CrossMapping: Harmonizing Memory Consistency in Cross-ISA Binary Translation

Chen Gao and Xiangwei Meng, *Lanzhou University;* Wei Li, *Tsinghua University;* Jinhui Lai, *Lanzhou University;* Yiran Zhang, *Beijing University of Posts and Telecommunications;* Fengyuan Ren, *Lanzhou University and Tsinghua University*

https://www.usenix.org/conference/atc24/presentation/gao-chen

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the 2024 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

CrossMapping: Harmonizing Memory Consistency in Cross-ISA Binary Translation

Chen Gao
Lanzhou University

Xiangwei Meng
Lanzhou University

Wei Li
Tsinghua University

Jinhui Lai
Lanzhou University

Yiran Zhang
Beijing University of Posts and Telecommunications

Fengyuan Ren*
Lanzhou University and Tsinghua University

Abstract

The increasing prevalence of new Instruction Set Architectures (ISAs) necessitates the migration of closed-source binary programs across ISAs. Dynamic Binary Translation (DBT) stands out as a crucial technology for the cross-ISA emulation of binary programs. However, due to the mismatch in memory consistency between guest ISA and host ISA, DBT systems face substantial challenges in guaranteeing correctness and translation performance for concurrent programs. Despite several attempts to bridge the memory inconsistency between guest and host ISA, prior work is either not universal for cross-ISA DBT systems or inefficient and even error-prone in translation.

This work presents CrossMapping, a general primitive mapping framework to enhance existing DBT systems for cross-ISA translation. By harmonizing memory consistency across diverse ISAs, CrossMapping enables smooth cross-ISA translation and accomplishes correct emulation. CrossMapping introduces specification tables to describe memory models in a unified and precise format, which facilitates the derivation of concurrent primitive mapping schemes based on a convenient comparison and analysis of memory models. The correctness of cross-ISA emulation is guaranteed by harmoniously integrating the derived mapping schemes with existing DBT systems. We evaluate CrossMapping for x86, ARMv8, and RISC-V on top of QEMU using the PARSEC benchmark suite. The results show that the average performance improvement can reach 8.5% when emulating x86 on ARMv8 and 7.3% when emulating x86 on RISC-V.

1 Introduction

Amidst the swift evolution of computer architecture, a multitude of processors embracing new ISAs such as ARM and RISC-V have emerged. Due to their cost-effectiveness, low power consumption, and license stability, many personal computer and server manufacturers are shifting from the traditional x86 architecture to these new architectures (e.g., Apple computers [8], Huawei Kunpeng servers [22], and Fujitsu's supercomputer Fugaku [17] run on ARM). Owing to the inherent incompatibility of programs across distinct ISAs, the adoption of Dynamic Binary Translation (DBT) technology, which fulfills the emulation of guest programs on the host through code translation, becomes imperative.

The correct and efficient emulation of concurrent programs poses a great challenge in DBT systems, which requires proper handling of memory consistency mismatch between guest ISA and host ISA. Modern ISAs, such as ARM and RISC-V, typically adhere to a Weak Memory Model (WMM) [4, 30, 50], whereas traditional ISAs, such as x86 and SPARC, often use a Total Store Order (TSO) model [23, 39, 40]. The strength of TSO model is stronger than WMM which allows more memory access pairs to be out-of-order. Emulating a TSO architecture on WMM architecture without auxiliary measures could lead to erroneous results. To properly handle memory consistency mismatch from strong to weak memory consistency, DBT systems need to insert memory fences in the host instruction sequence to keep guest orderings, which incurs performance degradation [33]. Emulating a WMM architecture on TSO architecture could not have any correctness issues, however many memory barriers for WMM become redundant for the TSO model.

Some efforts have been devoted to addressing memory inconsistency between guest ISA and host ISA. QEMU [11] enforces a stronger memory ordering than the guest's on the host through an overly conservative use of fences, leading to significant performance degradation. However, despite its conservative approach, QEMU fails to correctly handle Read-Modify-Write (RMW) instructions, potentially resulting in incorrect outcomes. Both Risotto [18] and Lasagne [46] proposed x86-to-ARMv8 mapping schemes through model formal analysis based on one-to-one memory, but do not support other ISAs. ArMOR [35] developed a mapping scheme based on Finite State Machines (FSM), which can handle multiple memory models. Nevertheless, it is not designed for cross-ISA DBT systems leveraging Intermediate Representation

*Corresponding author

(IR) and lacks support for one-way barriers and RMW in modern ISAs.

In this paper, we propose CrossMapping, a generic concurrent primitive mapping framework to harmonize memory consistency for cross-ISA DBT systems, which ensures correctness and maintains generality among diverse ISAs. CrossMapping introduces a memory model specification table to represent the memory models of the guest, host, and IR of DBT system in a unified and precise format. By comparing and analysing the memory models of guest, host, and IR, concurrent primitive mapping schemes for both guest-to-IR and IR-to-host can be obtained. In particular, the guest-to-IR mapping is implemented through an FSM, allowing for context-aware handling of concurrent primitives. Since the use of fences is related to the sequence of concurrent primitives, compared to direct mapping individual concurrent primitives, the FSM can provide more accurate mapping and reduce additional overhead. CrossMapping integrates its mapping schemes into DBT systems to guarantee the correct memory consistency during emulation execution.

We implement CrossMapping based on QEMU and evaluate its performance on the PARSEC benchmark suite [12]. CrossMapping enables correct binary translation among x86, ARMv8, and RISC-V ISAs and yields significant performance improvements. Compared with QEMU 8.0, CrossMapping improves the performance of runtime execution time by up to 15.8% (8.5% on average) for emulating x86 on ARMv8 and up to 12.7% (7.3% on average) for emulating x86 on RISC-V.

Our key contributions are summarized as follows:

- **Memory model specification table**: We introduce a specification table to describe memory models, greatly facilitating comparison and analysis of these models. Mapping schemes for any guest and host on any DBT system can be easily derived via the specification table, which is the foundation for ensuring the generality of cross-ISA binary translation.
- **Efficient mapping scheme**: We achieve efficient mapping by dividing it into two phases and employing targeted techniques. In the guest-to-IR mapping phase, an FSM is developed to capture the context of concurrent primitives to minimize the use of fences in IR. The IR-to-host mapping phase simply maps the IR fences to host fences without inserting new fences or eliminating old fences.
- **QEMU-based implementation**: As a case study, we implement CrossMapping on widely-used QEMU for x86, ARMv8, and RISC-V ISAs. Evaluations validate the efficient translation performance between x86 to ARMv8, x86 to RISC-V, ARMv8 to x86, and RISC-V to x86, which also demonstrate the generality of CrossMapping. The code is available at https://github.com/ChenGao1999/CrossMapping-for-QEMU.

2 Background

2.1 Dynamic Binary Translation

DBT technology can emulate guest binary programs on the host by translating codes at runtime [14, 44]. DBT systems generally perform translation at the granularity of at least one basic block. The typical workflow includes two steps: First, the guest machine code is translated into IR by the front end of the translator, and the IR code is optimized. Then, the IR is translated into host binary machine code by the back end of the translator. Basic blocks are usually cached to avoid redundant translation. QEMU [11] stands out as an exemplary open-source project and is widely adopted in academia and industry. In QEMU, the built-in Tiny Code Generator (TCG) front end transforms guest code to TCG IR, and the TCG backend generates host-executable machine code from TCG IR.

2.2 Memory Model

The memory model describes the behavior of concurrent primitives on shared memory. Due to different design choices, ISAs and IRs have different memory models, resulting in variations in concurrent primitives and their behaviors. The concurrent primitives can be categorized into four types: (1) Load and store access to shared memory, (2) Explicit memory barriers, also known as fences, (3) Implicit memory barriers associated with reads and writes, (4) Read-Modify-Write (RMW) access to shared memory.

Load and store accesses. Due to different memory models, the ordering of memory accesses varies. Store-load access pairs at different addresses are barely allowed to be out-of-order in TSO model, whereas all the four types of access pairs (load-load, load-store, store-load, and store-store) at different addresses are allowed to be out-of-order in WMM.

In WMM, maintaining logical correctness necessitates providing dependencies including address dependency, control dependency, and data dependency. Memory accesses become ordered when the value which has been read is utilized to calculate access address, the control condition, or the value written for subsequent memory accesses in program order.

In different ISAs, store accesses may become visible to different cores in the system at different times. Multiple-copy atomic stores must be visible to all cores in the system simultaneously. Strict multiple-copy atomicity comes with significant overhead and is uncommon, as it even prohibits forwarding from private local store buffers. Other-multiple-copy atomic stores must be visible to all cores simultaneously except for the issuing core. Non-multiple-copy atomic stores are visible to remote cores possibly in any order.

Explicit memory barriers. Explicit memory barriers, also known as fences, enforce the ordering of memory access before and after fence instructions. Most ISAs and IRs provide various memory fences for programmers. x86, ARMv8 and TCG IR offer full fences such as `MFENCE`, `DMB full`, and `Fsc`, so as to order any pair of memory accesses. Some WMM architectures also provide lightweight fences. For instance, ARMv8 offers `DMB ld` for ordering read accesses and their subsequent accesses, and `DMB st` for ordering a pair of store accesses. QEMU's TCG IR offers more flexible fences for load-load (`Frr`), load-store (`Frw`), store-load(`Fwr`), and store-store (`Fww`) access pairs to enforce the ordering. It is noteworthy that two fences can be combined. For example, `Frw` and `Fww` can be combined into a new barrier `Fmw` to order load-store and store-store access pairs.

Implicit memory barriers. Some ISAs provide load and store instructions with implicit barrier semantics. These instructions are typically one-way barriers and incur lower overhead compared to fences. For example, ARMv8 offers load-acquire, load-acquirePC, and store-release instructions. In this context, store-release is ordered with its predecessors, load-acquire and load-acquirePC are ordered with their successors, and store-release is ordered with its successor load-acquire.

RMW accesses. Read-Modify-Write (RMW) are atomic operations, like Compare-and-Swap, Test-and-Set, and Fetch-and-Add. They read a memory location and write a new value into it. ISAs provide various types of RMW primitives. For instance, x86 offers single-instruction RMW primitives, like `LOCK CMPXCHG` and `LOCK ADD`. ARMv8, in addition to single-instruction RMW primitives, like `CASAL`, provides RMW primitives through a pair of load-exclusive and store-exclusive instructions. Load-exclusive instructions contain acquiring semantics, and store-exclusive instructions contain releasing semantics, thereby allowing RMW primitives to act as barriers. Different RMW primitives present different memory behaviors. A successful single-instruction RMW primitive can act as a full barrier, and a pair of successful load-exclusive and store-exclusive can excite load-store access pairs to the same address, while a failed RMW primitive only produces a read access.

2.3 Memory Consistency Issues in DBT

Concurrency is often understood as the interleaved execution of multiple threads. If shared memory accesses within each thread follow program order, it is termed Sequential Consistency (SC) [26]. However, due to its poor performance, very few concurrent systems directly implement SC. Memory models weaker than SC can lead to non-SC behaviors. To ensure the correctness of program execution, it is necessary to enforce the ordering by inserting memory barriers.

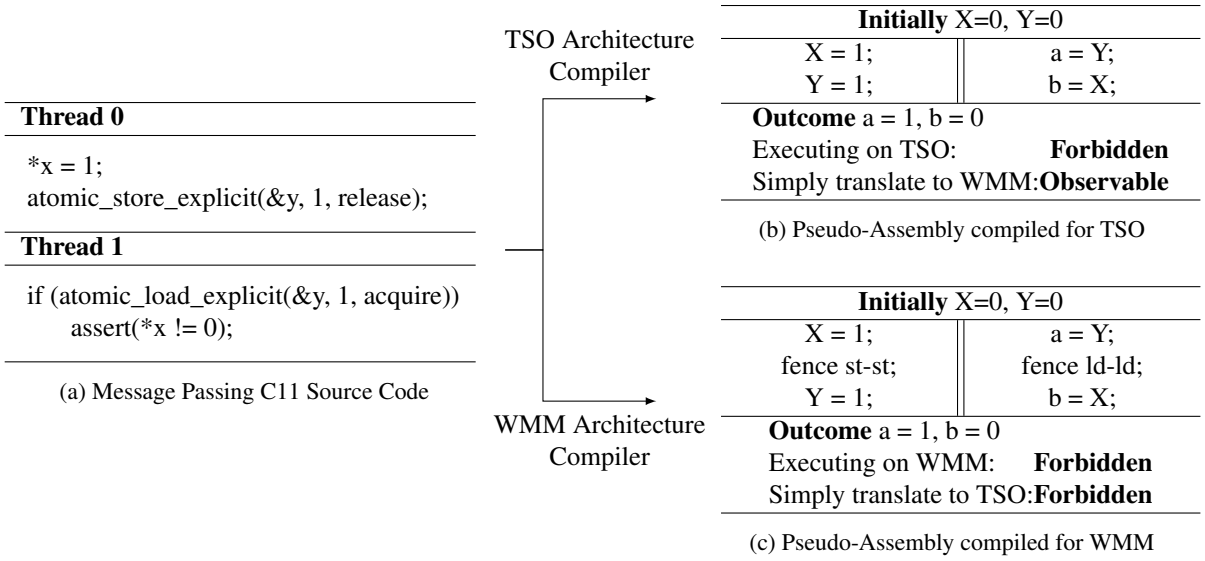
Existing work [9,10,25,42,43,47] has extensively explored memory consistency issues when compiling from high-level programming languages to various ISAs. Compilers ensure that the generated assembly code adheres to the ordering rules of the source code. However, for cross-ISA DBT systems, harmonizing memory consistency from guest to host presents significant challenges, which requires comparing and analysing the memory models of guest, host, and IR to derive a concurrent primitive mapping scheme from guest to host, and mapping concurrent primitives at runtime. Simple binary translation between ISAs may lead to incorrect results or performance penalty.

As shown in Figure 1, for the same segment of C11 code, different compilers targeting architectures with varying memory consistency models generate different assembly codes. For architectures with TSO model, like x86 and SPARC, load-load and store-store access pairs are ordered, so no additional fences need to be inserted. However, for architectures with WMM, like ARMv8 and RISC-V, memory access pairs are not ordered, so additional fences need to be inserted. Binary programs compiled directly from C11 code for different architectures would exhibit correct results (`a = 1, b = 0` cannot be observed). However, if a DBT system is used to translate a binary program from TSO architecture to WMM architecture without inserting any memory fence, erroneous results may occur in the WMM architecture (`a = 1, b = 0` could be observed), as shown in Figure 1(b). Conversely, although directly translating binary programs from WMM architecture to TSO architecture usually maintains correctness, it may remain unnecessary fences, as shown in Figure 1(c).

3 Related Work and Motivation

3.1 Cross-ISA DBT System

Cross-ISA DBT systems can be categorized into two types: system-level DBT systems emulating the entire machine and user-level DBT systems emulating applications. QEMU [11] supports both modes. In user mode, it can use multiple host threads to emulate concurrent programs. However, in system mode, it only supports executing the emulation of multiprocessors in a round-robin fashion in a single host thread if the host memory model cannot accommodate the guest [45]. CORMEU [49] and PQEMU [16] implement multi-threaded system-level emulation based on QEMU, but they do not address the mismatch in memory consistency models. Some DBT systems [13, 20, 21, 29] use LLVM IR [27] as the intermediate representation for performance optimization, but most of them cannot correctly support concurrent program emulation. Rosetta 2 [7] is a translator developed by Apple for program translation from x86 to ARM, it coordinates memory consistency by implementing x86 and ARM models in hardware [24]. It is regrettable that Rosetta is closed-source, lacking publicly available technical details.



(a) Message Passing C11 Source Code

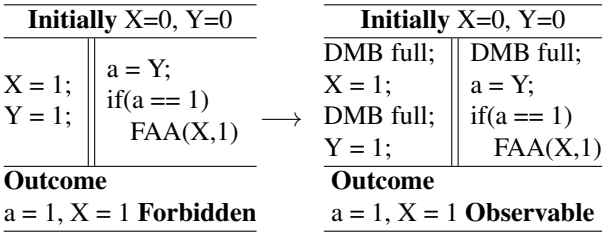
(b) Pseudo-Assembly compiled for TSO

(c) Pseudo-Assembly compiled for WMM

Figure 1: Message passing litmus test and pseudo-assembly for different architectures

Table 1: QEMU mapping schemes (x86 to ARMv8)

x86		TCG IR		ARMv8
Load	→	Fmr; ld	→	DMB ld; LDR
Store	→	Fmw; st	→	DMB full; STR
RMW	→	call	→	BLR; RMW; RET
MFENCE	→	Fsc	→	DMB full



(a) x86 Pseudo-Assembly

(b) Translated Code (ARMv8)

Figure 2: Fetch-And-Add litmus test

3.2 Harmonizing Memory Consistency

Some work [2,3,6,19,36,51] have aimed at comparing memory models and identifying their differences. Other investigations [1,5,28,31,32,37,38,48] address these differences through memory barrier placement strategies, but most are designed for compilers and are unsuitable for DBT systems.

To harmonize memory consistency in binary translation, some existing work suggest solutions through concurrent primitive mapping.

Native mapping schemes. QEMU adopts native concurrent primitive mapping schemes [45]. During the translation process from guest to TCG IR, a fence is inserted before each memory access to order any predecessors. In the translation process from TCG IR to host, TCG IR fence is translated into a sufficiently strong host fence to ensure both host and guest have the same memory ordering. Table 1 lists the mapping scheme from x86 to ARMv8. Taking the translation of store access as an example, the TCG frontend inserts an `Fmw` before store access, ordering store-store and load-store access pairs. Subsequently, the TCG backend translates `Fmw` into `DMB full` provided by ARMv8. However, `DMB full` orders any memory access pairs, resulting in performance penalty.

Although QEMU adopts very conservative mapping schemes, they cannot guarantee the correctness of RMW accesses on some WMMs. QEMU translates RMW instructions without additional barriers by calling helper functions, which may result in incorrect results. For instance, when translating x86 atomic instructions like `LOCK XCHG` and `LOCK ADD` to ARMv8, they are implemented using acquire-load-exclusive (`LDAXR`) and release-store-exclusive (`STLXR`) access pairs on ARMv8. Considering the litmus test illustrated in Figure 2, when translated from x86 to ARMv8 by QEMU, incorrect handling of RMW primitives leads to erroneous results. In x86, the Fetch-And-Add (FAA) operation is an atomic operation acting as a full barrier. It is ordered with the preceding load access. Therefore, when `if (a==1)` is true, the store instruction `X=1` in another core must have already been completed, and the FAA operation subsequently updates `X=2`. Consequently, the output `a=1, X=1` cannot be observed. In ARMv8, `FAA` is implemented using the `LDAXR-STLXR` access pair, where the load can be reordered with the succeeding `LDAXR`. Therefore, the load access to `X` in FAA might be completed before the `a=Y` instruction, leading to the erroneous result of observing `a=1, X=1`.

Table 2: Risotto and Lasagne mapping schemes (x86 to ARMv8)

x86		IR		ARMv8
Load	→	ld; Frm	→	LDR; DMB ld
Store	→	Fww; st	→	DMB st; STR
RMW	→	RMW	→	DMB full; RMW; DMB full
MFENCE	→	Fsc	→	DMB full

Special mapping schemes. Memory models are formally defined in various ISAs (e.g., x86 [40], ARMv8 [4], or RISC-V [50]). Due to the complexity of formally defining memory models and the usual incompatibility in these definitions, efficient and correct concurrent primitive mappings are constructed using formal analysis. This approach is specific to mapping one memory model to another, and the analysis process is quite intricate.

Risotto [18] and Lasagne [46] are two representative works on x86 to ARMv8 concurrent primitive mapping. Risotto is a dynamic binary translator based on QEMU, while Lasagne is a static binary translator based on LLVM. Both of them provide mapping schemes from x86 to ARMv8 through the IR, as shown in Table 2. Compared to QEMU's native concurrent primitive mapping approach, these mapping schemes reduce the use of excessive fences and fix the bugs in the mapping of RMW primitives. Additionally, they develop an enhanced scheme for fence merging in IR, namely merging two weaker fences into a stronger one to reduce fences further (e.g., `Frm` and `Fww` are merged into `Fsc`).

However, these schemes still introduce excessive fences. For instance, for load-store access pairs, when translating from x86 to TCG IR or LLVM IR, they insert `Frm` after a load access and `Fww` before a store access, merging the two fences into `Fsc`. When translating from TCG IR to ARMv8, `Fsc` is translated into ARMv8 full fence `DMB full`. The strength of `DMB full` is overkill, as the lightweight fence `DMB ld` is sufficient for ordering load-store access pairs.

Generic mapping schemes. ArMOR [35] is a framework designed for specifying and transforming memory models. The specification tables, which enable comparison of memory models, are introduced to define the ordering of memory accesses within a model, thus ArMOR can be adaptable to various memory models. The framework can generate Finite State Machines (FSMs) of memory model mapping based on both source and target memory models. Each state of the FSM indicates the memory ordering requirements after the execution of a concurrency primitive, determining whether concurrency primitives and their various successors need to be ordered. When encountering concurrency primitives during program execution, fences can be inserted based on the current state (the memory ordering requirements of preceding concurrency primitives) and the concurrent primitive itself.

Table 3: Summary of CrossMapping and Related Work

Tool	Cross-ISA DBT	Generality	Correct handling of RMW
QEMU	✔	✔	✘
Risotto	✔	✘	✔
Lasagne	✘	✘	✔
ArMOR	✘	✔	✘
Pico	✔	✘	✘
Cross-Mapping	✔	✔	✔

This approach accurately determines whether and what type of fence should be inserted between each memory access pair, effectively reducing the overhead caused by excessive fences.

However, the framework primarily targets heterogeneous architectures, and its concurrent primitive mapping relies on the Dynamic Binary Instrumentation (DBI) tool Pin [34]. Consequently, it cannot work in cross-ISA DBT systems where the translation process depends on the IR. Furthermore, ArMOR is unable to handle dependencies, one-way barriers, and RMW primitives, and considers RMW as a straightforward extension that behaves like load and store accesses. However, the behavior of RMW primitives varies across ISAs, and failure to address this issue may lead to errors similar to those illustrated in Figure 2. Pico [15] utilizes ArMOR to implement a DBT system from x86 to PowerPC, but does not ensure the correct translation of RMW primitives.

3.3 Motivation

Table 3 summarizes prior work, which primarily exhibit the following two shortcomings: (1) inserting excessive fences during mapping, resulting in additional overhead, and (2) special solution cannot provide a generic mapping scheme applicable to cross-ISA DBT systems.

Excessive fence insertion. The mapping schemes devised by QEMU, Risotto, and Lasagne need the insertion of fences for each memory access, thus they lack the capability to capture contextual information about memory accesses, namely they cannot recognize memory access pairs. This limits inserting the suitable strength of the memory fence. Instead, overkill or superfluous fences are inserted for ordering, resulting in excessive overhead. Both Risotto and Lasagne introduce fence merging optimization in IR and can combine two fences into one, thus reducing some superfluous fences. However, the strength of the merged fence is still overkill in some situations.

Inapplicable to generic cross-ISA DBT systems. While ArMOR is a general concurrent primitive mapping framework, it is designed for heterogeneous architectures and does

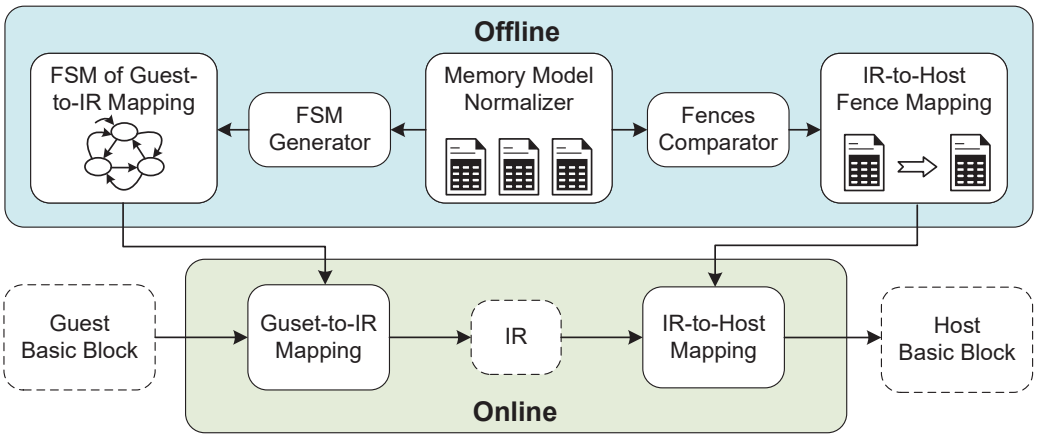


Figure 3: Overview of CrossMapping

not support DBT systems employing IR during translation. ArMOR, Pico, and QEMU fail to map certain RMW primitives provided by some WMM correctly, thus they can not properly work on modern WMM architectures like ARMv8. Risotto and Lasagne only support concurrent primitive mapping from x86 to ARMv8, and cannot be easily migrated to the memory models of other architectures.

Motivated by these issues, we develop CrossMapping to provide correct and efficient concurrent primitive mapping and support generic cross-ISA DBT systems.

4 Design of CrossMapping

4.1 Overview

CrossMapping is a generic framework for cross-ISA concurrent primitive mapping, and is adaptable to a range of guest and host ISAs, as well as DBTs. While ensuring correctness, CrossMapping achieves efficient concurrent primitive mapping. Figure 3 illustrates an overview of CrossMapping, including offline and online stages.

The runtime translation of guest binary programs to the host in DBT systems typically involves two steps: first, the translator's front end translates guest machine code into IR, and then, the translator's back end translates the IR into host machine code and executes it. The translation is usually done at the granularity of basic blocks to facilitate optimization and caching of these blocks. Therefore, the mapping process also occurs in two steps at the basic block level: guest-to-IR and IR-to-host. In the offline phase, CrossMapping separately derives these two mapping schemes, namely the FSM of mapping from guest to IR and the fence mapping from IR to host. In the online phase, conducting these two mappings within the DBT system can achieve the preservation of the guest memory orderings on the host during execution.

Offline stage. Due to the incompatibility of formalized memory models, direct comparison and exportation of mapping schemes are infeasible. To address this issue, we introduce a memory model normalizer to describe memory models using specification tables, which facilitate their comparison and analysis.

For the guest-to-IR mapping, we employ Finite State Machine (FSM) for efficient and precise memory fence insertion. The FSM can capture contextual information of concurrent primitives, and can conduce the identification of memory access pairs. The identification enables the strategic positioning of IR fences in the code and the selection of suitable types of IR fences for optimized synchronization. The FSM generator, by comparing the execution orderings of the guest with host memory models and considering the ordering specified by IR fences, outputs the FSM of guest-to-IR mapping.

For the IR-to-host mapping, the insertion or removal of fences is unnecessary. The key point focuses on mapping fences. Leveraging the fences comparator, the fence mapping from IR to host is derived by comparing the specification tables of IR fences and host fences.

Online stage. The online stage is integrated into the DBT system, enabling concurrent primitive mapping during program emulation. The FSM needs to be realized in the front end of the translator to achieve the mapping of concurrent primitives from guest to IR. The fence mapping needs to be conducted in the back end of the compiler to accomplish the mapping of concurrent primitives from IR to host.

4.2 Memory Model Normalizer

To handle the issue of the complexity and incompatibility in formalized memory models, Memory Model Normalizer is designed by leveraging a standard specification table to describe memory models. Specification tables can precisely describe the memory orderings on ISAs and the orderings

Table 4: Specification table of ARMv8 memory orderings

1st Ins. \ 2nd Ins. / Ord.	SA Ins.	DA ld	DA st	DA ld-aq	DA ld-PC	DA ld-rl
ld	✓	$✓_{dep}$	$✓_{dep}$	–	–	✓
st	✓	–	–	–	–	✓
ld-aq	✓	✓	✓	✓	✓	✓
ld-PC	✓	✓	✓	✓	✓	✓
st-rl	✓	–	–	–	–	✓

Table 5: Specification table of `DMB ld`

1st Ins. \ 2nd Ins. / Ord.	load	store
load	✓	✓
store	–	–

enforced by fences, and facilitate comparison and conversion between them.

Definition of specification table. Generally, the memory ordering of an ISA is described with one specification table, and the ordering enforced by each barrier is described with a separate specification table. An exception is implicit barriers, like load-acquire, store-release and load-acquirePC in ARMv8, which are attached to load and store accesses. They have dual semantics of both memory access and barrier, and are usually one-way barriers, so as to be arduously described by a single specification table. Therefore, we treat these implicit barriers as special memory accesses, and describe them in the memory ordering specification table.

As examples, Table 4 and Table 5 show the memory ordering specification tables for ARMv8 and its load fence `DMB ld`, respectively. The symbol ✓ indicates a pair of ordered memory accesses, $✓_{dep}$ indicates their ordering through dependencies, and – denotes out-of-order memory access. Each cell in the specification table specifies whether the memory access instruction of the type in the row header must be ordered with subsequent accesses of the type in the column header. In most of memory models, memory access pairs for identical and different addresses typically have different memory orderings. Accesses to the same address are generally ordered, known as coherence. However, for different addresses, different memory models typically have distinct memory orderings. Therefore, we explicitly distinguish between access to same address (`SA`) and access to different addresses (`DA`) in the specification table. For instance, in Table 4, load (`ld`) in the first row and subsequent load at different addresses (`DA ld`) in the second column rely on address dependency and control dependency, store (`st`) in the second row and subsequent load at different addresses in the second column are out-of-order, load-acquire (`ld-aq`) in the third row and subsequent load at different addresses in the second column are ordered.

Table 6: Refined specification table for `DMB ld`

1st Ins. \ 2nd Ins. / Ord.	SA Ins.	DA ld	DA st	DA ld-aq	DA ld-PC	DA ld-rl
ld	✓	✓	✓	✓	✓	✓
st	✓	–	–	–	–	✓
ld-aq	✓	✓	✓	✓	✓	✓
ld-PC	✓	✓	✓	✓	✓	✓
st-rl	✓	–	–	✓	–	✓

Refinement. As shown in Table 4 and Table 5, the memory ordering specification tables for ARMv8 and `DMB ld` have different rows and columns, which makes direct comparison impossible. Similar issues also arise in specification tables of different memory models. Therefore, it is necessary to refine the rows and columns of the specification tables into matching partitions.

The refinement is carried out in two steps. The first step is to find the set of memory accesses that serve as row and column headers, refining the specification tables to have the same rows and columns. Partition refinement techniques [41] can be used to merge the rows and columns from different specification tables into a finer-grained partition. The second step is to fill in the cells of refined specification table. This can be inferred by the contents of the original cells and the semantics of the refined set of memory accesses. Table 6 shows the refined specification table for `DMB ld`, which has the same rows and columns as the ARMv8 memory ordering specification table also illustrated in (Table 4). The refinement allows comparison between Table 4 and Table 6.

Comparison. The comparison of specification tables is conducted by analysing each pair of corresponding cells. If a specification table A can satisfy all memory orderings of another specification table B, then A is considered to be greater than or equal to B ($A \geq B$). Similarly, other comparison operations ($<, \leq, =, \neq, \geq, >$) can be defined on the specification tables. Union ($\cup$) and subtraction ($-$) operations can also be defined on the specification tables. Union produces a specification table that has a strength greater than or equal to each input table. Subtraction produces a specification table that specifies the orderings enforced by the first table but not the second one.

4.3 FSM of Guest-to-IR Mapping

The FSM guides the mapping of the guest concurrent primitive to IR. Since IR does not have its memory execution ordering, and its execution ordering depends on the host, the FSM of guest-to-IR mapping indicates how to insert fences into IR to ensure that the program ultimately satisfies the guest ordering when executed on the host. Each state in the FSM represents the memory orderings described by a specification table, and the orderings should be enforced at a certain moment on the host. Each transition condition is a concurrent primitive. The FSM outputs fences that should be inserted according to current state (current required memory orderings) when encountering concurrent primitives in the DBT system translation process, and transition to new state (new required memory orderings) after the concurrent primitives.

Typically, DBT systems translate at the granularity of basic blocks, and the translated basic blocks are cached for quick execution when the same block reappears. This means that memory model mapping also needs to be done at the granularity of basic blocks, and it is hard to predict the memory orderings satisfied by the first concurrent primitive within the basic block. Therefore, when entering a basic block, the FSM is initialized to the start state that the orderings are enforced in the guest but not in the host (i.e., Subtracting specification table of host memory ordering from specification table of guest memory ordering).

4.4 FSM Generator

The FSM generator calculates the start state of FSM, according to the specification tables of the guest and the host. Subsequently, the FSM generator calculates other states of the FSM, and determines what position and type of the IR fences should be inserted, as detailed in Algorithm 1. Depending on the type of concurrent primitives, four cases need to be handled: (1) For fences, directly map them to IR fences (Line 3-6); For single-instruction RMWs which can act as full barriers, it is necessary to check if the host has a corresponding single-instruction RMW. If the host has a corresponding instruction, then conduct the direct translation (Line 8-9). If not, the RMW is decomposed into load and store accesses, and fences are inserted sequentially to simulate a full barrier (Line 10-15); (3) In the cases where the previous operation is a single-instruction RMW and no other concurrent primitives are encountered at the end of the basic block, it is necessary to insert a barrier to ensure the RMW composed of load and store accesses can act as a full barrier (Line 16-18). (4) For other memory accesses, fences are inserted to comply with the required memory orderings (Line 19-22). Finally, the next state can be updated according to three parts: the current state, the inserted fences and the new orderings required by concurrent primitives (Line 23).

Algorithm 1: FSM Generation Algorithm

```
1  Function GetNextState(state, op):
2      requiredMo = guestMo−hostMo;
3      if IsFence(op) then
4          ordToEnforce = op;
5          fence = InsertIRFence(ordToEnforce);
6          newOrd = ∅;
7      else if IsSingleInstRMW(op) then
8          if GusetRMWActFullBarrier(op) then
9              newOrd = ∅;
10         else
11             ld, st = Split(op);
12             ldOrdToEnforce = KeepCol(state, ld);
13             fence = InsertIRFence(ldordToEnforce);
14             tmpState = (state−fence) ∪
                       KeepRow(requiredMo, ld);
15             newOrd = KeepRow(requiredMo, st);
16     else if IsEndOfBasicBlock(op) and
         isAfterRMW(state) then
17         fence = InsertIRFence(fullfence−state);
18         newOrd = ∅;
19     else
20         ordToEnforce = KeepCol(state, op);
21         InsertIRFence(ordToEnforce);
22         newOrd = KeepRow(requiredMo, op);
23     nextState = (state−fence) ∪ newOrd;
24     return nextState;
25 Function InsertIRFence(ordToEnforce):
26     return the weakest IR fence that satisfies
               fence ≥ ordToEnforce;
27 Function KeepRow(state, row):
28     for state[i][j] in state do
29         if i ≠ row then state[i][j] = −;
30     return state;
31 Function KeepCol(state, col):
32     for state[i][j] in state do
33         if i ≠ row then state[i][j] = −;
34     return state;
```

4.5 Fences Comparator

In the mapping process from IR to the host, direct mapping IR fences without any additional fence insertions to corresponding host fences is sufficient. The IR-to-host fence mapping scheme can be generated by the fence comparator by analysing the specification tables of both IR and host fences. For each IR fence, the comparator identifies the weakest host fence to satisfy all the enforced orderings of IR fence.

Table 7: Specification table for x86 memory orderings

(a) Original table

1st Ins. \ 2nd Ins. / Ord.	ld	st
ld	✓	✓
st	–	✓

(b) Refined table

1st Ins. \ 2nd Ins. / Ord.	ld	st	ld-aq	st-rl
ld	✓	✓	✓	✓
st	–	✓	–	✓
ld-aq	✓	✓	✓	✓
st-rl	–	✓	–	✓

4.6 Online Mapping

By applying the mapping scheme derived in the offline stage to the DBT system, we can achieve concurrent primitive mapping during binary translation, thereby ensuring the correct translation.

In the guest-to-IR mapping phase, fences need to be inserted into IR. Between two memory accesses, there may be some unrelated instructions, thus inserting a fence at any position in this instruction sequence has the same semantics. Liu et al. [33] pointed out that fences can slow down or even block subsequent unrelated instructions until a response is received from the bus. This means that inserting a fence closer to the end of the instruction sequence between two memory accesses has a weak influence on performance. Therefore, we choose to insert immediate fences before the succeeding access.

In the IR-to-host mapping phase, it is sufficient to simply map the fences without any additional operations.

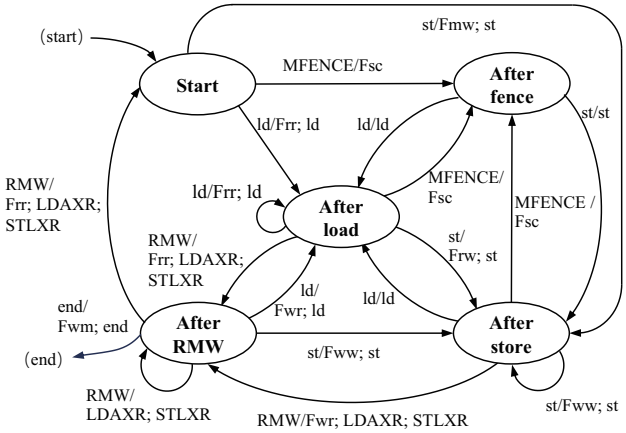
5 Case Study

In this section, based on QEMU's user-mode, we utilize CrossMapping to conduct some case studies, including strong-on-weak architectures (x86 to ARMv8, x86 to RISC-V) and weak-on-strong architectures (ARMv8 to x86, RISC-V to x86) mappings.

Specifically, due to the substantial overhead in QEMU of tracking memory access addresses and dependencies, we ignore these two features in subsequent experiments. That is, all memory accesses are considered to be at different addresses, and there is no dependency between access pairs. There is no influence on the correctness of experiment results since ignoring both of them excites the mapping scheme to insert abundant fences.

5.1 x86 to ARMv8 Mapping

Table 4 shows ARMv8's memory ordering specification table. The same-address memory access pairs and dependencies should be ignored, since the tracing of them leads to intolerable overhead in QEMU. The x86's memory ordering specification table is given in Table 7(a), and it is refined to have the same rows and columns as the ARMv8's specification table, as shown in Table 7(b). Both of them are other-multiple-copy atomic.



(a) FSM of mapping from x86 to TCG IR

TCG IR		ARMv8
Frr/Frw	→	DMB ld
Fww	→	DMB st
Fwr/Fmw/Fwm/Fsc	→	DMB full

(b) TCG IR to ARMv8 fence mapping scheme

Figure 4: x86 to ARMv8 mapping scheme

For fences, ARMv8 offers the load fence `DMB ld`, the store fence `DMB st`, and the full fence `DMB full`. x86 provides the full fence `MFENCE`. QEMU's TCG IR provides fences for any type of memory access pair.

For RMW primitives, they are all single-instruction and can act as a full barrier in x86. In ARMv8, only the compare-and-swap `CASAL` is single-instruction, while the rest of the RMW primitives are implemented through a `LDAXR-STLXR` pair and cannot act as a full barrier.

Based on the algorithm listed in Section 4.3, we can derive the complete mapping FSM, as shown in Figure 4(a). It has five states: **Start**, **After fence**, **After read**, **After write**, and **After RMW**. Each state can be described by a specification table, which specifies the memory orderings that should be enforced after a certain operation. When a concurrent primitive is encountered during the translation process, the FSM will output a sequence of IR instructions, and transition to a new state. For example, at the **Start** state, `ld/Frr;ld` means when a load access (`ld`) is input, the FSM will sequentially output `Frr` and a load access (`Frr;ld`) in the IR, and transition to the **After load** state.

- **Start state:** This state is entered at the beginning of translating a basic block. Table 8 shows the specification table

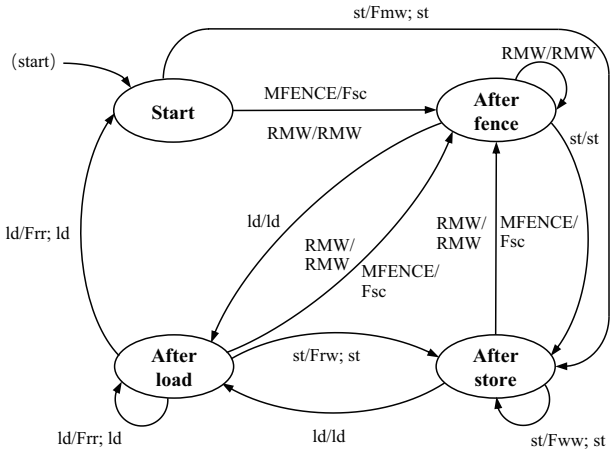
Table 8: **Start** state specification table

1st Ins. \ 2nd Ins. (Ord.)	ld	st	ld-aq	st-rl
ld	✓	✓	✓	–
st	–	✓	–	–
ld-aq	–	–	–	–
st-rl	–	✓	–	–

of this state, which is derived by subtracting ARMv8's memory ordering specification table from x86's, and presents all additional x86 memory orderings that should be enforced on ARMv8. In this state, when a load access is input, an `Frr` is inserted before the output access to order the load-load pair. When a store access is input, an `Fmw` is inserted to order load-store and store-store pairs. For RMW access, we treated it as a pair of acquire-load-exclusive (`LDAXR`) and release-store-exclusive (`STLXR`) accesses, the appropriate fences are inserted before and after the access pair in the output to emulate the behavior of x86 single-instruction RMW access. In this state, an `Fmr` is inserted before the `LDAXR` to enforce its ordering with any types of preceding accesses.

- **After load state:** Entered after load access. In this state, to enforce the ordering of load-load and load-store access pairs, `Frr` and `Frw` should be inserted before load and store accesses in the output, respectively. For RMW access, `Frr` is inserted before the `LDAXR` to enforce its ordering with preceding load accesses.
- **After store state:** Entered after store access. In this state, to enforce the ordering of store-store access pair, `Fww` is inserted before output store accesses. For RMW access, `Fwr` is inserted before the `LDAXR` to enforce its ordering with preceding store accesses.
- **After fence state:** Entered after `MFENCE` operation, and no memory ordering needs to be enforced. Thus, for any input operations, no additional fences are required in the output.
- **After RMW state:** Entered after RMW access. In this state, the ordering of `STLXR` and subsequent accesses must be enforced. `Fwr` and `Fww` are inserted respectively before output load and store accesses to enforce the ordering. When no new concurrent primitives are encountered by the end of the basic block, an `Fwm` is inserted before the end to prevent disorder between `STLXR` and subsequent basic block memory accesses.

By comparing the strength of fences in TCG IR and ARMv8, the TCG IR to ARMv8 mapping scheme can be derived, as shown in Figure 4(b).



(a) FSM of mapping from x86 to TCG IR

TCG IR		RISC-V
Frr	→	fence [r,r]
Frw	→	fence [r,w]
Fww	→	fence [w,w]
Fmw	→	fence [rw,w]
Fsc	→	fence [rw,rw]

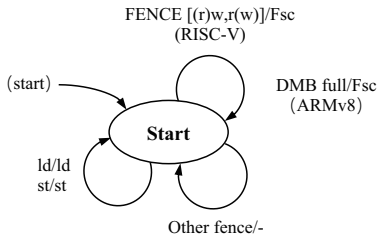
(b) TCG IR to RISC-V fence mapping scheme

Figure 5: x86 to RISC-V mapping scheme

5.2 x86 to RISC-V Mapping

Similar to ARMv8, RISC-V adopts WMM model, allowing memory accesses to different addresses to be out-of-order, and is other-multiple-copy atomic. RISC-V offers standard atomic instruction extensions to support RMW primitives [50]. It provides two forms of atomic instructions: load-reserved (`LR`) / store-conditional (`SC`) pairs and atomic memory operations (`AMO`). Both of them support various memory consistency orderings, including unordered, acquire (`aq`), release (`rl`), and sequentially consistent (`aqrl`) semantics. For `AMO` instructions, they can be set as `aqrl` to act as a full barrier. For `LR/SC` instruction pairs, the `LR` can be set as `rl` and the `SC` as `aq` to act as a full barrier. RISC-V also provides the fence instruction to order any combination of memory loads (`r`) and stores (`w`) relative to any types of their combination.

Figure 5 shows the mapping scheme from x86 to RISC-V. In the FSM of mapping from x86 to TCG IR, since RMW primitives in RISC-V can act as a full barrier, they can be directly translated from x86 to RISC-V. The FSM consists of four states: **Start**, **After read**, **After write**, and **After fence**. The transition conditions and outputs among these four states are similar to those in the mapping scheme for ARMv8.



(a) FSM of mapping from ARMv8/RISC-V to TCG IR

TCG IR		x86
Fsc	→	MFENCE

(b) TCG IR to x86 fence mapping scheme

Figure 6: ARMv8/RISC-V to x86 mapping scheme

5.3 ARMv8/RISC-V to x86 Mapping

The memory model of x86 is stronger than those of ARMv8 and RISC-V, meaning that x86 can naturally satisfy the ordering of ARMv8 and RISC-V without any additional fences. Therefore, when mapping ARMv8/RISC-V to x86, it only needs to eliminate extra fences and translate the necessary ones. Figure 6 shows the mapping scheme from ARMv8/RISC-V to x86. For ARMv8, `DMB full` should be translated to `Fsc`, and for RISC-V, the fence instruction should be translated to `Fsc` when its predecessor set includes `w` and the successor set includes `r`. As for other fences, they are redundant for x86 and can be eliminated.

6 Evaluation

To validate the basic function of CrossMapping and evaluate its translation performance, we implement the four mapping schemes in Section 5 based on QEMU v8.0.0, and conduct a performance evaluation. As mentioned in Section 3, Lasagne [46] is a static binary translator and cannot work in DBT systems. ArMOR [35] is not suitable for cross-ISA DBT systems and fails to handle RMW primitives correctly. Therefore, we compare CrossMapping with QEMU's native mapping scheme and Risotto [18] and analyze its performance gains.

6.1 Experiment Setup

Testbed. We conduct experiments on x86, ARMv8, and RISC-V platforms.

- **x86:** A server equipped with two Intel Xeon Silver 4210R processors (10 cores per chip, 2.4 GHz) and 64GB memory, and runs Ubuntu 20.04 with Linux kernel 5.15.
- **ARMv8:** A server equipped with two Huawei Kunpeng 920-4826 processors (ARMv8.2, 48 cores per chip, 2.6 GHz) and 192GB memory, and runs Ubuntu 18.04 with Linux kernel 5.0.
- **RISC-V:** RISC-V Linux development board LicheePi 4A with TH1520 SOC (RV64GCV, 4 cores, 1.85GHz) and 16GB memory, and runs Debian 12 with Linux kernel 5.10.

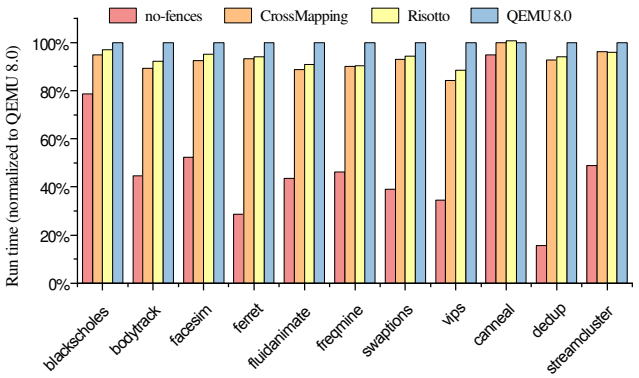
Benchmark. We use the PARSEC 3.0 benchmark suite [12]. The `raytrace` and `x264` benchmarks are omitted because they could not be natively built and run on ARMv8 and RISC-V. For emulated execution on x86 and ARMv8, the `native` input set is adopted. For RISC-V, due to performance limitations, the `simlarge` input set is adopted. Since the `facesim` benchmark only supports 2^n threads, we use hardware threads as many as possible, which implies 16 threads for x86, 64 threads for ARMv8, and 4 threads for RISC-V.

6.2 Strong-on-Weak Architecture Emulation

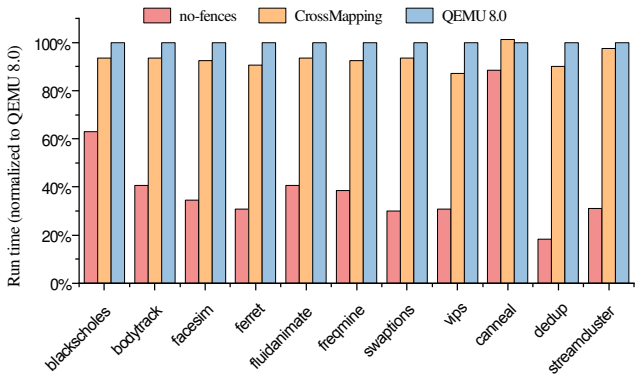
Figure 7 shows the run time for strong-on-weak architecture emulation.

Cost of enforcing memory ordering. For strong-on-weak architecture emulation, additional fences should be inserted to enforce memory orderings on WMM, which introduces extra overhead. We first assessed the overhead caused by QEMU's fence insertions. Testing the performance without fences (note that this is incorrect), we found that the additional fences occupied a significant portion of the execution time in benchmarks. For x86 to ARMv8 translation, the extra fences accounted for up to 84.4% (for `dedup`), averaging 52.1% of run time. For x86 to RISC-V, they reach up to 81.8% (for `dedup`), averaging 59.4%. These results indicate the necessity of reducing fence insertion while ensuring program correctness in strong-on-weak architecture emulation.

Comparison with QEMU. Taking the x86 to ARMv8 mapping as an example, QEMU's mapping scheme (Table 1) inserts `DMB full` for load-store and store-store pairs and `DMB ld` for store-load pairs to keep order. In contrast, CrossMapping's scheme (Figure 4) only needs to insert `DMB ld` for load-store pairs, `DMB st` for store-store, and no fences for store-load, significantly reducing the strength and number of extra fences. Additionally, CrossMapping corrects QEMU's erroneous translation of RMW primitives. Compared to QEMU, CrossMapping significantly improves execution performance without any impact on program correctness. For the x86 to ARMv8 translation, improvements reach up to 15.8% (for `vips`), with a geometric mean of 8.5%. For the x86 to RISC-V, improvement reaches up to 12.7% (for `vips`), with a geometric mean of 7.3%.



(a) Emulating x86 on ARMv8



(b) Emulating x86 on RISC-V

Figure 7: Run time of the PARSEC benchmark during strong-on-weak architecture emulation

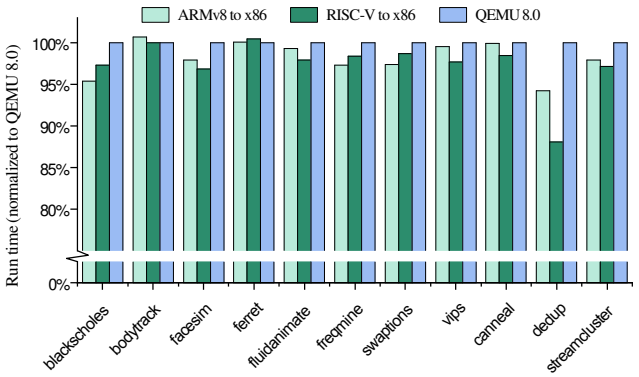


Figure 8: Run time of the PARSEC benchmark during weak-on-strong architecture emulation

Comparison with Risotto. Risotto only supports mapping from x86 to ARMv8 (Table 2). Compared with CrossMapping, it also inserts excessive fences. For load-store pairs, Risotto generates a `DMB full` after inserting and merging fences, while CrossMapping only needs a `DMB ld`. For RMW primitives combined on ARMv8, Risotto needs to insert a `DMB full` before and after RMW primitives, whereas CrossMapping can insert lighter fences depending on the context.

We also implement Risotto's mapping scheme based on QEMU v8.0.0. Taking comparison with it, CrossMapping achieves up to 5.0% improvement (for `vips`), with a geometric mean of 1.8% as shown in Figure 7(a).

6.3 Weak-on-Strong Architecture Emulation

Figure 8 shows the run time for weak-on-strong architecture emulation. In this case, only the redundant fences need to be removed. Taking the ARMv8 to x86 mapping as an example, Table 9 shows QEMU's mapping scheme. Compared with CrossMapping's scheme (Figure 6), the fences for x86 are identical. However, QEMU removes redundant fences in the TCG IR to x86 mapping stage, while CrossMapping removes them in the ARMv8 to TCG IR mapping stage. Removing fences earlier reduces the overhead during translation, thus improving the geometric mean of the x86 to ARMv8 translation by 1.9%, and the x86 to RISC-V translation by 2.8%.

Table 9: QEMU mapping schemes (ARMv8 to x86)

ARMv8		TCG IR		x86
LDR/STR	→	ld/st	→	Load/Store
RMW	→	call	→	BLR; RMW; RET
DMB ld	→	Fmr	→	–
DMB st	→	Fww	→	–
DMB full	→	Fsc	→	MFENCE

7 Conclusion

In this paper, we develop CrossMapping, a generic and efficient concurrent primitive mapping framework for cross-ISA DBT systems to reconcile memory consistency mismatch in binary translation. We present four cases of emulation between strong-on-weak and weak-on-strong architectures using CrossMapping and evaluate the performance under a typical multi-threaded benchmark suite. CrossMapping simultaneously achieves correctness, significant performance improvement, and the generality of cross-ISA binary translation, indicating that it is a promising substrate to strengthen existing DBT systems.

Acknowledgments

The authors gratefully acknowledge the anonymous reviewers for their constructive comments. This work is supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62132007, No. 62221003 and No. 62302055 as well as gifts from Huawei.

References

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Magnus Lång, and Tuan Phong Ngo. Precise and sound automatic fence insertion procedure under pso. In Ahmed Bouajjani and Hugues Fauconnier, editors, *Networked Systems*, pages 32–47, Cham, 2015. Springer International Publishing. https://doi.org/10.1007/978-3-319-26850-7_3.

[2] S.V. Adve and M.D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, 1993. https://doi.org/10.1109/71.242161.

[3] Jade Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design*, 41:178–210, 2012. https://doi.org/10.1007/s10703-012-0161-5.

[4] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. Armed cats: Formal concurrency modelling at arm. *ACM Trans. Program. Lang. Syst.*, 43(2), jul 2021. https://doi.org/10.1145/3458926.

[5] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 508–524, Cham, 2014. Springer International Publishing. https://doi.org/10.1007/978-3-319-08867-9_33.

[6] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 258–272, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-14295-6_25.

[7] Apple. About the rosetta translation environment. https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment/.

[8] Apple. Apple unveils m3, m3 pro, and m3 max, the most advanced chips for a personal computer. https://nr.apple.com/Di5I4t7da8, 2023.

[9] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and compiling c/c++ concurrency: From c++11 to power. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 509–520, New York, NY, USA, 2012. Association for Computing Machinery. https://doi.org/10.1145/2103656.2103717.

[10] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, page 55–66, New York, NY, USA, 2011. Association for Computing Machinery. https://doi.org/10.1145/1926385.1926394.

[11] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association. https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator.

[12] Christian Bienia. *Benchmarking modern multiprocessors*. PhD thesis, Princeton University, 2011. https://www.cs.princeton.edu/techreports/2010/890.pdf.

[13] Vitaly Chipounov and George Candea. Dynamically translating x86 to llvm using qemu. Technical report, 2010. https://infoscience.epfl.ch/record/149975/files/x86-llvm-translator-chipounov_2.pdf.

[14] Cifuentes and Malhotra. Binary translation: static, dynamic, retargetable? In *1996 Proceedings of International Conference on Software Maintenance*, pages 340–349, 1996. https://doi.org/10.1109/ICSM.1996.565037.

[15] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. Cross-isa machine emulation for multicores. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 210–220, 2017. https://doi.org/10.1109/CGO.2017.7863741.

[16] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. Pqemu: A parallel system emulator based on qemu. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 276–283, 2011. https://doi.org/10.1109/ICPADS.2011.102.

[17] Fujitsu. Supercomputer fugaku specifications. https://www.fujitsu.com/global/about/innovation/fugaku/specifications/.

[18] Redha Gouicem, Dennis Sprokholt, Jasper Ruehl, Rodrigo C. O. Rocha, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. Risotto: A dynamic binary translator for weak memory model architectures. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages*

and Operating Systems, Volume 1, ASPLOS 2023, page 107–122, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3567955.3567962.

[19] Lisa Higham, Jalal Kawash, and Nathaly Verwaal. Defining and comparing memory consistency models. In *PDCS'97*, 1997. http://hdl.handle.net/1880/45991.

[20] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. Hqemu: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, page 104–113, New York, NY, USA, 2012. Association for Computing Machinery. https://doi.org/10.1145/2259016.2259030.

[21] Chun-Chen Hsu, Pangfeng Liu, Chien-Min Wang, Jan-Jan Wu, Ding-Yong Hong, Pen-Chung Yew, and Wei-Chung Hsu. Lnq: Building high performance dynamic binary translators with existing compiler backends. In *2011 International Conference on Parallel Processing*, pages 226–234, 2011. https://doi.org/10.1109/ICPP.2011.57.

[22] Huawei. Kunpeng 920. https://www.hisilicon.com/en/products/Kunpeng/Huawei-Kunpeng/Huawei-Kunpeng-920.

[23] SPARC International Inc and David L Weaver. *The SPARC architecture manual*. Prentice-Hall Englewood Cliffs, NJ, USA, 1994. https://0x04.net/~mwk/doc/sparc/SPARCV9.pdf.

[24] Saagar Jha. Tsoenable - kernel extension that enables tso for apple silicon processes. https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment/.

[25] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 175–189, New York, NY, USA, 2017. Association for Computing Machinery. https://doi.org/10.1145/3009837.3009850.

[26] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979. https://doi.org/10.1109/TC.1979.1675439.

[27] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004. https://doi.org/10.1109/CGO.2004.1281665.

[28] Jaejin Lee and D.A. Padua. Hiding relaxed memory consistency with compilers. In *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00622)*, pages 111–122, 2000. https://doi.org/10.1109/PACT.2000.888336.

[29] Wei Li, Xiaohui Luo, Yiran Zhang, Qingkai Meng, and Fengyuan Ren. Crossdbt: An llvm-based user-level dynamic binary translation emulator. In José Cano and Phil Trinder, editors, *Euro-Par 2022: Parallel Processing*, pages 3–18, Cham, 2022. Springer International Publishing. https://doi.org/10.1007/978-3-031-12597-3_1.

[30] Arm Limited. Arm architecture reference manual. https://developer.arm.com/documentation/ddi0487/ja/?lang=en.

[31] Alexander Linden and Pierre Wolper. A verification-based approach to memory fence insertion in pso memory systems. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 339–353, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-36742-7_24.

[32] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 429–440, New York, NY, USA, 2012. Association for Computing Machinery. https://doi.org/10.1145/2254064.2254115.

[33] Nian Liu, Binyu Zang, and Haibo Chen. No barrier in the road: A comprehensive study and optimization of arm barriers. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, page 348–361, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3332466.3374535.

[34] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 190–200, New York,

NY, USA, 2005. Association for Computing Machinery. https://doi.org/10.1145/1065010.1065034.

[35] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. Armor: Defending against memory consistency model mismatches in heterogeneous architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 388–400, New York, NY, USA, 2015. Association for Computing Machinery. https://doi.org/10.1145/2749469.2750378.

[36] Sela Mador-Haim, Rajeev Alur, and Milo M. K. Martin. Generating litmus tests for contrasting memory consistency models. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 273–287, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-14295-6_26.

[37] Robin Morisset and Francesco Zappa Nardelli. Partially redundant fence elimination for x86, arm, and power processors. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, page 1–10, New York, NY, USA, 2017. Association for Computing Machinery. https://doi.org/10.1145/3033019.3033021.

[38] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. Vsync: Push-button verification and optimization for synchronization primitives on weak memory models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 530–545, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3445814.3446748.

[39] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In Theo D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, pages 478–503, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-14107-2_23.

[40] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 391–407, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-03359-9_27.

[41] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987. https://doi.org/10.1137/0216062.

[42] Gustavo Petri, Jan Vitek, and Suresh Jagannathan. Cooking the books: Formalizing jmm implementation recipes. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 445–469, Dagstuhl, Germany, 2015. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2015.445.

[43] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. 3(POPL), jan 2019. https://doi.org/10.1145/3290382.

[44] Mark Probst. Dynamic binary translation. In *UKUUG Linux Developer's Conference*, volume 2002, 2002. https://www.complang.tuwien.ac.at/schani/papers/bintrans.pdf.

[45] QEMU. Multi-threaded tcg. https://www.qemu.org/docs/master/devel/multi-thread-tcg.html.

[46] Rodrigo C. O. Rocha, Dennis Sprokholt, Martin Fink, Redha Gouicem, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. Lasagne: A static binary translator for weak memory model architectures. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 888–902, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3519939.3523719.

[47] Jaroslav Ševčík and David Aspinall. On validity of program transformations in the java memory model. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, pages 27–51, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-70592-5_3.

[48] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. 10(2):282–312, apr 1988. https://doi.org/10.1145/42190.42277.

[49] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. Coremu: A scalable and portable parallel full-system emulator. PPoPP '11, page 213–222, New York, NY, USA, 2011. Association for Computing Machinery. https://doi.org/10.1145/1941553.1941583.

[50] Andrew Waterman and Krste Asanovic. The risc-v instruction set manual volume i: Unprivileged

isa. Technical report, SiFive Inc., 2019. https://drive.google.com/file/d/1s0lZxUZaa7eV_O0_WsZzaurFLLww7ou5/view?pli=1.

[51] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 190–204, New York, NY, USA, 2017. Association for Computing Machinery. https://doi.org/10.1145/3009837.3009838.