Risotto: A Dynamic Binary Translator for Weak Memory Model Architectures

Redha Gouicem*
TU Munich
Germany
gouicem@in.tum.de

Dennis Sprokholt*
TU Delft
Netherlands
d.g.sprokholt@tudelft.nl

Jasper Ruehl
TU Munich
Germany
jasper.ruehl@tum.de

Rodrigo C. O. Rocha
University of Edinburgh
UK
rrocha@ed.ac.uk

Tom Spink
University of St Andrews
UK
tcs6@st-andrews.ac.uk

Soham Chakraborty
TU Delft
Netherlands
s.s.chakraborty@tudelft.nl

Pramod Bhatotia
TU Munich
Germany
pramod.bhatotia@tum.de

ABSTRACT

Dynamic Binary Translation (DBT) is a powerful approach to support cross-architecture emulation of unmodified binaries. However, DBT systems face correctness and performance challenges, when emulating *concurrent binaries from strong to weak memory consistency architectures*. As a matter of fact, we report several translation errors in Qemu, when emulating x86 binaries on Arm hosts.

To address these challenges, we propose an end-to-end approach that provides correct and efficient emulation for weak memory model architectures. Our contributions are twofold: First, we formalize Qemu's intermediate representation's memory model, and use it to propose formally verified mapping schemes to bridge the *strong-on-weak memory consistency mismatch*. Second, we implement these verified mappings in Risotto, a Qemu-based DBT system that optimizes memory fence placement while ensuring correctness. Risotto further improves performance via cross-architecture dynamic linking of native shared libraries and faster yet correct translation of compare-and-swap operations.

We evaluate Risotto using multi-threaded benchmark suites and real-world applications, and show that Risotto improves the emulation performance by 6.7% on average over "erroneous" Qemu, while ensuring correctness.

ACM Reference Format:
Redha Gouicem, Dennis Sprokholt, Jasper Ruehl, Rodrigo C. O. Rocha, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. 2022. Risotto: A Dynamic Binary Translator for Weak Memory Model Architectures. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*The first two authors contributed equally to this work.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With the emergence of new Instruction Set Architectures (ISAs) like Arm or RISC-V, the landscape of computing hardware is steadily shifting in the recent years [13, 36]. Major industry players are moving away from the currently dominating x86 to favor new features, performance, power efficiency, and license support [16, 32, 75]. However, this transition is not straightforward since existing applications are not compatible across different ISAs. To address this problem, DBT technology emulates the program's *guest* ISA on the *host* machine, by translating the code at run time [69, 78].

A major challenge for DBT systems is correct and performant emulation of concurrent binaries [28, 51]. The root cause of this issue is the mismatch in the memory model semantics between the guest and the host architectures, which is particularly problematic when translating from a strong memory model, *e.g.,* x86, to a weaker model, *e.g.,* Arm [6]. At a high-level, the DBTs must ensure that the behavior of the guest ISA is correctly reproduced on the host machine so that the application's original semantics are preserved.

In order to correctly support *strong-on-weak memory consistency* [88], DBTs must insert *memory fences* to preserve guest orderings, sacrificing performance [49]. For example, Qemu [69], a state-of-the-art DBT, tries to enforce a stronger ordering than x86's when emulating it on Arm, unnecessarily hurting performance. Despite its attempt at enforcing strong ordering, it fails to ensure correctness—we discover and report several translation errors in Qemu due to incorrect fence usage that may lead to errors at run time. Further, while reasoning about mapping correctness, we discover and report that the Arm memory model [6] does not facilitate optimal mapping as it requires additional fences in x86 to Arm translation.

Moreover, the runtime performance is paramount for the adoption of DBT systems. Many user-mode DBT systems translate the entire application up to the system call interface, and fail to take advantage of pre-compiled host instructions where available. For instance, commonly used shared libraries are often present in the

host system, however Qemu, instead of using the native and highly optimized version of the library, translates a guest version of the shared library function to the host ISA.

In this paper, we propose an end-to-end DBT approach based on Qemu that provides correct and efficient execution of concurrent x86 binaries on Arm architectures by combining: (a) formal verification of translation correctness for strong-on-weak architecture, and (b) a DBT system for run time binary translation based on these verified translation rules.

More specifically, on the formal verification aspect, we propose the *first formal concurrency memory model* of Qemu's intermediate representation (TCG IR). We use our formalization to offer verified mapping schemes, proving the correctness of *(1)* x86 to TCG IR and *(2)* TCG IR to Arm mapping schemes. We develop these correctness proofs using the Agda theorem prover [4].

Another aspect of Qemu's Tiny Code Generator (TCG) is the intermediate optimizations on the concurrency primitives, which may affect the translation correctness, as all transformation for sequential programs may be not be correct for concurrent programs [25, 57, 84]. Hence, only ensuring the memory model mismatches in architectures [28, 51] does not guarantee correct translation in Qemu. Therefore, we prove the correctness of a number of optimizations, including the ones performed by TCG. These verified optimizations, along with verified mappings, facilitate the development of an end-to-end DBT system based on Qemu.

On the system side, we build Risotto, a Qemu-based DBT system that implements these verified translation rules for mappings and optimizes fence placement. Risotto further enhances the emulation performance via a cross-architecture dynamic linker that uses native shared libraries whenever available, instead of translating their guest counterpart. In addition, Risotto leverages recent Arm atomic instructions to efficiently and correctly translate Compare-and-Swap (CAS) operations.

We evaluate Risotto on the PARSEC [19] and Phoenix [70] benchmark suites, as well as various real-world applications such as OpenSSL and SQLite. Our evaluation shows that Risotto improves performance whilst still being correct with regards to memory ordering by up to 19.7%, and 6.7% on average compared to "erroneous" Qemu. We also show that our dynamic linker allows applications using shared libraries to match the speed of native applications using these libraries.

Overall, our paper makes the following contributions:

- **Concurrency analysis in QEMU and Arm memory model.** We discover and report several translation errors in Qemu due to the incorrect usages of memory fences. We also report *undesired behavior* in the Arm memory model (herein referred to as Arm-Cats) for efficient x86-to-Arm translation [6], and propose revisions to the model for verified mappings. These revisions were accepted in the Arm-Cats model [37].
- **TCG IR memory model: Formalization, verified mappings and optimizations.** We formalize Qemu's TCG IR. Based on this formal model, we propose mapping schemes from x86 to TCG IR and TCG IR to Arm, which we verify to be semantically correct. We also prove the correctness of various optimizations on TCG IR model which are performed by QEMU. These mapping schemes have been submitted to the Qemu mailing list and are under review at the time of writing.
- **Risotto DBT system.** We build Risotto, an end-to-end DBT system that is based on the formally verified translations on QEMU. In addition, we implement a dynamic host library linker, which allows to match the speed of native execution when using native shared libraries instead of translated libraries. Lastly, we implement a fast and correct translation of CAS operations.

2 BACKGROUND

2.1 Weak Memory Model Architectures

Concurrency is often interpreted as an interleaving of operations performed by multiple threads, with the operations in each thread executing in program order. This is known as Sequential Consistency (SC) [42]. However, concurrent systems may also behave in ways that cannot be explained by interleaving semantics. These non-SC behaviors result in weak memory models.

Weak memory models arise in some architectures due to various microarchitectural design decisions, *e.g.,* the memory hierarchy, or out-of-order execution. Therefore, memory models may vary among different ISAs, *e.g.,* x86 and Arm. The example below shows how the allowed behaviors for a program may vary depending on the memory model.

$$\begin{array}{l} X = Y = 0; \\ X = 1; \\ Y = 1; \end{array} \left\| \begin{array}{l} a = Y; \\ b = X; \end{array} \right. \quad \text{(MP)}$$

Shared variables X and Y are initialized to zero. The program has two concurrent threads. Weak outcome $a = 1, b = 0$ is allowed in Arm but disallowed in x86.

Implication on binary translation. If we translate the MP program's binary from x86 to Arm, without taking their memory models into account, the resulting Arm binary may exhibit undesirable behaviors. This incorrectness is due to the different memory consistency models between the source and destination ISAs. It can be fixed by explicitly enforcing the memory model of the source ISA when translating into the destination ISA via memory fences. However, the introduction of additional fences has a significant impact on performance [49].

2.2 Dynamic Binary Translation

DBT systems typically operate as follows: *(1)* translate the instruction currently pointed at by the emulated Instruction Pointer (IP), and *(2)* execute the translated instruction, updating the IP to either the following instruction or the target of a jump. Most DBTs implement translation granularities of at least a *basic block*, and employ classic compiler optimizations to improve generated code quality (and hence run time performance). Basic blocks are often cached to avoid repeating translations.

Qemu is a state-of-the-art emulator capable of cross-ISA emulation that supports two operation modes: full system or user mode. Full system mode emulates the entire machine while user mode only emulates applications. In the latter, system calls are natively executed by the host machine and not emulated. In this paper, we focus on user-mode Qemu.

Shared libraries support. Qemu treats the application binary and any shared libraries as a unit, translating both guest application code and guest library code on-the-fly. This requires a guest version of the shared library to be available for the application to function

Access type	x86	TCG IR	Arm
Load	RMOV	ld	LDR
Store	WMOV	st	STR
Full-fence	MFENCE	Fsc	DMBFF
WW-fence		Fww	DMBST
RM-fence		Frm	DMBLD
MW fence		Fmw	
Atomic-update	RMW	RMW	$\texttt{RMW}^1$, $\texttt{RMW}^2$
Rel.Acq. atomic-update			$\texttt{RMW}^1_{\texttt{AL}}$, $\texttt{RMW}^2_{\texttt{AL}}$

$\texttt{RMW}^2 \triangleq \ell : \texttt{LX};\ \text{cmp};\ \text{bc}\ \ell';\ \texttt{SX};\ \text{bc}\ \ell;\ \ell' :$
$\texttt{RMW}^2_{\texttt{A}} \triangleq \ell : \texttt{LX}_{\texttt{A}};\ \text{cmp};\ \text{bc}\ \ell';\ \texttt{SX};\ \text{bc}\ \ell;\ \ell' :$
$\texttt{RMW}^2_{\texttt{L}} \triangleq \ell : \texttt{LX};\ \text{cmp};\ \text{bc}\ \ell';\ \texttt{SX}_{\texttt{L}};\ \text{bc}\ \ell;\ \ell' :$
$\texttt{RMW}^2_{\texttt{AL}} \triangleq \ell : \texttt{LX}_{\texttt{A}};\ \text{cmp};\ \text{bc}\ \ell';\ \texttt{SX}_{\texttt{L}};\ \text{bc}\ \ell;\ \ell' :$

Figure 1: Concurrency primitives in x86, TCG IR, and Arm which are used in the mapping schemes.

correctly. However, since many shared libraries are common across platforms, some libraries used by guest applications will also be available on the host system in native form. A classic example is the GNU C Library (glibc), which is used by most applications. This means that Qemu translates functions from the guest glibc, while a native and optimized version is almost certainly available.

2.3 TCG: Qemu's Dynamic Binary Translator

Qemu translates code through its TCG. Basic blocks are translated via an intermediate representation (IR) called the TCG IR. Architecture-independent optimizations are also applied on the basic blocks at the IR level.

TCG IR. The TCG IR is an assembly-like instruction set. It contains basic arithmetic, logic, and control flow instructions. However, floating-point arithmetic is emulated via integer-based computations.

Memory fences. The TCG IR provides fences for all types of pairs of accesses. For example, the Fww fence orders a store-store pair, while Frw orders a load-store pair. When generating fences in the IR, TCG takes the guest memory model into account to choose the fence accordingly. Section 3 provides a more detailed discussion.

Atomic read-modify-write (RMW). RMW accesses are currently translated into calls to helper functions in Qemu. Therefore, even if the host ISA has an equivalent atomic instruction, execution is still transferred from the emulated binary to Qemu. We discuss these primitives in Section 5.

Optimizations. TCG performs various optimizations on the translated basic blocks at the IR level. Some of the well-known optimizations are dead code elimination, constant propagation and folding, consecutive fence merging, etc.

2.4 Concurrency Primitives in Architectures

We categorize the concurrency primitives as follows: *(1)* load accesses that read from shared memory, *(2)* store accesses that write to shared memory, *(3)* RMW accesses that atomically update shared memory, and *(4)* fence operations that order memory accesses. Figure 1 lists the concurrency primitives from x86, Arm and TCG IR used in the mapping schemes discussed in this paper.

Load and store accesses. Most instructions in x86 can perform a memory access, so we denote the underlying x86 load and store operations as RMOV and WMOV. In Arm, LDR and STR perform the load and store operations.

In x86, RMOV-RMOV, RMOV-WMOV, WMOV-WMOV access pairs are always executed in order. In Arm, independent LDR and STR accesses on different locations may execute out-of-order.

Fence operations. The full fences in x86 and Arm are MFENCE and DMBFF respectively, which order any memory access pair. Arm also has lightweight fences, *e.g.,* DMBLD orders a read operation with its successors and DMBST orders a pair of writes.

RMW accesses. Both x86 and Arm provide various types of RMW primitives. x86 has the LOCK CMPXCHG instruction. Arm provides two types of RMW primitives that we denote by $\texttt{RMW}^2$ and $\texttt{RMW}^1$.

$\texttt{RMW}^2$ is constructed from load-exclusive (LX) and store-exclusive (SX). Arm also provides acquire-load-exclusive ($\texttt{LX}_{\texttt{A}}$) and release-store-exclusive ($\texttt{SX}_{\texttt{L}}$) instructions. A release access is ordered with its predecessors and an acquire is ordered with its successors. We can construct $\texttt{RMW}^2$, $\texttt{RMW}^2_{\texttt{A}}$, $\texttt{RMW}^2_{\texttt{L}}$, $\texttt{RMW}^2_{\texttt{AL}}$ primitives with these instructions as shown in Figure 1. $\texttt{RMW}^1$ denotes the single-instruction RMW instructions [6, 12]. Similar to $\texttt{RMW}^2$, $\texttt{RMW}^1$ accesses can also have release/acquire combinations as shown in Figure 1.

In x86, a successful RMW acts as a full fence whereas in Arm, only a successful $\texttt{RMW}^1_{\texttt{AL}}$ acts as a full fence.

3 MOTIVATION

In this section, we expose correctness and performance problems that arise when Qemu emulates concurrency. We also expose an error in an existing Arm mapping.

3.1 Emulation of Concurrent Programs in Qemu

Qemu does not officially support the emulation of strongly ordered ISAs, *e.g.,* x86, on weakly ordered ones, *e.g.,* Arm. However, in user mode emulation, the program runs without triggering any warning or error message to users, who can therefore think that support is available.

Qemu mapping schemes. Figure 2 shows Qemu's mapping schemes for translating memory-related x86 instructions to Arm. An Fmr fence is inserted before loads (RMOV), ordering the load with its preceding memory access. Since store-load reordering is allowed in x86, TCG demotes this fence to Frr, only ordering the load with a preceding load. This is an attempt to match the x86 memory model. An Fmw fence is inserted before stores (WMOV), ordering the store with its preceding memory access. These fences are then lowered to Arm's DMBLD and DMBFF fences.

RMW operations. Qemu translates RMW operations as calls to helper functions. These helper functions rely on GCC built-ins for the atomic accesses. As a result, depending on the GCC version, the instructions differ. For example, the helper function emulating the x86 CMPXCHG instruction uses an ldaxr-stlxr pair ($\texttt{RMW}^2_{\texttt{AL}}$) with GCC 9, but a casal instruction ($\texttt{RMW}^1_{\texttt{AL}}$) with GCC 10. Both are correct from GCC's standpoint since they both comply with the

x86		TCG IR		Arm
RMOV	→	Fmr; ld	→	DMBLD; LDR
WMOV	→	Fmw; st	→	DMBFF; STR
RMW	→	call	→	BLR; RMW; RET
MFENCE	→	F_{SC}	→	DMBFF

Figure 2: Qemu mappings: x86 to Arm via TCG IR.

x86		Arm
RMOV	→	LDR_Q
WMOV	→	STR_L
RMW	→	RMW^1_{AL}
MFENCE	→	DMBFF

Figure 3: Intended Arm mappings of Arm-Cats [6].

C/C++ memory model. However, this leads to inconsistencies for the x86 model.

3.2 Correctness: Errors in Qemu

We found several errors in Qemu's x86 to Arm translation, more specifically in handling RMW access (both RMW^1 and RMW^2). We demonstrate these errors by the translations of the MPQ and SBQ programs where RMW^1 and RMW^2 accesses are generated respectively.

We also show that the usage of F_{MR} fence in TCG IR may also result in an erroneous RAW transformation as demonstrated by the translation of the FMR program.

Error in mapping scheme with RMW^1_{AL}. Consider the x86 to Arm mapping by Qemu for the following program.

$$\begin{array}{l} X=Y=0; \\ X=1; \\ Y=1; \end{array} \Bigg\| \begin{array}{l} a=Y; \\ \textbf{if}\,(a==1) \\ \texttt{RMW}(X,1,2); \end{array} \quad \rightsquigarrow \quad \begin{array}{l} X=Y=0; \\ \texttt{DMBFF}; \\ X=1; \\ \texttt{DMBFF}; \\ Y=1; \end{array} \Bigg\| \begin{array}{l} \texttt{DMBLD}; \\ a=Y; \\ \textbf{if}\,(a==1) \\ \texttt{RMW}^1_{\texttt{AL}}(X,1,2); \end{array} \tag{MPQ}$$

In x86, $a = 1$ implies that all writes in the first thread are completed. Since reads are not reordered, the RMW always reads the $X = 1$ and successfully updates $X = 2$. As a result $a = 1, X = 1$ is never possible. In Arm, however, a read and a read-acquire pair can be reordered. This means that even though the first thread's writes are ordered by fences, the read of RMW^1_{AL} can be speculatively executed before the $a = Y$ instruction as they are unordered. In that case, the RMW^1_{AL} will not observe $X = 1$ and fail, but the result will still be committed after $a = Y$ sets a to 1. It results in the outcome $a = 1, X = 1$, which is disallowed in x86, hence an incorrect translation.

Error in mapping scheme with RMW^2_{AL}. Consider the following x86 to Arm translation.

$$\begin{array}{l} X=Y=Z=U=0; \\ X=1; \\ \texttt{RMW}(Z,0,1); \\ a=Y; \end{array} \Bigg\| \begin{array}{l} Y=1; \\ \texttt{RMW}(U,0,1); \\ b=X; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} X=Y=Z=U=0; \\ \texttt{DMBFF}; \\ X=1; \\ \texttt{RMW}^2_{\texttt{AL}}(Z,0,1); \\ \texttt{DMBLD}; \\ a=Y; \end{array} \Bigg\| \begin{array}{l} \texttt{DMBFF}; \\ Y=1; \\ \texttt{RMW}^2_{\texttt{AL}}(U,0,1); \\ \texttt{DMBLD}; \\ b=X; \end{array} \tag{SBQ}$$

The behavior in question is $Z = U = 1, a = b = 0$. In x86, successful RMW accesses order store-load access pairs in the executions. On the other hand, neither successful RMW^2_{AL} accesses nor DMBLD fences can order the store-load access pairs. Thus, the mapping results in a new outcome in the generated Arm program and, therefore, the overall translation is incorrect.

Error in RAW transformation in TCG IR. Qemu performs various constant propagation optimizations on TCG IR such as read-after write (RAW), *e.g.*, $Y = 2; a = Y; \rightsquigarrow Y = 2; a = 2;$. We note that in the presence of Fmr, the RAW transformation is incorrect as it introduces a new outcome. Consider the following example.

$$\begin{array}{l} X=Y=Z=0; \\ X=3; \\ \texttt{Fmr}; \\ Y=2; \\ a=Y; \\ \texttt{Frw}; \\ Z=2; \end{array} \Bigg\| \begin{array}{l} \textbf{if}\,(Z==2)\,\{ \\ \texttt{Frw}; \\ X=4; \\ c=X; \\ \} \end{array} \quad \rightsquigarrow \quad \begin{array}{l} X=Y=Z=0; \\ X=3; \\ \texttt{Fmr}; \\ Y=2; \\ a=2; \\ \texttt{Frw}; \\ Z=2; \end{array} \Bigg\| \begin{array}{l} \textbf{if}\,(Z==2)\,\{ \\ \texttt{Frw}; \\ X=4; \\ c=X; \\ \} \end{array} \tag{FMR}$$

Consider the outcome $a = 2, c = 3$. In the source TCG program, the Fmr and Frw fences in the first thread establish dependency-based ordering from $X = 3$ to $Z = 2$ via $a = Y$. In the second thread, Frw orders the read of Z with the successor accesses on X. As a result, the outcome $a = 2, c = 3$ is disallowed. The RAW transformation in the first thread remove the read of Y and hence $X = 3$ and $Z = 2$ are not ordered anymore. As a result, the $a = 2, c = 3$ outcome is allowed in target program, making the RAW transformation incorrect.

3.3 Correctness: Error in "Desired" Arm Mapping

We consider the x86 to Arm-Cats mapping [6]. While the authors do *not* explicitly give a mapping, we infer:

- LDAPR (LDR_Q) and STLR (STR_L) enable efficient emulation of x86-TSO on Arm-Cats [6, p.6]
- amo in Arm-Cats exclusively models RMW^1_{AL}, *e.g.*, casal, which *should act as a full barrier* [6, p.18].
- In x86, a successful RMW also behaves like a full barrier [6, 62].

We interpret their intended mapping as given in Figure 3.

While examining that mapping, we discover that it is incorrect following the memory models [6]. Consider the following example:

$$\begin{array}{l} X=Y=0; \\ \texttt{RMW}(X,0,1); \\ a=Y; \end{array} \Bigg\| \begin{array}{l} \texttt{RMW}(Y,0,1); \\ b=X; \end{array} \quad \rightsquigarrow \quad \begin{array}{l} X=Y=0; \\ \texttt{RMW}^1_{\texttt{AL}}(X,0,1); \\ a=Y_Q; \end{array} \Bigg\| \begin{array}{l} \texttt{RMW}^1_{\texttt{AL}}(Y,0,1); \\ b=X_Q; \end{array} \tag{SBAL}$$

The source x86 program *disallows* $X = Y = 1, a = b = 0$ as outcome, while the Arm program *allows* it. Therefore the mapping is erroneous.

Fixing this error. There are two options to fix this error in the model:

- Keep the current formal model and accept casal is insufficient to model x86 RMW.
- Strengthen the formal model slightly, so casal behaves like x86 RMW.

We choose the second option that we detail in Section 5. We hypothesize that hardware may already be consistent with our model.

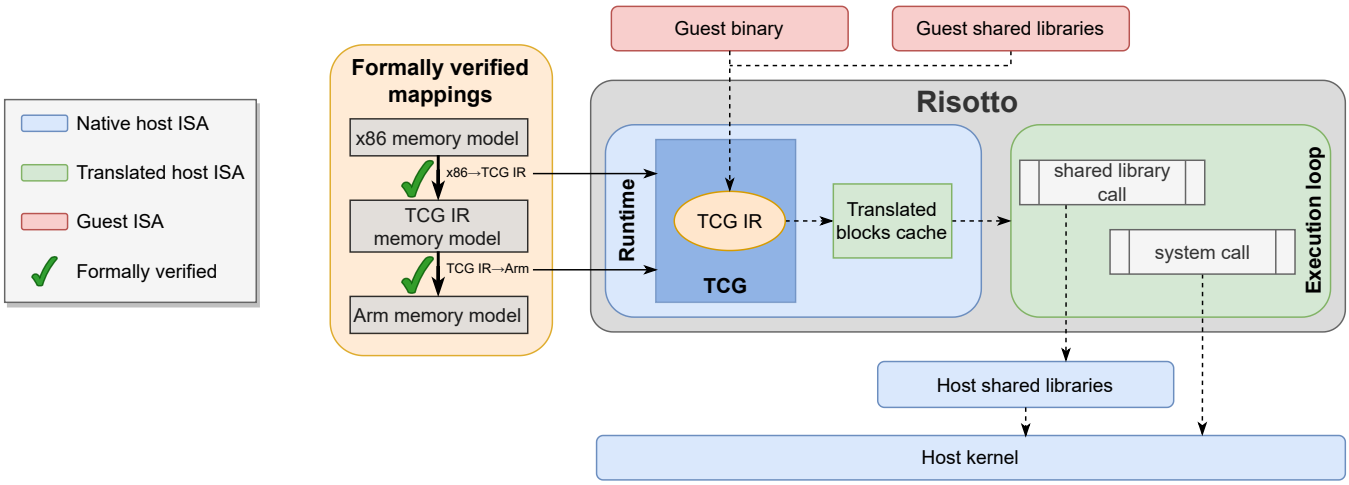

Figure 4: Overall architecture of Risotto.

We contacted the authors of Armed Cats, but they could not confirm hardware behavior with regards to our SBAL example in the new model. However, they still decided to strengthen the memory model like we proposed [37].

3.4 Performance: Fence and Shared Library Issues

Fence placement. Qemu's mapping schemes in Figure 2 prevent any reordering of memory accesses, even though the guest ISA (x86) allows some reorderings to happen. However, the CPU performs these reorderings to maximize its utilization. Not taking advantage of the CPU's instruction scheduling hurts performance. Additionally, having fences *before* every access makes it impossible to merge them.

Shared library. Qemu translates shared library functions from guest to host ISA, even when these same functions already exist on the host system in a native shared library. In general, translated code is less performant than natively compiled code, because the translation engine is unable to achieve the same level of optimization as the native compiler, when compiling from source code. Therefore, using pre-compiled native code when available, *i.e.,* the native version of a shared library, will lead to significant performance gains for guest programs that rely heavily on shared libraries.

4 OVERVIEW

We propose an end-to-end approach to improve the performance of strong-on-weak architecture DBT while maintaining semantic correctness.

4.1 Verified Mapping Schemes and Optimizations

We reason about the end-to-end translation steps: (1) x86 to TCG IR mapping (2) TCG IR to TCG IR optimization (3) TCG IR to Arm mapping.

TCG IR formalization. To reason about these steps formally, we use existing formal models of x86 and Arm [6], and propose a formalization of TCG IR. Based on this formalization, we ensure the correctness of the translations in all three steps.

Mappings in steps (1) & (3). We map the load, store, `RMW`, and fence accesses from the source to the corresponding accesses in the target models. The orderings between the accesses vary based on the consistency models. To ensure orderings between weaker accesses, we introduce additional leading or trailing fences along with the memory accesses. As fences are costly, our goal is to introduce only the minimal fences that are required to ensure correctness.

Moreover, we note that some TCG optimizations may perform read-after-write (RAW) transformations, which can introduce errors in the presence of Fmr or Fwr fences (see FMR example). Hence, we avoid generating any Fmr or Fwr fence in the x86 to TCG IR mapping scheme so that RAW transformations remain correct on the generated TCG IR programs.

Using all three formal models, we formally prove the correctness of the mapping schemes. These new schemes use minimal fences to preserve correctness.

IR transformations in step (2). Risotto performs several optimizations on the TCG IR before generating the Arm code. To ensure their correctness, we analyze the common transformations performed on the concurrency primitives. We show that the proposed TCG IR formalization allows the transformations performed by Risotto's optimizations.

More specifically, we reason about elimination of redundant shared memory accesses and reordering of shared memory accesses. We also reason about fence merging optimizations which can be performed when there are adjacent fences. Our x86 to TCG IR mapping scheme creates such adjacent fences which can be merged to improve performance as shown in Section 7.

4.2 Risotto: A Dynamic Binary Translator for Strong-on-Weak Architectures

We build Risotto upon the widely used emulator Qemu. We improve over the existing work through three contributions: *(i)* the implementation of formally verified memory mappings, *(ii)* a dynamic linker that uses host shared libraries instead of guest libraries, and *(iii)* a fast and correct translation of CAS instructions. Figure 4 shows the overall architecture of Risotto.

Memory mappings. We first replace the memory mapping schemes used by Qemu with our schemes presented in Section 5, which are formally verified to enforce the x86 memory model [6, 62, 63]. We implement this in the TCG shown in Figure 4, where the TCG IR code is generated. We also implement fence merging optimizations at the TCG IR level to minimize the cost of inserted fences.

Dynamic host linker. Qemu uses guest shared libraries that are translated to the host ISA. Since translated code is less efficient than native host code, maximizing the amount of native code used is a good way to improve performance. In Risotto, we target shared libraries to expand the amount of native code used because of their unique properties.

First, similar to system calls, they provide a clearly defined API to programs, which makes it possible to correctly marshal arguments and return values between guest and host ISAs. Second, even if a binary is only available for the guest ISA, the shared libraries that it uses may be available on the host ISA.

In Risotto, we implement a dynamic linker that connects invocations of shared library functions to native host shared libraries, instead of translating guest shared libraries (§ 6.2).

Fast and correct CAS. As previously stated, `RMW` primitives are emulated through a call to a helper function in Qemu and not translated. In addition to the performance hit, this can also trigger erroneous behaviors.

In Risotto, we aim at preserving correctness while maximizing performance. For atomic operations, we propose to translate the x86 atomic instructions, *e.g.,* `CMPXCHG`, directly into Arm assembly, *e.g.,* using the new `casal` instructions. This allows us to fix the errors in Qemu's current scheme as well as improve performance. We also implement this in the TCG (§ 6.3).

5 TCG IR CONCURRENCY MEMORY MODEL

In this section, we propose an axiomatic concurrency model for the TCG IR. Based on this model, we propose formally verified mapping schemes from x86 to Arm via the TCG IR.

5.1 Axiomatic Model for Concurrency

In axiomatic semantics, a program is represented by a set of finite executions where an execution constitutes of a set of events and relations. An event is generated from the execution of a shared memory access or a fence and the events are related by various relations. We can represent an execution as a graph where the nodes represent events and edges represent relations [6, 10, 18, 41].

The set of read, write, and fence events are R, W, and F respectively. The events are connected by various relations.

Notations. To define the formal models we use relation and set notations (similar to 'cat' notations [8]). Given a binary relation S on events, $\mathrm{dom}(S)$ and $\mathrm{codom}(S)$ are domain and range of S. We compose binary relations S_1 and S_2 by $S_1;S_2$. $[A]$ is an identity relation on a set A. Finally, on an execution graph, relation S is acyclic if S^+ (transitive closure of S) is irreflexive.

Relations. The events are primarily connected by program-order (po), reads-from (rf), coherence-order (co) relations. Relation po is a strict partial order that captures the syntactic order among the events, rf relates a pair of write and read events on same-location having same values, and co is a strict total order on same-location write events. We compose these relations to derive new ones.

- Relation from-read ($\mathsf{fr} \triangleq \mathsf{rf}^{-1};\mathsf{co}$) relates a read and write events r and w on same-location (rf^{-1} is inverse of rf). In this case, w is co-after the write u where $\mathsf{rf}(u,r)$ holds.
- A relation is external when it is not between po-related events, *e.g.,* external rf, co, fr relations are:
  $\mathsf{rfe} \triangleq \mathsf{rf} \setminus \mathsf{po}$ $\mathsf{coe} \triangleq \mathsf{co} \setminus \mathsf{po}$ $\mathsf{fre} \triangleq \mathsf{fr} \setminus \mathsf{po}$
- Relation rmw connects a pair of read and write events accessing same memory location. These two events are same-location-po-related ($\mathsf{po}|_{\mathsf{loc}}$) as well as immediate-po-related ($\mathsf{po}_{\mathsf{im}}$), *i.e.,* there is no intermediate event ($\mathsf{po}_{\mathsf{im}}\, x\, y \triangleq \mathsf{po}\, x\, y \wedge \nexists z.[\mathsf{po}\, x\, z \wedge \mathsf{po}\, z\, y]$).

Execution. Given an execution $\mathsf{X} = \langle \mathsf{E}, \mathsf{po}, \mathsf{rf}, \mathsf{co}\rangle$, X.E is the set of events, and X.po, X.rf, X.co are the set of po, rf, and co relations between the events in X.E. In an execution, all the memory locations are initialized.

From programs to executions. A program consists of the initialization of all shared memory locations followed by a parallel composition of threads. In a program, the concurrency primitives generate the events and relations during an execution. In an execution, we do not capture thread-local operations and accesses explicitly. However, we can always augment a program with additional shared variables to observe the values of thread-local variables.

Behavior. Given an execution, the final values of all memory locations define its behavior, *i.e.,* the values written by the writes which have no co-successors.

$$\mathrm{Behav}(\mathsf{X}) \triangleq \{\langle e.\mathsf{loc}, e.\mathsf{val}\rangle \mid e \in \mathsf{X.W} \wedge [\{e\}];\mathsf{X.co} = \emptyset\}$$

Consistency axioms. Based on these relations and events, we define the consistency axioms for a model. The consistency axioms capture certain architectural properties which are satisfied in an execution. If an execution satisfies all the axioms of the model, then it is consistent in that model. The set of consistent executions of program $\mathbb{P}$ in memory model M is denoted by $[[\mathbb{P}]]_M$. The set of behaviors exhibited by the consistent executions constitute the program behavior.

5.2 x86 and Arm Concurrency Models: A Preview

We briefly discuss the axiomatic models of x86 and Arm [6, 8, 10].

- An x86 `RMOV` or Arm `LDR` generates a read (R) event and an x86 `WMOV` or Arm `STR` generates a write (W) event.
- In both x86 and Arm, a *successful* `RMW` generates a pair of rmw-related events. In x86, these events are [R]; rmw; [W] related. In Arm, we categorize the rmw relations as amo and lxsx relations which result from $\texttt{RMW}^1$ and $\texttt{RMW}^2$ primitives. So, in Arm, rmw = lxsx ∪ amo holds. If an `RMW` fails in x86 or Arm, it generates an R event only.

(external) axiom
ob is irreflexive where

$$ob \triangleq (rfe \cup coe \cup fre \cup lob)^+$$
$$lob \triangleq (lws \cup dob \cup aob \cup bob)^+$$
$$\begin{aligned} bob \triangleq\ & po;[F];po \cup [R];po;[F_{LD}];po \\ & \cup [W];po;[F_{ST}];po;[W] \\ & \cup po;[dom([A];amo;[L])] \cup [codom([A];amo;[L])];po \\ & \cup \cdots \end{aligned}$$

Figure 5: Arm-Cats Arm model (corrected)

(GOrd) axiom
ghb is irreflexive where

$$ghb \triangleq (ord \cup rfe \cup coe \cup fre)^+$$
$$\begin{aligned} ord \triangleq\ & [R];po;[F_{RR}];po;[R] \cup [R];po;[F_{RW}];po;[W] \\ & \cup [R];po;[F_{RM}];po;[R \cup W] \cup [W];po;[F_{WR}];po;[R] \\ & \cup [W];po;[F_{WW}];po;[W] \cup [W];po;[F_{WM}];po;[R \cup W] \\ & \cup [R \cup W];po;[F_{MR}];po;[R] \cup [R \cup W];po;[F_{MW}];po;[W] \\ & \cup [R \cup W];po;[F_{MM}];po;[R \cup W] \\ & \cup po;[W_{SC} \cup dom(rmw)] \cup [R_{SC} \cup codom(rmw)];po \\ & \cup po;[F_{SC}] \cup [F_{SC}];po \end{aligned}$$

Figure 6: Proposed TCG IR model. TCG also satisfies the (sc-per-loc) and (atomicity) axioms, similarly to x86 and Arm.

- An x86 `MFENCE` or Arm `DMBFF` generates an F event.

Arm also generates events and relations for lightweight fences and synchronizing memory accesses.

- `DMBLD` and `DMBST` fences generate F_{LD} and F_{ST} events.
- Release store (*e.g.,* STR_L), acquire load (*e.g.,* LDR_A), acquirePC-load (*e.g.,* LDR_Q) generate L, A, Q events respectively. L is ordered with its predecessors, A and Q are ordered with its successors, and a L is ordered with its successor A event. Finally, $L \subseteq W$, $A \subseteq R$, and $Q \subseteq R$ hold.

Common features. Both x86 and Arm ensure coherence and atomicity which are captured by these axioms.

Coherence: The property enforces *SC-per-location* in an execution: the memory accesses per memory locations are totally ordered. The property is captured by (sc-per-loc) axiom: $(po|_{loc} \cup rf \cup co \cup fr)^+$ is irreflexive.

Atomicity: The read and write pair generated from a successful `RMW` access is atomic. Suppose r and w are rmw related read and write events. If there exists a write event w' between r and w, and X.fre(r, w') and X.coe(w', w) hold, then the execution violates atomicity. Both x86 and Arm restrict atomicity violation by (atomicity) axiom: $rmw \cap (fre; coe) = \emptyset$.

Distinguishing x86 and Arm concurrency. Now, we discuss the relations and axioms that differentiate the x86 and Arm formal models.

x86: The read-read, read-write, write-write event pairs are ordered by preserved-program-order (ppo) relation. In addition, access pairs are ordered by intermediate rmw or F accesses which is captured by implied relation. Using these relations, x86 defines (GHB) axiom which enforces a global order.

(GHB) $(implied \cup ppo \cup rfe \cup fr \cup co)^+$ is irreflexive where
$ppo \triangleq ((W \times W) \cup (R \times W) \cup (R \times R)) \cap po$
$implied \triangleq po;[At \cup F] \cup [At \cup F];po$
where $At \triangleq dom(rmw) \cup codom(rmw)$

Arm: In Figure 5, we show the (external) axiom from the official Arm model, with some revisions detailed later. Arm defines a transitive relation locally-ordered-before (lob) to order events in a thread. Relation lob has the following components:

- Relation local-write-successor (lws) orders a memory event to a same-location po-successor write event.
- Relation atomic-ordered-by (aob) is based on rmw.
- Relation dependency-ordered-before (dob) is derived from data, address, and control dependencies from a read to another write, memory accesses, and all events respectively.
- Relation barrier-ordered-by (bob) is based on fences and synchronizing memory accesses.

We discovered an undesirable scenario in the existing model, as elaborated in subsection 3.3. To ensure the `casal` instruction *acts as a full barrier*, we propose a fix to the model, where we replace po; [A]; amo; [L]; po in bob, which we marked green in Figure 5.

5.3 Formalizing TCG IR Concurrency

We begin with the TCG primitives along with generated events and relations in an execution.

Load and store accesses. TCG provides load (ld) and store (st) operations that respectively read and write shared memory locations. ld and st accesses generate R and W events.

Fence accesses. TCG provides different types of fences: Frr, Frw, Fww, Fwr, Facq, Frel, and Fsc. These fences generate F_{RR}, F_{RW}, F_{WW}, F_{WR}, F_{ACQ}, F_{REL}, and F_{SC} events respectively. They can be combined to define stronger fences, *e.g.,* we combine Frr and Frw to define Frm, that generates an F_{RM} event for the proposed mapping schemes. All these fences order certain memory accesses which we capture in *order* (ord) relations. For instance, a pair of po-related events (a, b) are in ord relation if a and b are W events with an intermediate F_{WW} event following the [W]; po; [F_{WW}]; po; [W] rule.

`RMW` accesses. TCG also provides a number of atomic read-modify-write (`RMW`) operations. These atomic `RMW` accesses follow SC semantics and do not allow reordering with other accesses. A successful `RMW` generates a rmw-related R_{SC} and W_{SC} event pair, *i.e.,* [R_{SC}]; rmw; [W_{SC}]. A failed `RMW` generates a R_{SC} event. Finally, $R_{SC} \subseteq R$ and $W_{SC} \subseteq W$ hold in the model. Events generated from `RMW` accesses also enforce ord relation as shown in the ord definition.

Finally, we define global-happen-before (ghb) relation to order events across different threads. On an execution graph, ghb(a, b) implies that there is a path from a to b by ord and external relations rfe, coe, fre.

Axioms. Based on these relations, we define the consistency constraints. Similar to x86 and Arm, the TCG IR model also includes the (sc-per-loc) and (atomicity). axioms. The (GOrd) axiom in Figure 6 ensures a global order between events.

x86	TCG IR
RMOV	ld; Frm
WMOV	Fww; st
RMW	RMW
MFENCE	Fsc

(a) x86 to TCG IR.

TCG IR	Arm
ld	LDR
st	STR
RMW	DMBFF; $\texttt{RMW}^2$; DMBFF or $\texttt{RMW}^1_{\text{AL}}$
Frr/Frw/Frm	DMBLD
Fww	DMBST
Fwr/Fmm/Fsc	DMBFF
Facq/Frel	-

(b) TCG IR to Arm.

x86		TCG IR		Arm
RMOV	→	ld; Frm	→	LDR; DMBLD
WMOV	→	Fww; st	→	DMBST; STR
RMW	→	RMW	→	DMBFF; $\texttt{RMW}^2$; DMBFF or $\texttt{RMW}^1_{\text{AL}}$
MFENCE	→	F_{SC}	→	DMBFF

(c) x86 to Arm via TCG IR.

Figure 7: Verified mapping schemes for x86 to Arm via TCG IR.

5.4 Verified Mappings and Transformations

Based on the proposed IR model, we verify the correctness of the transformation (mappings and transformations) steps.

Theorem 1 (Transformation Correctness). *Suppose a given source program* $\mathbb{P}_s$ *in model* M_s *is transformed to the target program* $\mathbb{P}_t$ *in model* M_t. *The transformation is correct if for each consistent target execution* $\mathsf{X}_t \in [[\mathbb{P}_t]]_{\mathsf{M}_t}$ *there exists a consistent source execution* $\mathsf{X}_s \in [[\mathbb{P}_s]]_{\mathsf{M}_s}$ *such that* $\text{Behav}(\mathsf{X}_t) = \text{Behav}(\mathsf{X}_s)$.

For mapping schemes, M_s and M_t differ. For transformations, M_s and M_t are the same.

Correct mapping schemes. We translate concurrency primitives from x86 to Arm in two steps: (1) x86 to TCG IR and (2) TCG IR to Arm. We formally prove Theorem 1 to ensure correctness of these mapping schemes. These mapping schemes are *precise*: each placed fence is necessary in some program. Yet, it is sufficient to preserve the required ordering in every program.

x86 to IR mapping scheme. The mapping scheme is in Figure 7a. It introduces additional fences along with the load and store accesses to enforce the same restrictions as x86. In order to ensure correctness, we prove Theorem 1.

The x86 to IR mapping scheme is minimal. In x86 load-load and load-store accesses are ordered (formally by ppo) unlike that of IR. To enforce these orderings (formally ord) in the generated programs we require the trailing and leading fences with load and store respectively as shown in Figure 8.

$$\begin{array}{l} X = Y = 0; \\ a = X; \,||\, b = Y; \\ Y = 1; \,||\, X = 1; \end{array} \quad \rightarrow \quad \begin{array}{l} X = Y = 0; \\ a = X; \,||\, b = Y; \\ \text{Frw}; \,||\, \text{Frw}; \\ : \,||\, : \\ Y = 1; \,||\, X = 1; \end{array} \qquad \text{(LB-IR)}$$

Disallowed outcome $a = b = 1$.

$$\begin{array}{l} X = Y = 0; \\ X = 1; \,||\, a = Y; \\ Y = 1; \,||\, b = X; \end{array} \quad \rightarrow \quad \begin{array}{l} X = Y = 0; \\ : \,||\, a = Y; \\ X = 1; \,||\, \text{Frr} \\ \text{Fww}; \,||\, b = X; \\ Y = 1; \,||\, : \end{array} \qquad \text{(MP-IR)}$$

Disallowed outcome $a = 1, b = 0$.

Figure 8: LB-IR and MP-IR disallow $a = b = 1$ and $a = 1, b = 0$ by enforcing ld-st and ld-ld orders using at least Frw and Frr fences. We combine these fences and insert a trailing Frm with a load in the x86 to IR mapping. The leading Fww fence orders st-st in MP-IR. Hence we introduce a leading Fww with a store access in the x86 to IR mapping.

IR to Arm mapping scheme. The mapping scheme is in Figure 7b. We prove the correctness theorem to ensure that the mapping scheme preserves correctness.

The IR to Arm mapping scheme is minimal. We analyze the fences in this mapping. If a TCG RMW generates $\texttt{RMW}^2$ access then it introduces leading and trailing DMBFF fences. These fences are required to preserve the mapping correctness as shown in Figure 9. The mapping scheme generates a DMBLD from a Frr/Frw/Frm fence in the IR to preserve the order of a load with its successor memory accesses. A Fwr/Fmm/Fsc fence in the IR generates a DMBFF fence to preserve the order between store-load pair on different locations. The Facq and Frel fences do not generate any instruction in Arm.

The examples in Figure 9 show that the DMBFF fences with $\texttt{RMW}^2$ accesses are required to preserve the mappings.

x86 to IR to Arm mapping. In Figure 7c, we combine the translations from x86 TCG and from TCG to Arm to obtain x86 to Arm translation.

Optimizing transformations. We formally prove Theorem 1 for various transformations on the concurrency primitives in the TCG IR. The verified transformations ensure the correctness of the translations in Risotto.

Memory access eliminations: TCG performs constant propagation and folding on the IR. These transformations may also be performed on shared memory accesses. Hence, we prove the correctness of the following transformations on executions, where $a \cdot b$ denotes po_{im}-related events with the labels a and b. The memory access pairs are on the same-location and may have any type of intermediate fences denoted by F_o where $o \in \{\text{RM}, \text{WW}\}$, or F_τ where $\tau \in \{\text{SC}, \text{WW}\}$.

Fence merging: It is correct to merge a fence to a same or stronger fence. We can also strengthen a fence to a stronger fence. We can combine these transformations as follows:

$$\text{F}_{\text{RM}} \cdot \text{F}_{\text{WW}} \xrightarrow{\text{strengthen}} \text{F}_{\text{SC}} \cdot \text{F}_{\text{SC}} \xrightarrow{\text{merge}} \text{F}_{\text{SC}}$$

$$\begin{array}{c} X = Y = 0; \\ X = 2; \;\|\; Y = 2; \\ \texttt{RMW}(Y,0,1); \;\|\; \texttt{RMW}(X,0,1); \end{array} \rightarrow \begin{array}{c} X = Y = 0; \\ X = 2; \;\|\; Y = 2; \\ \texttt{DMBFF}; \;\|\; \texttt{DMBFF}; \\ \texttt{RMW}^2(Y,0,1); \;\|\; \texttt{RMW}^2(X,0,1); \\ \texttt{DMBFF}; \;\|\; \texttt{DMBFF}; \end{array}$$

Disallowed outcome $X = Y = 1$.

$$\begin{array}{c} X = Y = 0; \\ \texttt{RMW}(X,0,1); \;\|\; \texttt{RMW}(Y,0,1); \\ a = Y; \;\|\; b = X; \end{array} \rightarrow \begin{array}{c} X = Y = 0; \\ \texttt{DMBFF}; \;\|\; \texttt{DMBFF}; \\ \texttt{RMW}^2(X,0,1); \;\|\; \texttt{RMW}^2(Y,0,1); \\ \texttt{DMBFF}; \;\|\; \texttt{DMBFF}; \\ a = Y; \;\|\; b = X; \end{array}$$

Disallowed outcome $a = b = 0$.

Figure 9: The DMBFF fences preserve correctness in IR to Arm mapping. The outcomes are disallowed in the IR model. Arm would allow these outcomes without the intermediate DMBFF fences and the translations would be incorrect.

$\mathsf{R}(X,v) \cdot \mathsf{R}(X,v')$	$\rightsquigarrow \mathsf{R}(X,v)$	(RAR)
$\mathsf{W}(X,v) \cdot \mathsf{R}(X,v)$	$\rightsquigarrow \mathsf{W}(X,v)$	(RAW)
$\mathsf{W}(X,v) \cdot \mathsf{W}(X,v')$	$\rightsquigarrow \mathsf{W}(X,v')$	(WAW)
$\mathsf{R}(X,v) \cdot \mathsf{F}_o \cdot \mathsf{R}(X,v')$	$\rightsquigarrow \mathsf{R}(X,v) \cdot \mathsf{F}_o$	(F-RAR)
$\mathsf{W}(X,v) \cdot \mathsf{F}_\tau \cdot \mathsf{R}(X,v)$	$\rightsquigarrow \mathsf{W}(X,v) \cdot \mathsf{F}_\tau$	(F-RAW)
$\mathsf{W}(X,v) \cdot \mathsf{F}_o \cdot \mathsf{W}(X,v')$	$\rightsquigarrow \mathsf{F}_o \cdot \mathsf{W}(X,v')$	(F-WAW)

Figure 10: Elimination Transformations

Reordering: The plain memory accesses are unordered in TCG IR unlike in x86, and hence can be reordered freely. The proposed TCG IR model allows the reorderings of independent memory access pairs on different locations. Moreover, dependencies do not enforce any ordering in TCG IR unlike that of Arm, and hence TCG can remove false dependencies. These transformations are formally correct as the TCG IR model do not order accesses based on dependencies.

We prove that reordering $a \cdot b \rightsquigarrow b \cdot a$ is correct where a and b are the labels of non-RMW memory events which are independent and access different memory locations.

Mechanized Proofs: We prove the correctness of our transformations – from some source program $\mathbb{P}_{\text{src}}$ to a target program $\mathbb{P}_{\text{tgt}}$ – in three steps. First, given a M_t-consistent execution X_t of $\mathbb{P}_{\text{tgt}}$, we define a source execution X_s from $\mathbb{P}_{\text{src}}$. Secondly, we relate the relations in M_s and M_t to show that X_s satisfies the axioms in M_s, because X_t satisfies those of M_t. Finally, we show that the $\mathsf{X}_\mathsf{t}.\mathsf{co}$ and $\mathsf{X}_\mathsf{s}.\mathsf{co}$ relations match, which means X_t and X_s have identical behaviors.

We mechanize all proofs in 14,000 lines of Agda [4].

6 RISOTTO SYSTEM ARCHITECTURE

Risotto is based on Qemu 6.1.0 [69]. In Risotto, we implement our verified mapping schemes, a dynamic linker to use host shared libraries and a fast and correct translation of the x86 CAS instructions.

6.1 Formally Verified Memory Mappings

We implement the formally verified memory mappings described in Section 5.3 in Risotto. More precisely, we implement the mapping schemes from Figure 7. We obtain the following performance benefits compared to the existing Qemu implementation.

Lightweight fences. Compared to Qemu that generates Fmr and Fmw fences before load and store operations, we generate Frm and Fww fences in the TCG IR. While Qemu's fences end up as a DMBLD or DMBFF fence, our scheme produces either a DMBLD or a DMBST fence. These fences are less costly in terms of performance than full fences [49].

Newly allowed reorderings. Enforcing the proper x86 model also allows for reorderings of memory operations that were not possible with Qemu. Indeed, in our mapping scheme, there is no fence between a store and a load access. This allows store-load access pairs to be freely reordered by the processor if there is no dependency between them.

Fence merging optimizations. We implement an optimization pass over the TCG IR to merge fences that have no intermediate memory access. We merge the fences as a stronger one that suffices, and place it where the earliest fence was. As an example, we show the translation of a program from x86 to Arm: (1) x86 to TCG IR following Figure 7a, (2) fence merging, and (3) TCG IR to (4) Arm following Figure 7b.

$$\begin{array}{l} a = X; \\ Y = 1; \end{array} \rightsquigarrow \begin{array}{l} a = X; \\ \mathsf{Frm}; \\ \mathsf{Fww}; \\ Y = 1; \end{array} \rightsquigarrow \begin{array}{l} a = X; \\ \mathsf{Fsc}; \\ Y = 1; \end{array} \rightsquigarrow \begin{array}{l} a = X; \\ \texttt{DMBFF}; \\ Y = 1; \end{array}$$

False dependency elimination. We perform false dependency elimination (*e.g.*, $X = a * 0 \rightsquigarrow X = 0$) on the TCG IR. It is trivially correct as the TCG IR model does not use dependency relations for any ordering, unlike in Arm.

6.2 Dynamic Host Library Linker

In order to use host shared library functions, Risotto must detect when the emulated program calls a shared library function, and link to the host library instead of emulating the guest linker and guest library function.

Supporting host shared libraries. To support native shared library execution, the shared library functions in use must be described to the DBT runtime, so that translated guest code can correctly transition to and from native library execution. The transition process translates function arguments from the guest representation to the host, and return values back from the host representation to the guest.

Function signature descriptions are necessary because parameters types, and their semantics are not specified in the raw application binary, and this information is necessary to perform parameter values translation. The runtime effectively needs to map the guest calling convention to the host's, which requires the parameter types must be known, so that appropriate value marshaling can take place. This would be unnecessary if both the host and guest Application Binary Interfaces (ABIs) were fully compatible. In our setup, we have no control over the OS, which means we cannot change the ABI to make it compatible across ISAs.

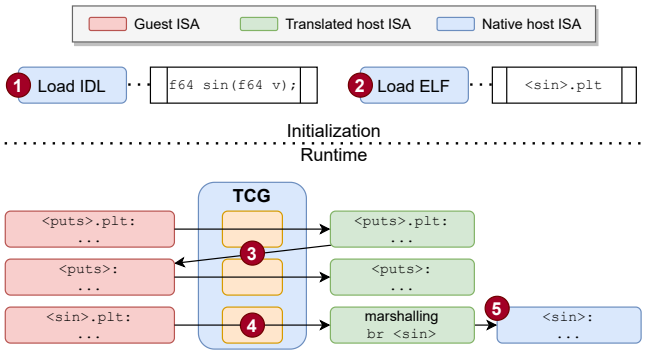

Figure 11: Risotto's dynamic linker workflow.

To describe function signatures to the runtime, we introduce an *Interface Definition Language* (IDL) that provides this information at run time to the translation system. Our IDL describes function signatures in a form similar to C function prototypes.

Capturing shared library calls. The key idea is to detect calls to shared library functions in the guest program, and, instead of performing binary translation as usual, emit code that directly calls the host shared library function. To do this, we exploit the dynamic linking mechanism of the ELF binary format. ELF files use a Procedure Linkage Table (PLT) that contains short code sequences that transfer control to the dynamic linker when a shared library function is invoked. Each imported shared library function has a PLT entry.

All shared library calls are made via the corresponding PLT entries – application code makes a call to the PLT entry when it wants to invoke such a function. When Risotto encounters a PLT entry, instead of translating the routine, it generates a code sequence that directly calls the host version of the shared library function.

Sequence of events. Figure 11 show the workflow of our linking mechanism. First, we read the IDL file, which identifies the shared library functions that are to be executed natively, and store the function signature information (1). Then, as Risotto loads the guest ELF binary, it parses the `.dynsym` section to determine the shared library functions that the program imports. For each detected function, the signature is looked up, and if present, *i.e.,* it has been described in the IDL, the corresponding PLT entry is located in the binary. The address of the PLT entry, along with a pointer to the function signature description, is stored in a lookup table (2).

When Risotto is about to translate a basic block, the address of the block is checked in the lookup table. If it was not specified in the IDL, the PLT entry as well as the guest library function are translated, as shown with the `puts` function (3). If it was, we generate code to marshal the function arguments from guest to host representation (4), and ultimately call the host function directly, as shown with the `sin` function (5). In practice, for Arm and x86, guest register values are copied into host register values, and vice-versa for the function return value.

Discussion on correctness. Using native libraries may cause inconsistencies due to the mismatch in memory models. If it is stronger than the guest model, then there is no problem. If it is weaker, incorrect behaviors only happen if the shared library function and emulated code interact with the same data location concurrently, which we have not observed in our applications.

6.3 Fast and Correct CAS Instructions

As previously detailed, Qemu translates CAS operations as calls to helper functions that in turn rely on GCC built-ins. In order to avoid the correctness problems this creates, as well as the performance degradation due to unnecessary jumps, we design a direct translation of CAS instructions. In particular, we target the translation of the x86 `CMPXCHG` instruction to Arm.

Risotto directly translates the x86 `CMPXCHG` instruction to the Arm `casal` instruction, without using a helper function. We do this by adding a new instruction to the TCG IR, `CAS`. Instructions implementing a CAS semantic in the guest ISA are translated to this new TCG IR instruction if the host supports native CAS. Otherwise, the usual call to the helper function is generated. When translating back from TCG IR to the host ISA, the `CAS` instruction is translated to the corresponding host CAS instruction. More specifically, in Arm, we translate it to a `casal` instruction.

Correctness. We follow the mapping schemes from Figure 7 for the `RMW` translation. x86 `RMW` acts as a full fence, and only a successful $\texttt{RMW}^1_{\texttt{AL}}$ in Arm does the same (see Section 2.4). Since `CMPXCHG` is an x86 `RMW` and Arm's `casal` is an $\texttt{RMW}^1_{\texttt{AL}}$, both have the same semantics in terms of ordering, making our translation correct.

7 EVALUATION

We evaluate Risotto's overall performance (§ 7.2), dynamic library linker (§ 7.3) and CAS translation (§ 7.4).

7.1 Experimental Setup

Testbed. We perform our evaluation on a server equipped with two Marvell ThunderX2 CN9975 processors (ARMv8.1, 28 cores per chip, 4-way SMT, 2.0 GHz), 256 GB of DDR4 memory (4×64 GB, 3200 MHz, ECC) and a 960 GB SSD (SATA 6Gb/s).

Benchmark suites and applications. We perform our evaluation on a set of applications from two benchmark suites: PARSEC 3.0 [19] and Phoenix [70]. For PARSEC, we omit the `raytrace` and `x264` benchmarks because they respectively fail to build and run natively on Arm.

Setups. We run our experiments on Qemu v6.1.0, as well as multiple variants: one that does not enforce any ordering, *i.e.,* no fences generated, noted `no-fences`; one that enforces our verified mappings, noted `tcg-ver`, and finally Risotto, with all features described previously. Note that `no-fences` is incorrect, but still serves as an oracle of the maximal performance improvement possible when optimizing ordering fences. We also run binaries natively, *i.e.,* Arm binaries without emulation, to show the performance gap. In every plot, the red line shows the average performance of Qemu, with the raw values in red.

Methodology. We run every experiment five times and compute the speedups compared to our baseline, Qemu. For better reproducibility, we disable turbo boost and use the `performance` scaling governor of Linux which uses a fixed frequency of 2.0 GHz on our CPU. We also pin our experiments on a single socket to avoid

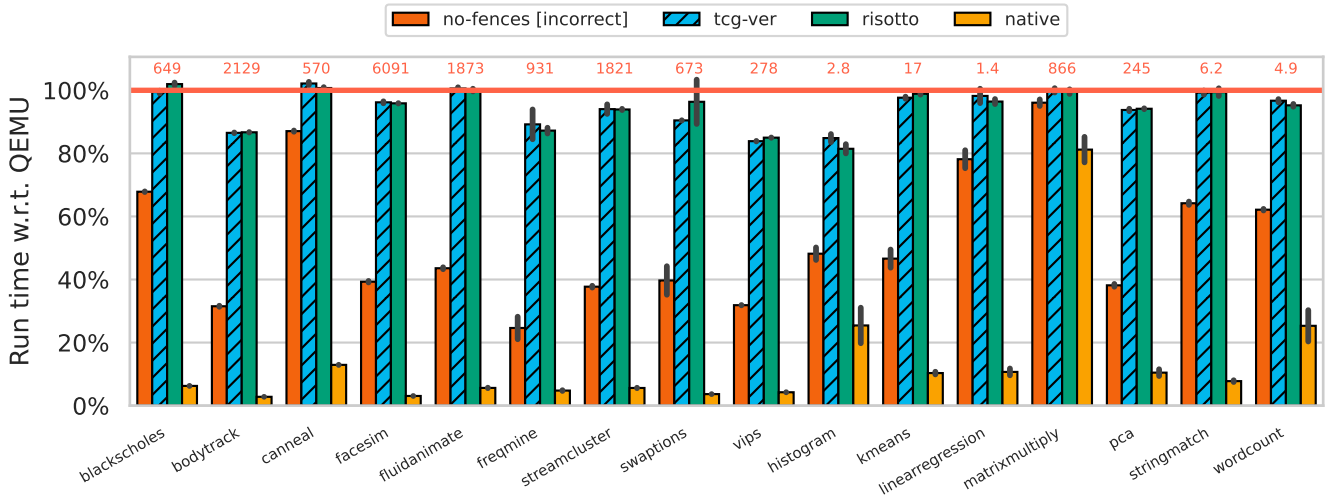

Figure 12: Run time of PARSEC and Phoenix benchmarks, running with Qemu with no fence generation (`no-fences`), Qemu with our verified mappings (`tcg-ver`) and Risotto, relative to Qemu. Native execution is also shown (`native`). Lower is better, raw values in seconds.

NUMA effects, therefore using 112 hardware threads. We compile all variants of Qemu and Risotto with GCC 10.3.0.

7.2 Overall Performance

First, we evaluate the raw performance of Risotto on PARSEC and Phoenix benchmarks. Figure 12 shows the performance results relative to the baseline, Qemu (red horizontal line), lower is better.

Cost of memory ordering enforcement. In order to better understand Risotto's performance, we first analyze the cost of Qemu's fence mapping. By observing the performance of `no-fences`, we see that fences account for a large portion of the execution time of our benchmarks, up to 75% (for `freqmine`), 48% on average. These results highlight the importance of reducing overhead associated with fences while still preserving its correctness.

Risotto's verified mappings. `tcg-ver` achieves significant performance gains without compromising the program's correctness. Compared to Qemu, we achieve improvements of up to 19.7% (6.7% on average), thanks to fence merging and weaker fence usage (Section 6.1).

7.3 Dynamic Host Library Linker

We evaluate Risotto's dynamic linker on well-known libraries that are extensively used in real-world applications. We evaluate the OpenSSL cryptography library [61] (*libssl, libcrypto*), the sqlite3 database engine [79] (*libsqlite*) and a stress microbenchmark on the standard math library (*libm*).

OpenSSL and sqlite. We run popular digests and ciphers with OpenSSL 1.1.1, such as RSA, MD5, SHA-1, and SHA-256, with different input sizes, as well as the speedtest benchmark of sqlite. We measure their throughput, *i.e.,* sign/s, verify/s or ops/s. We also run the sqlite speedtest1 benchmark and report its throughput.

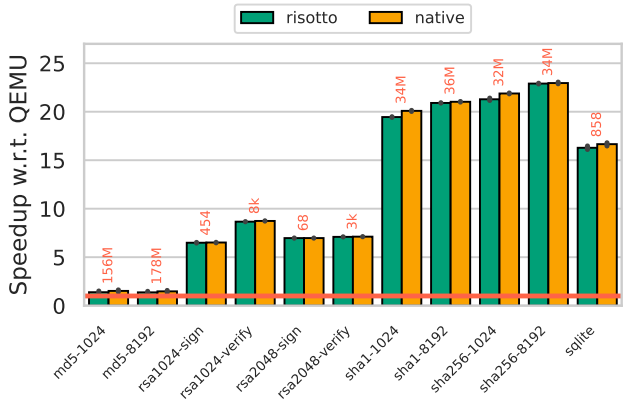

Figure 13: Speed-up of openssl and sqlite benchmarks against Qemu. Higher is better, raw values in ops/s.

Figure 13 shows the speedup of both Risotto and the native version over Qemu. Speedups vary from 1.4× (md5-1024) to 23× (sha256-8192), on a par with the native execution. Overall, we match the speed of native execution of shared libraries when using our dynamic host linker.

Math library. We evaluate the performance gains on functions from the standard math library. We run these functions 100M times and compute their throughput. Results are shown in Figure 14, with Risotto and native compared to Qemu. We observe speedups ranging from 1× (sqrt) to 10× (cos) with Risotto. Even though we significantly improve performance, we do not match the native version, that achieves up to 25× speedups. This difference is explained by the short duration of the library calls, preventing the overhead of argument marshaling to be amortized.

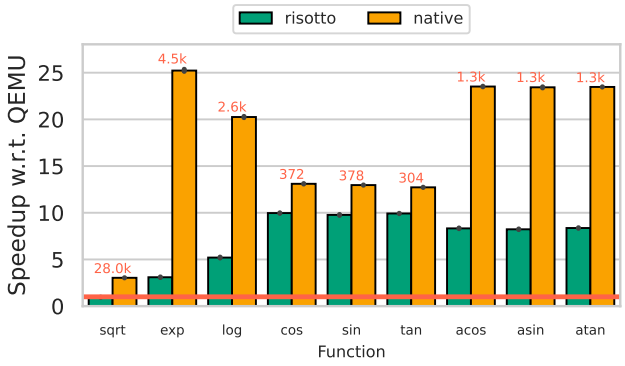

Figure 14: Speed-up of math library functions with Risotto compared to Qemu. Higher is better, raw values in ops/ms.

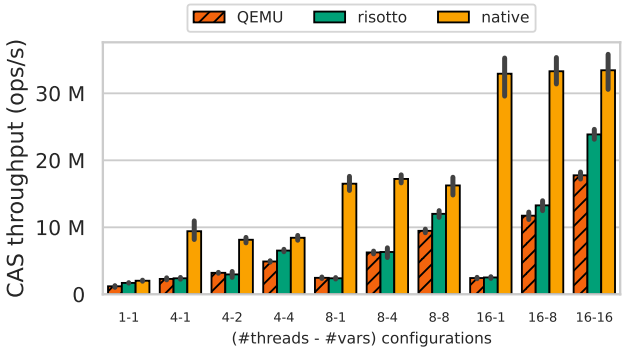

Figure 15: Throughput of the CAS instruction with various levels of contention. Higher is better.

Floating point emulation. Using host libraries for functions with floating point (FP) computation offers another benefit. Correctly emulating FP instructions across the variety of implementations is hard, and so Qemu implements a software floating-point implementation, drastically impacting performance. By using host shared libraries, we can take advantage of native FP instructions, adding to the performance improvement, as exposed by the math library benchmark.

Overhead of host library calls. Calling a host library function instead of a guest one requires to perform argument marshaling (§6.2). The OpenSSL and sqlite results show that there is no overhead in performing host shared library calls. However, the math library results show a clear difference between Risotto and native execution (Figure 14). This stems from the duration of the linked functions. Math functions are very short, meaning that argument marshaling dominates the execution time. This is not the case with the other benchmarks, where functions have a longer duration. Still, even in the worst-case scenario, using host libraries is clearly beneficial.

Overhead of our dynamic linker. We evaluate the overhead of our dynamic linker when unused. Indeed, programs that make no use of shared libraries should not be slowed down by this feature. Conveniently, PARSEC and Phoenix do not make extensive use of shared libraries, except libc for thread management. Figure 12 shows no difference between Risotto (with the linker) and `tcg-ver` (without the linker). Thus, our linker has no impact on performance if no host function is linked.

7.4 Fast Compare-and-Swap

To evaluate our CAS translation, we implement a micro-benchmark that stresses this component in a multi-threaded setup. We vary the number of threads and variables accessed by CAS instructions to show various levels of contention. Figure 15 shows the throughput of Qemu, Risotto, and a native Arm binary. Note that Qemu's helper functions also use the `casal` instruction.

We observe that Risotto outperforms Qemu only when there is no contention (#threads = #variables) by up to 53% (14.5% on average). However, under contention, they perform similarly. Indeed, the `casal` instruction then dominates the execution time, reducing the relative impact of the additional jumps performed by Qemu.

8 RELATED WORK

Concurrency and memory models. Mappings of concurrency primitives from programming languages to different architecture have been studied widely in the literature [17, 18, 39, 41, 64, 65, 71] where correctness is established based on formal semantics. The correctness of program transformations under relaxed memory models is also well explored [25–27, 39, 40, 45, 73, 74, 84]. Similar to these approaches, we use formal semantics of the architectures to define correct and precise mapping schemes as well as correct translations on TCG IR.

Prior works have formalized informal concurrency specifications such as C/C++ [18, 21], LLVM IR [26], Power and ARMv7 [10, 53, 72]. The earlier formalization of ARMv8 by Pulte *et al.* [68] is updated by Alglave *et al.* [6] with the semantics of `casal` accesses. To our knowledge, we are the first to formalize the TCG IR concurrency model to obtain formally verified cross-architecture translations.

There are several results on identifying the differences between weak memory models [2, 3, 5, 9, 33, 34, 52, 80, 87]. To address these differences, a number of optimized fence placement approaches have been proposed [31, 43, 56, 76, 81, 85]. However, optimal fence placement is an undecidable problem in the general case [14].

Recently, VSync [59] proposed using model checking to identify efficient fence insertion in Arm and RISC-V programs. Others have developed analyses to check if a program is SC-robust/stable against weaker models, inserting fences where necessary [1, 7, 23, 44, 46–48, 76]. Tao *et al.* [83] implement a KVM-based hypervisor that satisfies *weak data race free conditions* on an SC model which also holds on the Arm model. However, using model checking to insert fences is computationally expensive, and rarely scales beyond small programs.

Binary translation. While many Qemu-based DBTs support multi-threaded programs, most fail to address mismatches among memory consistency models [29, 35, 86]. Similarly, modern static binary translators target the LLVM IR, allowing for better whole program optimization [15, 20, 24, 77, 89]. However, they do not support concurrency either.

Apple's Rosetta 2 [11] is an emulator developed for their x86 to Arm transition. It uses both static and dynamic binary translation. It handles the memory model mismatch by implementing both x86

and Arm models in hardware [38]. Microsoft also enables emulation of x86 binaries on Arm machines through the WOW64 layer [54]. They use a caching system that optimizes the generated code after a first execution. Unfortunately, both Microsoft's and Apple's solutions rely on their control over the hardware and software ecosystems, and are closed source, with scarcely available technical details.

ArMOR. ArMOR [51] proposes a specification format that defines the ordering of memory accesses in architectures, and other properties such as *multicopy-atomicity* (MCA), allowing it to identify the required fences during a program execution. However, it has several limitations:

- **No DBT** — ArMOR generates DFSMs to insert fences, which they applied inside the Pin [50] instruction instrumentation tool. Pin, however, is not a DBT system. Pico [28] leverages ArMOR to obtain mapping rules for load and store accesses. As ArMOR cannot handle RMWs, Pico defines their own mapping rules for RMWs, *without any formal guarantees* of correctness. Additionally, Pico translates PowerPC to x86, which differs from translating from x86 to Armv8 through TCG.
- **No RMWs** — ArMOR considers RMWs a straightforward extension, *as long as ordering behavior is correctly specified*. Through our formal proofs, we discover that RMWs may display intricate behavior, which differs subtly between architectures. Moreover, Arm's `LX/SX` RMWs suffer from spurious failures unlike x86 RMWs, which goes beyond ordering rules. Hence, we believe that ArMOR cannot *easily* handle RMWs without major extensions.
- **Dependency tracking** — We carefully analyze dependencies in Arm and discover their behavior to be quite intricate. We thus elect to *eliminate them with our mappings*. If dependencies were included in ArMOR, we foresee some challenges: *(1)* it is *computationally expensive* to track dependencies for an *arbitrary number* of memory location, and *(2)* dependency rules may be exceedingly complex. For instance, Arm's dob can order an event a with another *write* event b, if there is another instruction in-between that is address-dependent on a.
- **QEMU** — Qemu translates programs at basic block granularity, across which no information propagates. In ArMOR, this corresponds to a *stream interruption*, which may cause inserting unnecessarily strong fences. Additionally, Qemu performs intermediate optimizations on concurrency primitives, for which it is not clear how it interacts with ArMOR's approach.

Host shared libraries. Tan *et al.* [82] use Qemu's helper functions to support calls to native shared library functions, adding a level of indirection, and requiring hard-coding the helper functions. Qemu has to be recompiled when adding support for a function. `box86/64` [66, 67] implement native shared libraries in their instruction set simulator with "wrapped libraries". This approach also requires hard-coding a glue layer that supports native shared library invocation.

Microsoft's Windows-on-Arm supports this feature by changing the ABI of Windows, easing the translation from x86 to Arm [55]. Rosetta 2 also uses a common ABI for x86 and Arm and performs lazy binding of shared library functions [58].

9 CONCLUSION

We present an end-to-end approach to provide correct and efficient execution of legacy x86 software on the weak memory Arm architecture. To achieve this, we formalize Qemu's TCG IR memory model, and use it to propose formally verified mapping schemes. We leverage these schemes in Risotto, a Qemu-based DBT system that optimizes fence placement while ensuring correctness. Risotto further optimizes performance by cross-architecture dynamic linking of native shared libraries and a fast and correct CAS translation. We evaluate Risotto using multi-threaded benchmark suites and real-world applications, and show that Risotto improves the emulation performance, while ensuring correctness.

Open source contributions. We contacted the Arm-Cats authors to propose a strengthening of the Arm model that was accepted [37]. We also submitted our new mapping schemes to the Qemu mailing list. The patch is currently under review with a positive feedback.

Artifacts. The proofs will be publicly available. Risotto is publicly available at: https://github.com/rgouicem/qemu/tree/risotto.

ACKNOWLEDGMENTS

REFERENCES

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Magnus Lång, and Tuan Phong Ngo. Precise and sound automatic fence insertion procedure under PSO. In *NETYS*, volume 9466 of *Lecture Notes in Computer Science*, pages 32–47, 2015.

[2] S. V. Adve and J. K. Aggarwal. A unified formalization of four shared-memory models. *IEEE Trans. Parallel Distrib. Syst.*, 4(6):613–624, June 1993.

[3] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[4] Agda Development Team. *Agda 2.6.2 documentation*, 2021.

[5] Jade Alglave. A formal hierarchy of weak memory models. *Form. Methods Syst. Des.*, 41(2):178–210, 2012.

[6] 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), 2021.

[7] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence: A static analysis approach to automatic fence insertion. *ACM Trans. Program. Lang. Syst.*, 39(2):6:1–6:38, 2017.

[8] Jade Alglave and Luc Maranget. herd7 consistency model simulator. http://diy.inria.fr/www/.

[9] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *CAV'10*, page 258–272, 2010.

[10] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: modelling, simulation, testing, and data-mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

[11] Apple. WWDC2020 Keynote (at 1:39:25). https://developer.apple.com/videos/play/wwdc2020/101/, June 2020.

[12] ARM. ARM Cortex-A72 MPCore Processor Technical Reference Manual – Memory access sequence. https://developer.arm.com/documentation/100095/0003/Memory-Management-Unit/Memory-access-sequence.

[13] ARM. Arm cortex-a series programmer's guide for armv8-a. https://developer.arm.com/documentation/den0024/a/, 2015.

[14] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. What's decidable about weak memory models? In *ESOP'12*, pages 26–46, 2012.

[15] avast. A retargetable machine-code decompiler based on llvm. https://github.com/avast/retdec.

[16] Amazon AWS. Aws graviton processor. https://aws.amazon.com/ec2/graviton.

[17] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and compiling C/C++ concurrency: From C++11 to POWER. In *POPL'12*, pages 509–520. ACM, 2012.

[18] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL'11*, pages 55–66. ACM, 2011.

[19] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[20] Lifting Bits. Framework for lifting x86, amd64, and aarch64 program binaries to llvm bitcode. https://github.com/lifting-bits/mcsema.

[21] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *PLDI'08*, 2008.

[22] Jupyter book community. Jupyter homepage. https://jupyter.org.
[23] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against TSO. In *ESOP 2013*, pages 533–553, 2013.
[24] Ahmed Bougacha. Binary translator to llvm ir. https://github.com/repzret/dagger.
[25] Soham Chakraborty and Viktor Vafeiadis. Validating optimizations of concurrent C/C++ programs. In *CGO'16*, pages 216–226. ACM, 2016.
[26] Soham Chakraborty and Viktor Vafeiadis. Formalizing the concurrency semantics of an llvm fragment. In *CGO '17*, pages 100–110. IEEE, 2017.
[27] Soham Chakraborty and Viktor Vafeiadis. Grounding thin-air reads with event structures. *Proc. ACM Program. Lang.*, 3(POPL), 2019.
[28] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. Cross-isa machine emulation for multicores. In *CGO'2017*, page 210–220. IEEE Press, 2017.
[29] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. PQEMU: A parallel system emulator based on QEMU. In *ICPADS'11*, pages 276–283, 2011.
[30] Docker. Docker homepage. https://www.docker.com.
[31] Reinoud Elhorst. Lowering C11 atomics for ARM in LLVM. In *European LLVM Conference*, 2014.
[32] Andrei Frumusanu. Amazon's Arm-based Graviton2 Against AMD and Intel: Comparing Cloud Compute – Anandtech. https://www.anandtech.com/show/15578/cloud-clash-amazon-graviton2-arm-against-intel-and-amd, 2020.
[33] L. Higham, J. Kawash, and Nathaly Verwaal. Defining and comparing memory consistency models. In *PDCS'97*, 1997.
[34] Lisa Higham, Lillanne Jackson, and Jalal Kawash. Specifying memory consistency of write buffer multiprocessors. *ACM Trans. Comput. Syst.*, 2007.
[35] 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 *CGO'12*, page 104–113, 2012.
[36] RISC-V International. Risc-v. https://riscv.org/.
[37] jalglave. [aarch64 cat] atomics strengthening #322. https://github.com/herd/herdtools7/pull/322.
[38] Saagar Jha. TSOEnabler – Kernel extension that enables TSO for Apple silicon processes. https://github.com/saagarjha/TSOEnabler.
[39] Jeehoon Kang, Hur, Chung-Kil, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *POPL'17*. ACM, 2017.
[40] Ori Lahav and Viktor Vafeiadis. Explaining relaxed memory models with program transformations. In *FM'16*, pages 479–495, 2016.
[41] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *PLDI 2017*, pages 618–632, 2017. Technical Appendix Available at https://plv.mpi-sws.org/scfix/full.pdf.
[42] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.
[43] J. Lee and D. Padua. Hiding relaxed memory consistency with a compiler. *IEEE Trans. Computers*, 50:824–833, 2001.
[44] J. Lee and D. A. Padua. Hiding relaxed memory consistency with a compiler. *IEEE Transactions on Computers*, 50(8):824–833, 2001.
[45] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. Promising 2.0: Global optimizations in relaxed memory concurrency. In *PLDI 2020*, page 362–376, 2020.
[46] Alexander Linden and Pierre Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In *SPIN'11*, pages 144–160, 2011.
[47] Alexander Linden and Pierre Wolper. A verification-based approach to memory fence insertion in pso memory systems. In *TACAS*, 2013.
[48] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *PLDI '12*, pages 429–440, 2012.
[49] Nian Liu, Binyu Zang, and Haibo Chen. No barrier in the road: A comprehensive study and optimization of arm barriers. In *PPOPP'20*, page 348–361, 2020.
[50] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI 2005*, page 190–200, 2005.
[51] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. Armor: Defending against memory consistency model mismatches in heterogeneous architectures. In *ISCA'15*, page 388–400, 2015.
[52] Sela Mador-Haim, Rajeev Alur, and Milo M K. Martin. Generating litmus tests for contrasting memory consistency models. In *CAV'10*, page 273–287, 2010.
[53] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the arm and power relaxed memory models, 2012. Draft.
[54] Microsoft. How x86 emulation works on arm. https://docs.microsoft.com/en-us/windows/uwp/porting/apps-on-arm-x86-emulation.
[55] Microsoft. Using arm64ec to build apps for windows 11 on arm devices. https://docs.microsoft.com/en-us/windows/uwp/porting/arm64ec.
[56] Robin Morisset and Francesco Zappa Nardelli. Partially redundant fence elimination for x86, arm, and power processors. In *CC'17*, pages 1–10, 2017.
[57] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *PLDI'13*, pages 187–196. ACM, 2013.
[58] Koh M. Nakagawa. Reverse-engineering rosetta 2 part1: Analyzing aot files and the rosetta 2 runtime. https://ffri.github.io/ProjectChampollion/part1/#x86_64-address-resolution-and-lazy-binding, 2021.
[59] Jonas Oberhauser, R. Chehab, Diogo Behrens, Ming Fu, A. 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. *ASPLOS'21*, 2021.
[60] Maintainers of nix. Nixos homepage. https://nixos.org/download.html.
[61] OpenSSL. Openssl – cryptography and ssl/tls toolkit. https://www.openssl.org/.
[62] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP*, pages 478–503, 2010.
[63] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *TPHOLs*, pages 391–407, 2009.
[64] Gustavo Petri, Jan Vitek, and Suresh Jagannathan. Cooking the books: Formalizing JMM implementation recipes. In *ECOOP 2015*, volume 37 of *LIPIcs*, pages 445–469, 2015.
[65] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.*, 3(POPL), 2019.
[66] ptitSeb. box64. https://github.com/ptitSeb/box64, 2021.
[67] ptitSeb. box86. https://github.com/ptitSeb/box86, 2021.
[68] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *PACMPL*, 2(POPL):19:1–19:29, 2018.
[69] QEMU. the fast! processor emulator. https://www.qemu.org/.
[70] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R. Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA*, pages 13–24. IEEE Computer Society, 2007.
[71] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. Synchronising C/C++ and POWER. In *PLDI'12*, pages 311–322. ACM, 2012.
[72] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *PLDI '11*, pages 175–186, 2011.
[73] Jaroslav Sevcík. Safe optimisations for shared-memory concurrent programs. In *PLDI 2011*, pages 306–316, 2011.
[74] Jaroslav Sevcík and David Aspinall. On validity of program transformations in the java memory model. In *ECOOP 2008*, pages 27–51, 2008.
[75] Agam Shah. We're closing the gap with arm and x86, claims sifive: New risc-v cpu core for pcs, servers, mobile incoming – the register. https://www.theregister.com/2021/10/21/sifive_riscv_cpu/, 10 2021.
[76] Dennis E. Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.
[77] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. Llbt: An llvm-based static binary translator. In *CASES 2012*, page 51–60, 2012.
[78] Tom Spink, Harry Wagstaff, and Björn Franke. A retargetable system-level DBT hypervisor. In *USENIX Annual Technical Conference*, pages 505–520. USENIX Association, 2019.
[79] SQLite. Database speed comparison. https://www.sqlite.org/speed.html.
[80] Robert C. Steinke and Gary J. Nutt. A unified theory of shared memory consistency. *J. ACM*, 51(5):800–849, 2004.
[81] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler techniques for high performance sequentially consistent java programs. In *PPOPP'05*, page 2–13, 2005.
[82] Jie Tan, Jian-min Pang, and Shuai-bing Lu. Using local library function in binary translation. In *Current Trends in Computer Science and Mechanical Automation Vol. 1*, pages 123–132. De Gruyter Open Poland, 2018.
[83] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. Formal verification of a multiprocessor hypervisor on arm relaxed memory hardware. In *SOSP*, pages 866–881. ACM, 2021.
[84] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *POPL'15*, pages 209–220. ACM, 2015.
[85] Viktor Vafeiadis and Francesco Zappa Nardelli. Verifying fence elimination optimisations. In *SAS'11*, volume 6887 of *LNCS*, pages 146–162. Springer, 2011.
[86] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. COREMU: a scalable and portable parallel full-system emulator. In Calin Cascaval and Pen-Chung Yew, editors, *PPOPP'11*, pages 213–222, 2011.
[87] John Wickerson, Mark Batty, Tyler Sorensen, and George A. Constantinides. Automatically comparing memory consistency models. In *POPL'17*, pages 190–204. ACM, 2017.
[88] QEMU wiki. Features/tcg-multithread. https://wiki.qemu.org/Features/tcg-multithread.
[89] S. Bharadwaj Yadavalli and Aaron Smith. Raising binaries to llvm ir with mctoll (wip paper). In *LCTES 2019*, page 213–218, 2019.

A ARTIFACT APPENDIX

The artifact is available in the following GitHub repository: https://github.com/binary-translation/risotto-artifact-asplos23. It contains full documentation on how to reproduce the results of this paper. This appendix only contains the necessary subset of information from the documentation. If you run into problems or want a more fine-grained control over the reproduction of the results, please refer to the documentation in the repository. We advise using the documentation in the repository if you want to coy commands more easily.

A.1 Requirements

Hardware. To run these experiments, you need a machine with an Arm processor implementing at least the ARMv8.2 revision. We also recommend using an x86_64 machine to compile the benchmarks' x86 binaries used in the evaluation. You can also cross-compile them on the Arm machine, but we do not provide instructions for this.

Software. Our evaluation requires a Linux-based system, and uses Nix [60] to manage the dependencies. We explain later on how Nix is used. If you want to generate the plots on your local machine, you will require the following Python packages: notebook, pandas, seaborn and matplotlib.

To reproduce the proofs, we provide a Docker [30] image, which means you need to install Docker on your system.

Benchmarks. We use the PARSEC 3.0 [19] and Phoenix [70] benchmark suites, as well as openssl [61], sqlite [79] and some micro-benchmarks of our design. We provide scripts to either download pre-built binaries or build all the benchmarks from source.

A.2 Quick Setup

Installing Nix. You first need to install Nix on the Arm machine you will use for evaluation. You can do this by following the instructions on their webpage, which, at the time of this writing, amount to running the following command:

```
sh <(curl -L https://nixos.org/nix/install) -daemon
```

This needs to be done only once.

Setting up the environment. Before doing anything, you need to setup the environment by running the following command from the root directory of the repository:

```
source sourceme
```

You will need to do this every time you want to run experiments or build software from this repository from a different shell.

Building the binaries. You can now build all the binaries used in this paper with the following command from the root of the repository:

```
./scripts/build.sh
```

This will start the compilation of all four Qemu variants used in this paper:

- `master-6.1.0`: vanilla Qemu 6.1.0
- `no-fences`: vanilla Qemu 6.1.0 that doesn't enforce any memory model
- `tcg-tso`: vanilla Qemu 6.1.0 with our memory mappings
- `risotto`: Risotto (Qemu 6.1.0 with our memory mappings, dynamic host linker and CAS translation)

It will also download pre-built binaries of all benchmarks used in the paper. All binaries will be available in the `build/` directory. All configurations available in the repository assume that you use these binaries.

A.3 Reproducing the Experimental Results

With everything setup, you can now reproduce the evaluation of the paper.

Running the benchmarks. You can run the benchmarks with the following command:

```
./scripts/run_benchmarks.sh
```

This will execute all the benchmarks from the paper's evaluation. The raw results are available in the `results/` directory as CSV files.

Plotting the results. After the benchmarks finish their execution, you can plot the figures from the paper, namely Figures 12–15, by executing the following command:

```
./scripts/plot.sh
```

This will produce the figures as PDF files available in the `results/` directory.

A.4 Verifying the Proofs

We provide the proofs in the repository's `proofs` directory (with separate `README.md`). Additionally, we provide pre-built Docker images. To check the proofs, run:

```
docker run -it --rm \
sourcedennis/risotto-proofs:latest \
agda src/Main.agda --safe
```

You can generate HTML-rendered Agda with the following command (in local directory `html/`):

```
docker run -it --rm \
-v "$PWD/html:/proofs/html" \
sourcedennis/risotto-proofs:latest \
agda --html --html-dir=html src/Main.agda
```

You can open `html/Main.html` in any web browser.

A.5 Detailed Instructions

You can find more information on how to reproduce individual results in the `README.md` file from the repository. We now provide some of this information for interested readers.

Building the benchmarks from source. You can build each benchmark individually from source with the scripts available in the `scripts/` directory. Note that you need two versions of each benchmark:

- x86, which will be executed through the binary translators, *i.e.,* the Qemu variants
- aarch64, which will be executed natively on the machine as a comparison

We provide instructions to build these benchmarks on their respective architectures. You can also do this on a single architecture using cross-compilation, with some modifications to our scripts. We do not cover this.

On both the x86 and Arm machines, run the following commands to build the benchmarks:

```
source sourceme
nix-shell -run scripts/build_benchmarks.sh \
default.nix
```

You can check the `scripts/build_benchmarks.sh` scripts and the scripts it calls for more details on how the benchmarks are built.

For the PARSEC and Phoenix benchmarks, you also need to download the input datasets available on their respective website and repository.

Note that you may need to change some configuration files to properly match the paths of your newly built benchmarks in the `config` directory.

Running the benchmarks individually. You can check the commands in the `scripts/run_benchmarks.sh` script to see how each benchmark is executed individually.

Plotting the results. In addition to generating the plots as PDFs as explained previously, you can also generate them through interactive Jupyter notebooks [22] on your local machine. You just need to run the following command in the root directory of the repository after executing the benchmarks and downloading the resulting CSV files on your local machine in the `results` directory:

```
jupyter notebook
```

This will open a browser window, where you can access the Jupyter notebooks in the `plots/` directory. After opening a notebook, click the *Run* button to run every cell (or *Run All* in the *Cell* menu).