS, it is important to map the RA register of target planform to the host RA register, reducing the overhead of function returns. Also, four scratch registers (SCRs) are designed in the LoongArch which provide additional temporary registers for binary translation. LAST maps SCR2 and SCR3 to the HI and LO registers in MIPS, and SCR0 and SCR1 are used as temporary registers.

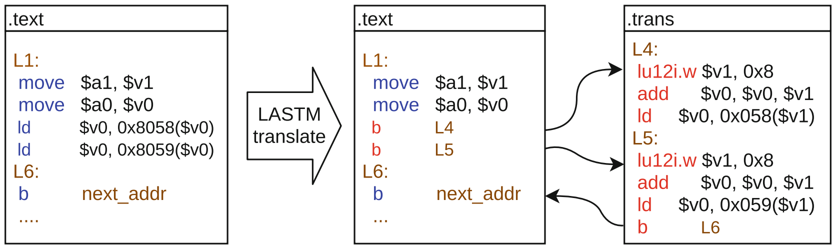
In addition, LAST takes into consideration the scenario where the number of guest registers exceeds that of the host registers. In this case, the translator can still implement the translation process by using memory as virtual registers. The translator keeps track of the frequency of register usage, and saves and restores the less frequently used registers, treating them as temporary registers. These marked temporary registers can be used when necessary by loading them from memory, where they were saved earlier. It is worth noting that this load/restore overhead is unavoidable. This case *NumRegs(Host)* $>=$ *NumRegs(Guest)* requires additional loading and restoring in any translator.

4.2 In-place Instruction Translation

Instruction translation is the critical part of LAST, which is related to the efficient execution of the translated program.

LAST's translator disassembles each input instruction and stores the detailed information in the IR. Then the translator will identify the classification of the instruction from the IR and translate them using different translation functions. LAST classifies instruction translations into three types.

One-to-One Translation. Both the host ISA and the guest ISA are RISC architectures, the behavior of the instructions is relatively similar. So some guest instructions can be translated into host instructions one-to-one. For such instructions, LAST can replace them at the original addresses.



```
.text
L1:
 move   $a1, $v1
 move   $a0, $v0
 ld     $v0, 0x8058($v0)
 ld     $v0, 0x8059($v0)
L6:
 b      next_addr
 ....
```

```
.text
L1:
 move   $a1, $v1
 move   $a0, $v0
 b      L4
 b      L5
L6:
 b      next_addr
 ...
```

```
.trans
L4:
 lu12i.w $v1, 0x8
 add     $v0, $v0, $v1
 ld      $v0, 0x058($v1)
L5:
 lu12i.w $v1, 0x8
 add     $v0, $v0, $v1
 ld      $v0, 0x059($v1)
 b       L6
```

Fig. 6. One-to-many translation in LAST

One-to-Many Translation. Because of the differences between the host ISA and the guest ISA, some instructions require one-to-many translation, such as system call instruction and some instructions with 16-bit immediate. In this case, the translator will translate this instruction and put these translated codes in an additional code segment. At the same time, the original instruction will be replaced by a direct jump whose target address is this additional code segment. For example, Fig. 2 will be translated into the case of Fig. 6. The instruction *ld $v0,0x8058($v0)* is replaced by *b L4*, and the segment *.trans* is used to store these "one-to-many translated code". At the end of the translated code, a branch instruction *b L6* will be added to return to the original control flow.

In addition, if multiple consecutive instructions require one-to-many translation, they need to jump to the translated code segment only once. Thus, *b L4* need not to return next instruction.

Many-to-Many Translation. Compilers often combine several instructions as patterns. If we translate these instructions one by one, it may result in multiple *one-to-many translations*. However, if we use *many-to-many translation* to translate this instruction pattern, we often get good results. LAST can recognize these instruction patterns and translate them into instruction sequences with the same function.

For instance, in the case of MIPS, the instruction sequence $Lui + Ori$ is utilized to load 32-bits immediate values. Conversely, in LoongArch, the corresponding instruction has a different immediate width. Nonetheless, both architectures provide instructions capable of loading 32-bits immediate values. In such scenarios, LAST treats the combination of $Lui+Ori$ as a unit and replaces them with suitable instructions from the host instruction set.

4.3 System Call Handling

The translated programs cannot execute on the platform directly due to the differences between the host and guest operating systems. So, System calls must be handled in binary translation.

LAST stores a system call table in the translation code segment, which is used to handle system call differences between the guest Kernel and the host Kernel. Figure 7 shows the execution flow of the system call. Whenever the guest program needs to run a system call, the program first jumps to the header of the system call table, where a piece of code is stored for preprocessing. Then, LAST will use the guest system call number as an index to find the handler's entry in the system call table and jump over to execute the handler function. In these handlers, LAST will convert the system call parameters, including system call numbers and some structures in memory.

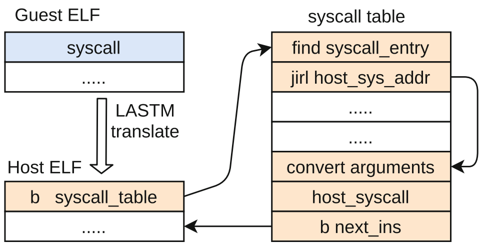


Fig. 7. Syscall Execution Flow in LAST

Another thing we need to consider is that there may be some special system calls that exist only in the guest platform and cannot be implemented by the host system calls. It is difficult to implement a non-existent system call in user mode. To handle those system calls, LAST uses system call simulation by employing other host system calls to mimic the function of the guest system call. For instance, the CLONE system call in MIPS can be emulated by the FORK system call in LoongArch.

4.4 Delay Slot

Due to historical reasons, the MIPS has designed the delay slot. However, delay slots are not available in the LoongArch, which leads to additional processing in LASTM.

In general, the instruction in the delay slot and the branch instruction have no data dependencies and can be easily translated by swapping the positions of two instructions.

When data dependencies exist, the relationship between the instructions needs to be handled with care, as depicted in Fig. 8. The daddiu instruction is an example of a data-dependent operation instruction, and simply changing the order of its execution can cause the beq instruction to execute incorrectly, resulting in program errors. To solve this error, LASTM need to translate according to the following steps. First, the value in the dependent register needs to be copied to a temporary register. Then, the delay slot instruction is executed. Finally, the branch instruction where the dependent register is replaced by the temporary register will be executed.

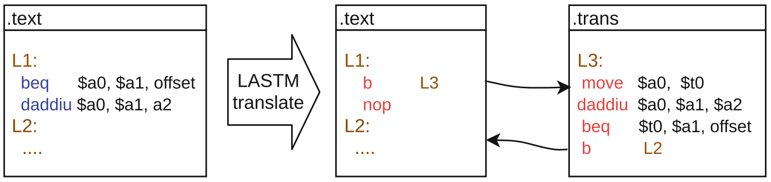


Fig. 8. Data Dependencies Delay Shot Handling

5 Evaluation

5.1 Evaluation Setup

Table 2. Evaluation Platform

	Loongson 3A4000	Loongson 3A5000
Architecture	GS464V	GS464V
ISA	**MIPS64 Release 5**	**LoongArch**
Compiler	gcc 8.3	gcc 8.3 (LoongArch)
Options	-Ofast -static	-Ofast -static
Frequency	**1.8 GHz**	**2.3 GHz**
L1 cache I/D	64 KiB	64 KiB
L2 cache	256 KiB	256 KiB
LLC	**8 MiB**	**16 MiB**

LAST offers the capability to convert MIPS or RISCV programs into LoongArch programs, which are referred to as LASTM and LASTR, respectively, for clarity. To evaluate its performance, we conducted tests in two distinct environments.

The term *MIPS* denotes the execution of native MIPS programs on the Loongson 3A4000 platform, while *LA* represents the execution of LoongArch programs on the Loongson 3A5000 platform. Although LAST supports RISCV-to-LoongArch translation, our evaluation was limited to the simulation environment for RISCV. Therefore, we were unable to perform actual chip tests for RISCV. The detail information about the experimental environment is shown in Table 2.

For evaluation purposes, we utilized the Coremark and SPEC CPU2000 benchmark testing programs. Coremark is specifically designed to assess the fixed-point performance of CPUs, and although it has a relatively small test scale, it serves as a suitable tool for evaluating the performance of binary translation. In addition to Coremark, we employed SPEC CPU2000 to obtain more detailed insights into the performance of binary translation. SPEC CPU2000 encompasses both fixed-point and floating-point testing, enabling a more comprehensive evaluation of performance details. All benchmarks were compiled by GCC version 8.3 with -Ofast as the optimization level.

The 3A5000 is an evolution of the 3A4000 and they share the same microarchitecture. There are three main differences, ISA, frequency, and LLC capacity. Other microarchitectural features are unchanged, so the programs behave very similarly on the 3A4000 and 3A5000. Therefore, for LAST, it is significant to compare the original MIPS programs on 3A4000 with the LAST-translated programs on 3A5000.

5.2 Performance

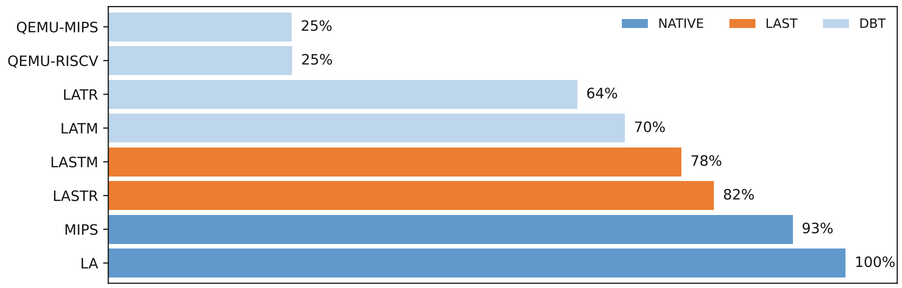


Fig. 9. Coremark Relative Ratio of the Scores-per-GHz The baseline is the scores-per-GHz of the *native LA*.

Figure 9 presents a performance comparison of various translators and native programs, using the score of the native LA as the baseline. It is obvious that LAST has significant performance advantages over other dynamic binary translator.

QEMU [4] is a commonly used binary translator in the industry that supports mutual translation of multiple architectures, but its efficiency is low due to the use of TCG as the intermediate code for translation. LATR and LATM are dynamic translators developed in the research group that use one-to-one instruction translation and special optimization for indirect jump, resulting in higher efficiency than QEMU. LASTR and LASTM represent LAST's translation of RISC and MIPS programs, respectively, and show much higher efficiency than dynamic translators, because In-place static translators do not generate indirect jump overhead and do not require translation time.

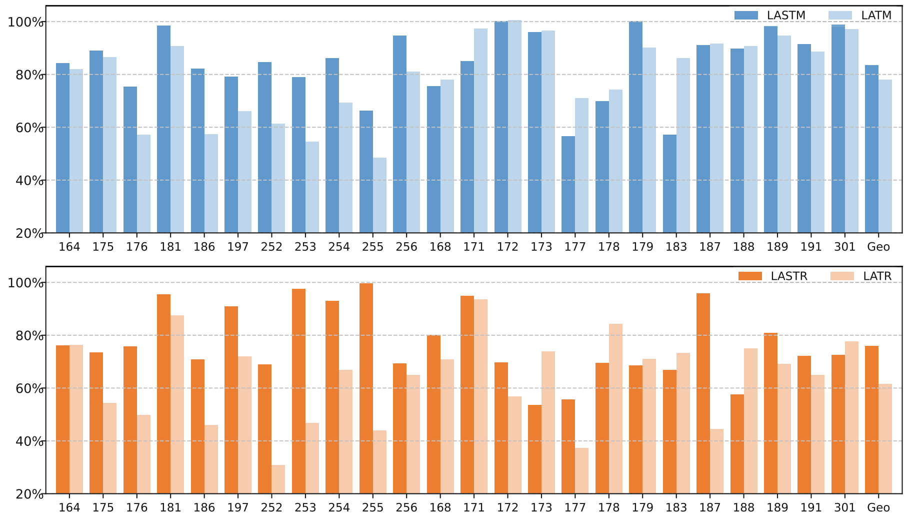


Fig. 10. SPEC2000 Relative Ratio of the Scores-per-GHz. The baseline is the scores-per-GHz of the *native LA*.

Figure 10 illustrates the SPEC2000 performance of LAST, with the native LA program serving as the baseline. On average, LASTM achieves over 80% performance compared to the *native LA* program, while LASTR achieves over 75% performance. Comparing these results with those of dynamic translators, it is evident that LAST demonstrates superior performance across most sub-items. The key factor contributing to LAST's higher performance is its ability to address the overhead of indirect jumps through the translation of interpolated instructions, without incurring the performance loss associated with dynamic translation techniques.

We further analyze the relevance between the instruction expansion ratio and the actual performance in LASTM. Figure 11 shows the relationship between the instruction expansion ratio and the execution time. Due to our translation rules, the number of *LASTM*'s dynamic executed instructions is always bigger than the number of *native MIPS*'s. So the value in the x-axis is always greater than 1.

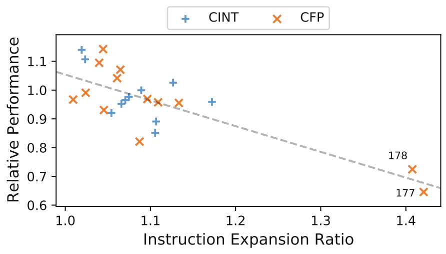


Fig. 11. Instruction Expansion Ratio vs Relative Performance. The x-axis is the ratio of the dynamic instruction count, which stands for the instruction expansion ratio. The y-axis is the ratio of the number of execution cycles, which stands for the relative performance.

The y-axis is the ratio of the number of execution cycles. We divide the number of *native MIPS*'s execution cycles by the number of *LASTM*'s.

Note that there are two points at the bottom-right corner. They are 177.mesa and 178.galgel, whose performance is less than 80%. The reason why their performance is such low is that their instruction expansion ratio is too high. They contain many instructions that can not be translated by the one-to-one translator.

5.3 Translation vs. Compilation

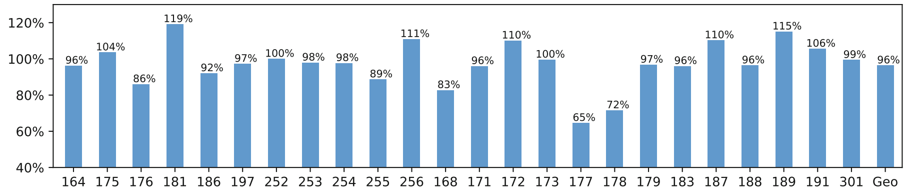


Fig. 12. SPEC2000 Relative Ratio of the Scores-per-GHz. The baseline is the scores-per-GHz of the *native MIPS*.

Figure 12 shows the SPEC2000 performance of LASTM. The result of *native MIPS* is the baseline. On average, the performance for SPEC benchmarks is 96%. Note that there are only two benchmarks whose performance is lower than 80%. And they both belong to the floating-point benchmark. In all, we can conclude from the result that LASTM almost does not lose performance compared to the native MIPS program. As shown in Fig. 10, the SPEC2000 performance based on the LA program is only 84%.

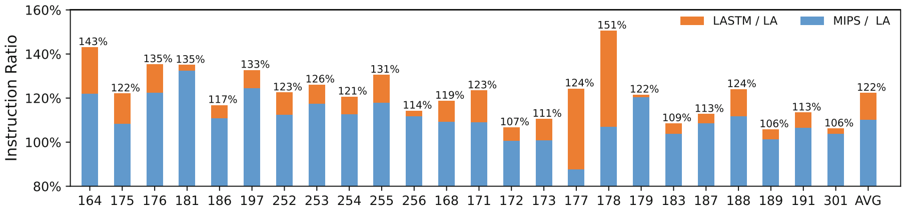


Fig. 13. Relative Ratio of Instruction Count. The baseline is the instruction count of *native LA*

However, this result is heavily influenced by compilers on different platforms. Figure 13 shows the relative ratio of the instruction count compared to the instruction count of the *native LA*. On average, for integer benchmarks, the instruction count of *native MIPS* is 17% more than that of *native LA* and *LASTM* is 27%. Since LASTM is translating MIPS instructions into LoongArch instructions, a sizeable proportion of the instruction expansion comes from the difference in the compiler.

In fact, the microarchitecture of 3A4000 and 3A5000 is nearly the same except for the size of the LLC. Comparing the performance of LASTM in 3A5000 with the performance of MIPS in 3A4000 while considering the difference in the hardware platform is a better way to describe the actual efficiency of our binary translator.

6 Conclusion

In conclusion, this paper introduces a novel approach, in-place static binary translation, which effectively addresses the challenge of address relocation and significantly improves the efficiency of SBT. The implementation of LAST on the LoongArch platform successfully converts MIPS or RISCV programs into the LA program. Experimental results demonstrate that while the translated program may experience a slight increase in direct jumps, it has minimal impact on efficiency. In fact, the translated program achieves over 80% performance compared to the native LA program, confirming the high efficiency of this approach. These findings highlight the effectiveness and potential of in-place static binary translation for efficient program translation and execution.

References

1. Apple: Rosetta. https://support.apple.com/en-us/HT211861
2. Baraz, L., et al.: IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium/spl reg/-based systems. In: 2003 Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-36, pp. 191–201 (2003). https://doi.org/10.1109/MICRO.2003.1253195

3. Bauman, E., Lin, Z., Hamlen, K.W.: Superset disassembly: statically rewriting x86 binaries without heuristics. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018. Internet Society, Reston (2018). wOS:000722005800038. https://doi.org/10.14722/ndss.2018.23300
4. Bellard, F.: QEMU, a fast and portable dynamic translator. In: Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, 10–15 April 2005, pp. 41–46. USENIX, Anaheim (2005). https://www.usenix.org/events/usenix05/tech/freenix/bellard.html
5. Bruening, D., Garnett, T., Amarasinghe, S.: An infrastructure for adaptive dynamic optimization. In: 2003 International Symposium on Code Generation and Optimization, CGO 2003, pp. 265–275 (2003). https://doi.org/10.1109/CGO.2003.1191551
6. Bruening, D., Garnett, T., Amarasinghe, S.: An infrastructure for adaptive dynamic optimization. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO 2003, pp. 265–275. IEEE Computer Society, USA (2003)
7. Chow, F.C., Himelstein, M.I., Killian, E., Weber, L.: Engineering a RISC compiler system. In: COMPCON, pp. 132–137 (1986)
8. Cifuentes, C., Van Emmerik, M.: UQBT: adaptable binary translation at low cost. Computer **33**(3), 60–66 (2000). https://doi.org/10.1109/2.825697
9. Cota, E.G., Bonzini, P., Bennée, A., Carloni, L.P.: Cross-ISA machine emulation for multicores. In: Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, pp. 210–220. IEEE Press (2017)
10. Cota, E.G., Carloni, L.P.: Cross-ISA machine instrumentation using fast and scalable dynamic binary translation. In: Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2019, pp. 74–87. Association for Computing Machinery, New York (2019). https://doi.org/10.1145/3313808.3313811
11. Cunha, M., Fournel, N., Pétrot, F.: Collecting traces in dynamic binary translation based virtual prototyping platforms. In: Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools, RAPIDO 2015, Association for Computing Machinery, New York (2015). https://doi.org/10.1145/2693433.2693437
12. d'Antras, A., Gorgovan, C., Garside, J., Luján, M.: Optimizing indirect branches in dynamic binary translators. ACM Trans. Archit. Code Optim. **13**(1) (2016). https://doi.org/10.1145/2866573
13. Dehnert, J.C., et al.: The Transmeta Code Morphing^{TM} software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, CGO 2003, pp. 15–24. IEEE Computer Society, USA (2003)
14. Eyolfson, J., Lam, P.: Detecting unread memory using dynamic binary translation. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 49–63. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35632-2_8
15. Federico, A.D., Agosta, G.: A jump-target identification method for multi-architecture static binary translation. In: 2016 International Conference on Compliers, Architectures, and Synthesis of Embedded Systems (CASES), pp. 1–10 (2016)
16. Horspool, R., Marovac, N.: An approach to the problem of detranslation of computer-programs. Comput. J. **23**(3), 223–229 (1980). WOS:A1980KD91500005. https://doi.org/10.1093/comjnl/23.3.223

17. Hu, W., et al.: Godson-3B: a 1GHz 40W 8-core 128GFLOPS processor in 65nm CMOS. In: 2011 IEEE International Solid-State Circuits Conference, pp. 76–78 (2011). https://doi.org/10.1109/ISSCC.2011.5746226
18. Hu, W., Yang, L., Fan, B., Wang, H., Chen, Y.: An 8-core MIPS-compatible processor in 32/28 nm bulk CMOS. IEEE J. Solid-State Circ. **49**(1), 41–49 (2014). https://doi.org/10.1109/JSSC.2013.2284649
19. Hu, W., et al.: Godson-3B1500: a 32nm 1.35GHz 40W 172.8GFLOPS 8-core processor. In: 2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers, pp. 54–55 (2013). https://doi.org/10.1109/ISSCC.2013.6487634
20. Luk, C.K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 190–200. Association for Computing Machinery, New York (2005). https://doi.org/10.1145/1065010.1065034
21. Molnar, I.: Performance counters for Linux (2009). https://lwn.net/Articles/337493/. Accessed 23 Feb 2022
22. Niu, G., Zhang, F., Li, X.: Eliminate the overhead of interrupt checking in full-system dynamic binary translator. In: Proceedings of the 15th ACM International Conference on Systems and Storage (2022)
23. Prasad, M.: A binary rewriting defense against stack-based buffer overflow attacks. In: 2003 USENIX Annual Technical Conference, USENIX ATC 03, San Antonio, TX, June 2003. USENIX Association (2003). https://www.usenix.org/conference/2003-usenix-annual-technical-conference/binary-rewriting-defense-against-stack-based
24. Shen, B.Y., Chen, J.Y., Hsu, W.C., Yang, W.: LLBT: an LLVM-based static binary translator. In: International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (2012)
25. Shen, B.Y., Hsu, W.C., Yang, W.: A retargetable static binary translator for the arm architecture. ACM Trans. Archit. Code Optim. **11**(2) (2014). https://doi.org/10.1145/2629335
26. Loongson Technology: Loongarch documentation (2022). https://loongson.github.io/LoongArch-Documentation/
27. Weiwu, H., et al.: Loongson instruction set architecture technology. J. Comput. Res. Dev., 1–22 (2022)
28. Wenzl, M., Merzdovnik, G., Ullrich, J., Weippl, E.: From hack to elaborate technique - a survey on binary rewriting. ACM Comput. Surv. **52**(3), 1–37 (2020)