Arancini: A Hybrid Binary Translator for Weak Memory Model Architectures

Sebastian Reimers*
TU Munich
Munich, Germany
sebastian.reimers@tum.de

Dennis Sprokholt*†
TU Munich
Munich, Germany
dennis.sprokholt@tum.de

Martin Fink
TU Munich
Munich, Germany
martin.fink@cit.tum.de

Theofilos Augoustis
TU Munich
Munich, Germany
theofilos.augoustis@tum.de

Simon Kammermeier
TU Munich
Munich, Germany
simon.kammermeier@tum.de

Rodrigo C. O. Rocha‡
Huawei Research
Edinburgh, UK
rodrigo.rocha@huawei.com

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

Redha Gouicem
RWTH Aachen University
Aachen, Germany
gouicem@cs.rwth-aachen.de

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

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

Abstract

Binary translation is a powerful approach to support cross-architecture emulation of unmodified binaries in increasingly heterogeneous computing environments. However, binary translation systems face *correctness* issues, due to the *strong-on-weak memory model mismatch* (e.g., from x86-64 to Arm/RISC-V) for concurrent programs. Besides, the current landscape of binary translation systems is fundamentally limited in terms of *completeness* for *static* systems and *performance* for *dynamic* ones.

To address these limitations, we propose Arancini, a hybrid binary translator system designed and implemented from the ground up that strives for correct, complete, and efficient emulation for weak memory model architectures. Our system makes three foundational contributions to achieve these design goals: ArancinIR, a unified intermediate representation for static and dynamic binary translators; a formalization of ArancinIR's memory model and formally verified mapping schemes from x86-64 to Arm and RISC-V, to ensure strong-on-weak correctness; and Arancini, a complete and performant hybrid binary translator, implementing the verified mapping schemes for correctness.

We evaluate Arancini using a multi-threaded benchmark suite with two backends (Arm and RISC-V), and show that Arancini can be up to 5× faster than QEMU-based translators while ensuring correctness and completeness.

To our knowledge, Arancini is the first hybrid binary translator whose implementation is guided by formal proofs, to ensure correct execution of strong memory guests on weak memory hosts. It is also the first translator to address mixed-sized accesses for Arm targets.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Formal methods**; • **Theory of computation** → *Concurrency*.

Keywords: Binary translation; Memory models; Formal verification

ACM Reference Format:
Sebastian Reimers, Dennis Sprokholt, Martin Fink, Theofilos Augoustis, Simon Kammermeier, Rodrigo C. O. Rocha, Tom Spink, Redha Gouicem, Soham Chakraborty, and Pramod Bhatotia. 2026. Arancini: A Hybrid Binary Translator for Weak Memory Model Architectures. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3779212.3790127

1 Introduction

In the last decade, the landscape of heterogeneous computing hardware has greatly evolved, with a shift from the dominating x86-64 architecture to new Instruction Set Architectures (ISAs) such as Arm and RISC-V [9, 14]. This change stems from the industry's demand for better performance, energy

*Both authors contributed equally to the paper.
†Work done while the author was at TU Delft.
‡Work done while the author was at the University of Edinburgh.
§Jointly led the project.

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790127

efficiency, and licensing [34, 80]. For instance, Arm CPUs are now being deployed widely, including Google Axion [88], Apple M3 [42], AWS Graviton [17], Microsoft Cobalt [47], and Huawei Kunpeng [40]. For this transition to happen, applications written for one architecture must be ported to others.

Binary translators provide an effective solution in porting applications across these diverse architectures, particularly in the absence of source code. Given an application binary, such translators statically generate a binary for the another ISA ahead of time, or dynamically execute the *guest* binary on the new *host* ISA by emulating the guest architecture. Despite significant advancements, state-of-the-art binary translators struggle to *(i)* tackle the complexities of modern concurrency primitives, *(ii)* statically translate certain features without run time information, and *(iii)* achieve good performance.

Static Binary Translators (SBTs) [27, 35, 38, 76, 96] offer performance, but lack completeness as they cannot translate all features ahead-of-time [11], due to the undecidable nature of static disassembly [74]. In contrast, Dynamic Binary Translators (DBTs) [37, 72, 85] are complete, but come with a significant performance penalty due to both runtime compilation latency and reduced optimization opportunities. Hybrid systems combine both approaches by dynamically translating only the instructions that cannot be translated statically. A hybrid translator could theoretically achieve the completeness of a DBT with near-SBT performance.

In addition, any translator faces challenges when translating concurrent programs between ISAs. Architectures such as x86-64, Armv8, and RISC-V follow different architectural rules, such as their memory hierarchy and out-of-order execution, and consequently, their memory consistency models differ. The mismatch between the consistency models may cause incorrect translation of programs, particularly when translating from a stronger (e.g., x86-64) to a weaker architecture (e.g., Armv8 or RISC-V). QEMU [72], a state-of-the-art binary translator, was shown to translate concurrent programs *incorrectly* [37], and thus disables concurrency altogether, severely affecting performance.

Designing a correct hybrid translation system poses several challenges: *(i)* Unifying the Intermediate Representations (IRs) required for both static and dynamic translation. Static translation benefits from a high-level representation for whole-program optimization, while dynamic translation benefits from low-level details, making their integration non-trivial. A common base-IR can bridge the gap between the two abstraction levels. *(ii)* Ensuring architectural correctness. Any translator must ensure that the memory semantics of the guest architecture are correctly enforced on the host architecture, particularly when translating from a stronger to a weaker architecture. A formally verified mapping scheme is necessary to ensure correctness. *(iii)* Efficient inter-translation switching. Enabling transitions between static and dynamic translations, without losing performance, requires an efficient mapping between original guest code and the translations. A new hybrid binary format is required to provide the necessary information at runtime.

To address these challenges, we propose Arancini, an end-to-end Hybrid Binary Translator (HBT), designed and implemented from the ground up, for correct and efficient *strong-on-weak* binary translation with mixed-size accesses.

Specifically, Arancini employs ArancinIR, our unified low-level IR to handle challenge *(i)*. ArancinIR is able to retain high-level information for static translation, while also being able to be transformed quickly into the target IR for dynamic translation. We address challenge *(ii)* by formally defining a weak memory model for ArancinIR, which we use to derive formally proven correct memory access mappings, from the strong guest memory model to the weak host models. In particular, our mappings correctly translate mixed-size accesses [5, 33] from x86-64 to ArancinIR to Arm. We implement those correct mappings in Arancini for both Arm and RISC-V architectures. Lastly, we address challenge *(iii)* by designing a new hybrid binary format which contains the original guest code, its static translation in the form of native host code, and metadata to efficiently switch between static and dynamic translation at runtime.

Challenge *(ii)* is further complicated by mixed-size extensions to weak memory models, introducing subtle complexities in program behavior, and thus translation correctness [60, 78]. Prior mapping schemes from x86-64 to Armv8 accompanied by formal proofs [37, 76], did not consider mixed-size concurrency models. Our proven mapping scheme from x86-64 to Armv8 addresses this non-trivial challenge, which required extensive proof mechanization efforts. Additionally, translating to RISC-V with mechanized proofs is a new contribution over earlier works.

We use those new mappings in Arancini with an x86-64 frontend and two backends: Arm and RISC-V. We evaluate Arancini on the concurrent applications from the Phoenix benchmark suite [73] and micro-benchmarks on both hosts across three dimensions: correctness, completeness, and performance. Arancini's *performance* closely matches that of Lasagne [76], a state-of-the-art SBT system, while also achieving *completeness* approaching that of a state-of-the-art DBT like Risotto [37]. Although our design enables Arancini to be complete like any DBT, we have not currently implemented all features of Risotto and QEMU. Unlike QEMU, our verified translation rules *correctly* enforce the x86-64 ordering on Armv8 and RISC-V, even for subtle mixed-size accesses (on Armv8), while being up to 5× faster than Risotto.

Contributions. We make three foundational contributions:

- We propose the **ArancinIR, a unified low-level IR**, that bridges the gap between static and dynamic binary translation (Section 4), for strong-on-weak emulation. ArancinIR is designed to closely represent binary-level information for dynamic translation without sacrificing the ability to use the high-level representation for static translation.

- We present a **formal memory model of ArancinIR with correctness proofs of memory mappings** from x86-64 to ArancinIR, and from ArancinIR to both Armv8 and RISC-V, mechanized in the Agda proof assistant [4]. Our formal model includes semantics for mixed-size accesses, which are thus considered in the verified mapping from x86-64 to Armv8 through ArancinIR. By implementing these translation rules in Arancini, we ensure correct translation of strong weak memory behaviors on weak hosts (Section 5).
- We offer a **design and implementation of Arancini**, and how it solves the performance-completeness trade-off, whilst soundly emulating the strong guest memory model on weaker hosts. (Sections 3 and 6 to 8). We evaluate Arancini on two backends, Arm and RISC-V, against state-of-the-art SBT and DBT systems, and across three dimensions: performance, completeness, and correctness, showing that our hybrid approach gets the best of both static and dynamic worlds (Section 9).

2 Background and Motivation

Binary translators traditionally classify as either *static*, translating binary programs *ahead of time*, or *dynamic*, translating them *at runtime*. Each approach offers unique strengths, which we incorporate in our solution.

2.1 Static Binary Translation

A Static Binary Translator (SBT) translates programs before execution, and has ample time for optimizations before generating the final program. To do so, state-of-the-art SBTs often *lift* binary programs to a higher-level IR [16, 22, 24, 76, 82, 94, 96], such as LLVM IR [50, 57]. From this high-level abstraction, common compiler infrastructures such as LLVM can be used to generate target code. The primary challenge in static translation is lifting the source binary to an IR.

Completeness of binary translation. SBTs [22, 76] are *necessarily* incomplete [11, 74], meaning they fail to translate some programs. When a binary program is compiled from a high-level language, much information is lost. In particular, control flow structures and jump targets are removed. Reconstructing that information from a binary is difficult in practice and *impossible* in general [74]. Additionally, lifting binary programs brings issues that do not exist in the source language, such as distinguishing between pointers and constants, or between data and instruction bytes [11]. This is a *fundamental limitation* of SBTs, not only of any specific implementation. Consequently, state-of-the-art SBTs [22, 46, 76] based on LLVM fail to lift many binary programs.

2.2 Dynamic Binary Translation

In contrast, a DBT can be complete, since it only translates program fragments at runtime within their local execution contexts. For instance, the state-of-the-art DBT QEMU [72] translates a binary program to its Tiny Code Generator (TCG) IR [71]. State-of-the-art DBTs [29, 37, 41, 72, 85] distinguish themselves by emulating different fragments of the host machine, such as `RMW` instructions and floating-point operations.

Performance of binary translation. Unfortunately, while local translation ensures completeness, it lacks a global view of the program, preventing many optimizations. To showcase the performance gap between DBT and SBT, we compare two state-of-the-art cross-ISA binary translators, Lasagne [76] and Risotto [37], against native host binaries for the Phoenix [73] benchmark suite, which both translators use in their evaluation. Although Risotto is 5–10× slower than native, Lasagne can generate binaries that are only 2–3× slower. It performs well for two reasons: *(i)* it relies on the LLVM compiler and its function-level optimizations, and *(ii)* it doesn't need to fully emulate the x86-64 CPU state (unlike DBTs).

This performance overhead makes DBTs unappealing, but its completeness is more valuable in practice. In the current landscape of binary translation, one has to choose between *completeness* and *performance*.

Differences in IRs. Static and dynamic translators operate at different levels of abstraction, and thus their IRs embody different design decisions. LLVM IR was designed to compile high-level languages, where control flow is known. It provides high-level constructs, such as functions, that enable large-scale optimizations. However, said aggressive optimizations are often time-consuming and memory-heavy, making it too slow for use in dynamic translation. TCG operates on smaller local fragments together with runtime information. This is necessitated by dynamic jump targets being unknown in the general case, in binary programs [11]. Additionally, TCG is designed to heavily interact with the emulated hardware state and to be lowered quickly to the target ISA.

2.3 Weak Memory Concurrency

Traditionally, concurrency is explained by Sequential Consistency (SC) [49], i.e., accesses on shared memory locations in each thread execute in order, and multiple threads interleave arbitrarily. However, multi-core architectures demonstrate additional behaviors that cannot be explained by interleaving semantics only, for instance, by executing instructions out-of-order. Those additional executions are known as weak memory behaviors.

Primitives. Different architectures preserve the order among different instructions by default. However, each provides primitives to *explicitly* order those that are unordered. For our purpose, the important differences concern fences and atomic-update instructions, as shown in Table 1.

x86-64 lacks WW (write-write) and RM (read-memory) fences, because its strong memory model orders these access pairs by default. As Arm and RISC-V have weaker models, they require fences to explicitly order those pairs. All architectures provide *full fences* that *explicitly* order all access pairs, which are x86-64's `MFENCE`, Arm's `DMBFF`, and RISC-V's Fmm. `RMW` (Read-Modify-Write) instructions atomically update (i.e., read *and* write) memory.

Mixed-size concurrency. Recently, memory models started supporting *mixed-size* accesses in units of multiple sizes (e.g., 1/2/4/8-byte units) [5, 33]. For example, `lockref` and read-copy-update (RCU) implementations in the Linux kernel, Arm v8.0 ticketed spinlock, and the implementation of FreeBSD/i386 PAE page table bit manipulation perform mixed-size accesses in concurrent programs. In addition, state-of-the-art architectures such as x86-64 and Arm provide instructions to perform mixed-size accesses to optimize program performance. These mixed-size accesses affect concurrency behaviors, as shown in the following example:

$$\begin{array}{l} Y_L = 0\text{x}01 \\ t = Y; \end{array} \Bigg\| \; Y = 0\text{x}202; \qquad \text{(Mixed)}$$

Y is a 16-bit variable shared by the two concurrently running threads. Y_L and Y_H denote the least and most significant byte, respectively. The outcome Y = 0x202, t = 0x201 is forbidden in both x86-64 and Arm.

Correctness of binary translation. Ignoring the differences in weak memory models across architectures can cause errors when translating between them. Correctly translating concurrent programs demands paying close attention to these formal memory models. However, unlike earlier approaches [37, 76], we must also consider mixed-size concurrency models to correctly translate mixed-size accesses. Finally, while those approaches translated only to Arm, we also target RISC-V, demanding separate formal examination.

2.4 Proposal: A Correct Hybrid Binary Translator

While formal reasoning over memory models grants *correctness*, and DBT grants *completeness*, we can take advantage of SBT as much as possible to gain *performance*. Hence, to address performance, completeness, and correctness, we propose an end-to-end hybrid binary translation system that: *(i)* statically generates performant binaries; *(ii)* provides dynamic capabilities to achieve completeness; and *(iii)* relies on formally verified translation rules for concurrent programs.

3 Overview

We present Arancini, an end-to-end hybrid binary translation system that can translate a binary compiled for one ISA into an equivalent binary that will run on another ISA, especially targeting weak memory model architectures.

Access type	x86-64	Arm	RISC-V
Load	RMOV	LDR	LD
Store	WMOV	STR	ST
Full fence	MFENCE	DMBFF	Fmm
WW fence		DMBST	Fww
RM fence		DMBLD	Frm
Atomic-update	RMW	CAS_{AL}	RMW_{SC}

Table 1. Concurrency primitives in x86-64, Arm, and RISC-V. WW = write-write, ordering write-write pairs. RM = read-memory, ordering read-read and read-write pairs.

3.1 System Overview

Figure 1 gives a high-level overview of the components required to support hybrid binary translation and consists of the following components.

Static components. *(marked in blue)* The static translator performs an initial translation of a guest binary *ahead of time*, producing a hybrid binary. The translator may run on any architecture to support cross-translation.

Runtime components. *(marked in orange)* The runtime components facilitate execution of the initial translation, including dynamic translation of guest code at runtime. The dynamic translation module shares frontend and ArancinIR with the static translation module, but employs a lightweight host backend for low-latency JIT code generation, in contrast to the heavyweight LLVM backend.

Intermediary components. *(marked in red)* Intermediary components ease the translation to different target architectures. Sharing an IR between static and dynamic translation is a key contribution of our hybrid translation system.

Formal components. *(marked in green)* The formal components specify the rules and guarantees for correct translations between ISAs, specifically memory instructions to preserve correct memory ordering behavior. This is especially important for strong-on-weak ISA translations.

3.2 System Workflow

Figure 1 shows how a guest binary is translated into a host binary. In the *static phase*, the *guest frontend* ① scans the guest binary for instructions, via the symbol table, and lifts them to ArancinIR. This process uses the *formally verified mappings* to dictate how memory operations are lifted.

After generating ArancinIR, it is converted into LLVM IR, again using the *formally verified mappings* to generate appropriate LLVM memory operations. Following standard (and aggressive) LLVM optimizations, the static backend produces a *hybrid binary*, which contains both the translated code, the original guest machine code, and translation metadata.

In the *runtime* phase, execution of the hybrid binary begins (③) by running the statically translated code until an untranslated Program Counter (PC) value is encountered. The hybrid binary then calls the runtime support library to request a dynamic translation ④. Similar to the static phase, the same *guest frontend* is used to translate dynamically discovered guest instructions into ArancinIR. The ArancinIR is then translated directly to host machine instructions by the fast Arancini host backend, and stored in a code cache. After executing the dynamically translated code (⑤), control returns back to either the dynamic translator or statically translated code. Additionally, if appropriate shared libraries are natively available on the host machine, calls from the guest are mapped to their native version.

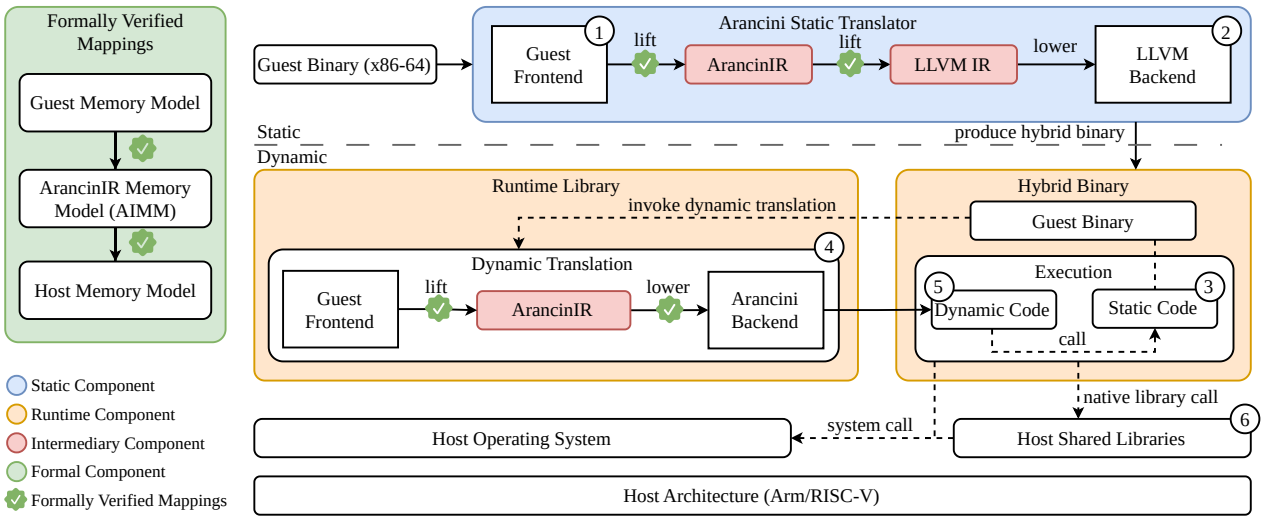

Figure 1. An overview of Arancini. First, the static translator ① lifts a guest binary to ArancinIR, then converts it to LLVM IR, producing a hybrid binary ②. At runtime, the statically translated code ③ may invoke the dynamic translation engine ④, which then dynamically generates code ⑤. Calls to native shared libraries are forwarded to the host's native shared libraries ⑥.

3.3 Design Challenges and Key Ideas

Combining static and dynamic translation presents several challenges, arising from their conflicting requirements. We identify three core challenges that must be overcome to effectively implement a hybrid translation scheme.

Challenge #1: Bridging the gap between static and dynamic binary translation. Existing IRs like TCG IR and LLVM IR are designed for a single translation scheme only.

While low-level IRs, like TCG IR are designed to efficiently encode guest hardware state, they are unsuitable for a static translator. Fundamentally, TCG operates on basic blocks, a granularity too small for large-scale, aggressive optimizations. LLVM IR is more generic and provides high-level constructs, such as functions, making it well-suited for static translation. However, the aggressive optimizations that it can perform are often time-consuming and memory-heavy, making host code generation much slower. Additionally, during hybrid translation, both passes must produce matching translations, to facilitate switching modes on demand.

We design ArancinIR to facilitate both high-performance static translation and low-overhead dynamic translation, by either being lifted to a high-level IR, or directly being lowered to host machine code. Using a single common frontend also reduces translation mismatches unique to hybrid translation.

Challenge #2: Ensuring architectural correctness. Translators should honor guest architecture semantics when generating host code. Particularly, memory ordering semantics, which may differ vastly between architectures. A naive and conservative approach would emit memory fences at each memory operation. While preserving guest semantics, this severely impacts run time performance. More optimal approaches insert memory fences without guaranteeing soundness [21]. In contrast, using a formal model and proof to guide memory fence placement ensures that a translation with an efficient mapping scheme is correct and adheres to the guest architecture's semantics [1].

Challenge #3: Efficient inter-translation switching. Hybrid binary translation requires the integration of individual static and dynamic translations. For this, a hybrid binary format is required, which is designed to execute a static translation natively on the host machine, but contains the original guest instructions too. Efficient mappings between translation and original code must be provided, in case dynamic control flow directs to an existing translation.

4 ArancinIR

In this section, we define ArancinIR, our new IR specifically designed for hybrid binary translation. It serves as the common IR for both the static and dynamic translation passes used by Arancini. No existing IR is suitable for this task, as they either operate on high-level concepts, thus cannot be lowered to the target ISAs efficiently, or are too low-level to enable heavy performance critical optimizations.

ArancinIR is a multi-level, node-based IR that is specifically designed for ISA translation, with first-class support for manipulating and interacting with the visible architectural state (memory and registers). ArancinIR does not model the control flow graph of the guest binary, but instead, it can be seen as the sequence of changes to the guest architectural state in the same order as they would happen if the guest binary was run natively.

Emulated guest state. A guest's state comprises memory and registers. For example, an x86-64 guest includes general

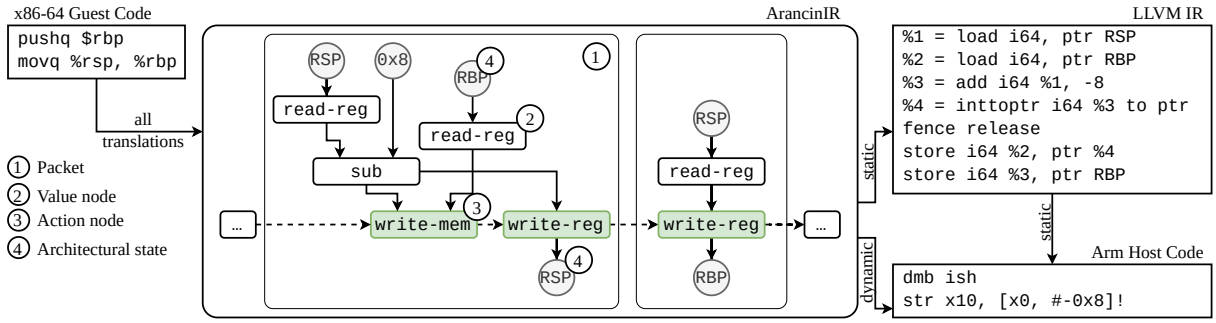

Figure 2. An example of how x86-64 guest code maps to ArancinIR, LLVM IR, and finally Arm host code. In the example, the two guest instructions are translated to two ArancinIR *packets* ①. These consist of one or more *value nodes* ② and *action nodes* ③, reading and modifying visible architectural guest state ④. Dashed lines represent program order, not node dependencies.

purpose registers, the flags register, the 80-bit extended floating point registers, and the status registers to manage them. ArancinIR is target agnostic, and is not involved in mapping the guest's state to the target architecture, this is performed by the specific translation backends.

Components. To describe the components of ArancinIR, we use Figure 2, showing the translation of x86-64 instructions to Arm via ArancinIR and LLVM IR.

Nodes in ArancinIR represent simple operations that either produce at least one value or cause an architectural side effect. Nodes that purely operate on values are called *value nodes* ②. Nodes that produce side effects (such as register or memory writes) are called *action nodes* ③ and mark points after which the observable guest state must be consistent.

Nodes are part of a Directed Acyclic Graph (DAG), with action nodes specifying the order of events. Each DAG represents a single guest instruction and is called a *packet* ①. It has the following restrictions: *(i)* no action node has outgoing edges, as they do not produce values; and *(ii)* each value node is part of a path ending in an action node, ensuring that all computations have visible side effects.

The edges in this graph are labeled with *ports*, which describe how output values should be interpreted. In most cases, the output of a node is the result of the operation it performs. However, for arithmetic operations, it may also be the value of a flag. Observing that a wide variety of ISAs support status flags, this powerful representation allows us to model them directly, without resorting to clumsy calculations.

Lastly, multiple packets can be combined into *chunks*. Chunks represent linear sequences of instructions, and could be entire functions when translating statically, or individual basic blocks, when translating dynamically. This model allows us to represent translations at different granularity, supporting both static and dynamic translation.

Optimizations. Optimizations applying to all translation modules can operate on ArancinIR. One example is the elimination of unused flags. When a flag is unused in a later packet, it is safe to remove the action nodes writing to the emulated flag registers. In case of disconnected value nodes (see *ii*) after an optimization, their entire sub-graphs can safely be removed. Notably, the loose ordering—imposed only by action nodes—allows for optimizations targeting instruction scheduling. Those optimizations are out of scope for the current implementation and are left as future work.

5 ArancinIR Memory Model and Mappings

When translating programs from x86-64 to Arm and RISC-V through our novel ArancinIR, we must not erroneously introduce new weak memory behaviors. Defining a correct translation is a major challenge, requiring a new formal memory model for ArancinIR while carefully inspecting the existing formal models of all involved architectures. To address those challenges, we propose a novel formal ArancinIR memory model (AIMM) and *mapping schemes* to correctly translate concurrent programs from x86-64 to Armv8/RISC-V by Arancini, which we prove correct.

We implement those mapping schemes in Arancini, where Figure 1 in Section 3 depicts their place in our architecture. In particular, we define verified mapping schemes from x86-64 to ArancinIR and for the lowering to Armv8 and RISC-V, which we show in Table 2. Their implementation in Arancini *formally guarantees* correct translation of weak memory behaviors from x86 to Armv8 and RISC-V. Specifically, our mapping restricts the weak behaviors on Armv8 and RISC-V to those observable on x86. In this section, we formally define AIMM and give a high-level explanation of our mapping proofs, which we mechanized in the Agda proof assistant [4].

5.1 Axiomatic Model for Concurrency

Weak memory models represent the behavior of programs with a set of executions. An execution consists of events, where each event results from the execution of a shared memory access or fence instructions; and various binary relations among those events [5, 8, 20]. We represent an execution with a graph where the nodes are events with edges capturing relations between them, as shown in Figure 3. We denote the set of shared memory read, write, and fence events

x86-64		AarcinIR		Arm	RISC-V
RMOV	→	ld;Frm	→	LDR;DMBLD	LD;Frm
WMOV	→	Fww;st	→	DMBST;STR	Fww;ST
RMW	→	RMW	→	RMW_{AL}	RMW_{SC}
MFENCE	→	Fmm	→	DMBFF	Fmm
		Frr/Frw	→	DMBLD	Frr/Frw
		Fwr/Fwm/ Fmr/Fmw	→	DMBFF	Fwr/Fwm/ Fmr/Fmw

Table 2. Verified mapping from x86-64 to Arm and RISC-V via AarcinIR. (Expanded in Appendix B.2.)

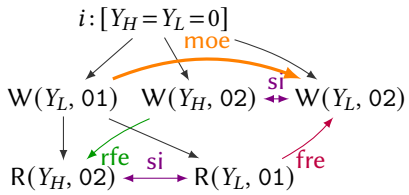

Figure 3. The execution of the Mixed program with outcome $Y = 0x202$, $t = 0x201$. It is forbidden in both x86 and Arm, but would be allowed without the recent si relation.

by R, W, and F, respectively. We use the conventional relations po (program-order), rf (reads-from), mo (modification order), fr (from-read), and rmw (read-modify-write).

We use existing models of x86, Armv8, and RISC-V [5, 7–9], whose formal definitions we include in Appendix B.3 for completeness. Note that the RISC-V model does not currently support mixed-size accesses.

Mixed-Size. Unique to mixed-size models, every load/store instruction may produce multiple events of the same type, which represents mixed-size accesses. For instance, a 32-bit RMOV instruction produces four R events, one for each byte read. Additionally, the primitive same-instruction (si) equivalence relation relates the events produced by the same instruction. Figure 3 shows an example execution with events and relations, including si.

Consistency axioms. Using the events and relations, a memory model defines consistency axioms. Intuitively, these axioms specify whether a given execution is observable in their corresponding model. For instance, the execution in Figure 3 is *not* observable in x86 and Arm, as their axioms disallow it. In contrast, when an execution satisfies all axioms, it is observable. For instance, in prior models without mixed-size accesses –and thus without the si relation– the execution in Figure 3 would be allowed.

Program behaviors. Given a program $\mathbb{P}$ and memory model M, the set of M-consistent execution defines the behavior of program $\mathbb{P}$. From all those consistent executions, we can obtain the final values stored in shared memory, which collectively represent the program's behaviors.

5.2 Concurrency Models: x86, Arm, and RISC-V

We briefly discuss the axiomatic models of x86 [5, 67], Armv8 [5, 6], and RISC-V [9]. All three architectures enforce (Coherence) and (Atomicity).

$$(\text{poloc} \cup \text{rf} \cup \text{mo} \cup \text{fr}) \text{ is acylic} \qquad \text{(Coherence)}$$

(Coherence) ensures that accesses to the same memory location are totally ordered. Intuitively, that implies a program with only one shared variable is sequentially consistent.

$$\text{rmw} \cap (\text{fre};\text{moe}) = \varnothing \qquad \text{(Atomicity)}$$

(Atomicity) ensures that there is no intermediate write on the same location between a pair of rmw-related events, as that would have failed the RMW instruction.

Global Axioms. These three architectures differ in the global orderings they enforce, which are shown in Figure 4. Notably, all three architectures include the same inter-thread communication 'obs $\triangleq$ rfe ∪ moe ∪ fre' in their global order (which we restructured for presentation from the full models included in Appendix B.3). However, the thread-local orderings they enforce differ, as x86's xlob, Arm's lob, and RISC-V's ppo differ. For instance, although x86 orders W–W pairs by default along po—denoted by [W];po;[W] in xlob—Arm and RISC-V do not, as their respective bob and ppo relations lack the same clause.

To capture mixed-size accesses in the models of x86 (in Figure 4a) and Arm (in Figure 4b), obs composes with the same-instruction si relation. Observe that RISC-V (in Figure 4c) does not include the si relation, meaning its model does not formally consider mixed-size accesses — we thus do not know the behavior of mixed-size accesses on RISC-V.

5.3 Concurrency Model: AIMM

We define the Arancini primitives with their generated AarcinIR events and relations in an execution.

Memory accesses. Arancini provides load (ld) and store (st) operations that respectively read from and write to shared memory locations. ld and st accesses generate R and W events. It also provides an atomic read-modify-write (RMW) operation, which orders with surrounding events, like x86's RMW, Arm's RMW_{AL}, and RISC-V's RMW_{SC}. To ensure it also orders upon failure, we define the RMW to generate a distinct R_{SC} event and, if successful, a subsequent W_{SC} event.

Fence accesses. Arancini provides different types of fences: Frr, Frw, Frm, Fwr, Fww, Fwm, Fmr, Fmw, and Fmm. These fences generate F_{RR}, F_{RW}, F_{RM}, F_{WR}, F_{WW}, F_{WM}, F_{MR}, F_{MW}, F_{MM} events, respectively. 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 if a and b are R events with an intermediate F_{RR} event, which follows the [R];po;[F_{RR}];po;[R] case of ord.

Relations. Next, to capture mixed-size accesses we compose rfe, moe, and fre—which together form obs—with the si relation, denoted by obs;si. For example, if $(w,r) \in$ rfe and

(obs;si ∪ xlob) is acyclic
where
obs ≜ rfe ∪ moe ∪ fre
xlob ≜ [R];po;[W] ∪ [W];po;[W]
∪ [R];po;[R] ∪ po;[F];po ∪...

(a) x86 [5, 67]

(obs;si ∪ lob ∪ ...) is acyclic
where
obs ≜ rfe ∪ moe ∪ fre
lob ≜ bob ∪ ...
bob ≜ po;[F];po ∪ [R];po;[F_{LD}];po;[W]
∪ [W];po;[F_{ST}];po;[W]
∪ [A∪Q];po ∪ po;[L] ∪ [L];po;[A]
∪ ...

(b) Arm [5, 6]

(obs ∪ ppo) is acyclic
where
obs ≜ rfe ∪ moe ∪ fre
ppo ≜ [Acq];po;[M] ∪ [M];po;[Rel]
∪ fence ∪ ...
fence ≜ [R∪W];po;[F_{MM}];po;[R∪W]
∪ [R];po;[F_{RM}];po;[R∪W]
∪ [W];po;[F_{WW}];po;[W] ∪ ...

(c) RISC-V [9]

Figure 4. Global axioms for each architecture we consider, refactored for presentation. While x86 and Armv8 support mixed-size accesses (with si), RISC-V does not. We give the full models in Appendix B.3

$(r,e) \in$ si then w and e are ordered by rfe;si $\subseteq$ obs;si, which can happen when reading 16 bits from a 8-bit write. Finally, we define *global-happens-before* (ghb) using ord with the obs;si relation to order events across different threads.

AIMM Axioms. Like x86, Arm, and RISC-V, AIMM also enforces (Coherence) and (Atomicity). Finally, AIMM includes our new *global order* (GOrd) axiom to disallow ghb cycles in an AIMM execution.

ghb is acyclic where (GOrd)

ghb ≜ (obs;si ∪ ord)$^+$ where

obs ≜ rfe ∪ moe ∪ fre

ord ≜ ([R];po;[F_{RR}];po;[R])∪...∪([W];po;[F_{WW}];po;[W])

ord captures the thread-local orders between instructions. AIMM does not order W−W and R−R pairs, like Arm and RISC-V. Also, similar to all three architectures, AIMM forbids the Mixed execution in Figure 3.

5.4 Mappings and Optimizations

Based on AIMM, we define the mapping schemes in Table 2. For each mapping we prove Theorem 1.

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* M_t*-consistent target execution of* $\mathbb{P}_t$ *there exists a* M_s*-consistent source execution of* $\mathbb{P}_s$ *with same behavior.*

Correct mapping schemes. We translate concurrency primitives from x86-64 to Arm/RISC-V in two steps: (1) x86-64 to AIMM and (2) AIMM to Arm/RISC-V. We formally prove Theorem 1 to ensure correctness of these mappings. These mappings are *minimal*: for each fence in the mapping, there exists a program where the fence is necessary and sufficient to preserve correctness, i.e., no weaker fence is sufficient and no stronger fence is necessary.

Table 2 shows all three mapping schemes: from *(i)* x86-64 to AIMM, *(ii)* from AIMM to Arm, and *(iii)* from AIMM to RISC-V. In particular, for the first, we insert fences in the resulting ArancinIR program with load and store accesses to enforce x86-64's stronger ordering. For all three mappings we prove Theorem 1. As only the models for x86, AIMM, and Arm support mixed-size accesses, only mappings between those have formal guarantees. The composed mapping from x86 to Arm thus also correctly translates mixed-size accesses.

These mapping schemes are intended to be *correct in isolation*, meaning they require no knowledge about the program surrounding the mapped memory instructions; for *any* surrounding program that is mapped with the same scheme,

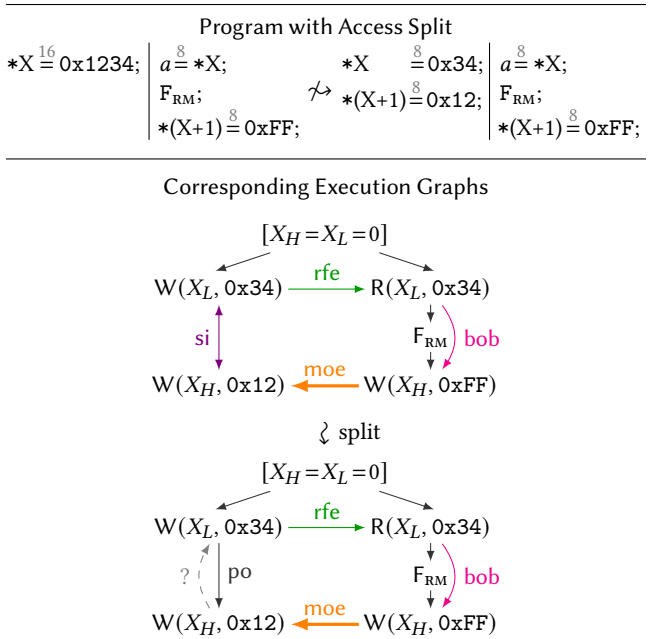

Figure 5. Example Arm program showing that splitting an access is incorrect. Splitting the 16-bit stores to X into two 8-bit stores to its high (X_H) and low (X_L) bytes is incorrect because it allows the new outcome $X = 0x1234$, $a = 0x34$.

these mapping schemes ensure correct translation. However, program context could help reduce fences after mapping [76], for instance, by omitting fences around thread-local accesses (e.g., on the stack) or by merging adjacent fences.

Optimizations. Although ArancinIR allows optimizations, through our proof efforts we observed that *splitting* memory accesses is invalid. For instance, it is incorrect to split a 16-bit store into two 8-bit stores, shown in Figure 5.

In the original program (left), terminating with X = 0x1234 happens only when $a = 0$ because the right thread completed before the left starts. However, by splitting the 16-bit store to X, we *incorrectly* introduce the additional outcome X = 0x1234, a = 0x34. This is not a consequence of AIMM's decisions, as the example considers Arm, while splitting is also incorrect on x86-64 and RISC-V. In general, correctly splitting accesses requires globally locking the split instructions.

5.5 Proofs

We mechanize all our proofs in the Agda [4] proof assistant. Regarding mixed-size accesses, we model the si relation as an *equivalence* relation between a pair of R-R or W-W accesses. Our particular proof mechanization challenges include:

- The definitions of the mixed and non-mixed architecture models for x86 and Arm are structured very differently, making our novel mechanized proofs very different from existing non-mixed proofs between x86 and Arm [37, 76].
- As the RISC-V model differs significantly from Arm, it required separate additional proof efforts.
- Modeling the si relation and related lemmas affects significant parts of the other proof components.

6 Hybrid Translation

Hybrid translation is the optimal translation scheme, providing both performance and completeness. Combining static and dynamic translation, and using them interchangeably is a challenging task. Specifically, *(i)* all information needed for dynamic translation, must be available at any time; *(ii)* the transitions between static and dynamic translations must be seamless. As shown in Figure 1, Arancini relies on runtime components for HBT, which will be explained in this section: *(i)* our hybrid binary format storing all necessary information for execution, and *(ii)* our runtime library mediating between statically and dynamically translated code, and the system environment (OS, libraries). We leave static- and dynamic-specific details for Sections 7 and 8 respectively.

6.1 Hybrid Binary Format

We now describe the layout of the hybrid binaries produced by the static engine, as well as the handling of shared objects.

Layout overview. Our hybrid binary format extends the standard ELF, and includes both guest and host instructions, with additional metadata. We copy all sections from the guest binary (`.text`, `.data`, etc.) into guest-specific sections of our hybrid ELF, while maintaining the original addresses. The host `.text` section is populated with the translation. We also generate a *static function map* mapping each translated symbol, i.e., function, to its guest counterpart.

Handling shared objects. Arancini is able to use translated and native shared libraries. We use the traditional Procedure Linkage Table (PLT) mechanism to resolve the final addresses of the symbols. We thus need to maintain the relocations and symbols of both guest and host shared library functions in our binary header. Calling a native shared function requires a wrapper to translate between guest and host Application Binary Interfaces (ABIs). These wrappers are statically generated in a similar fashion to Risotto [37].

6.2 Runtime Library

Arancini's runtime is responsible for executing guest code on the target architecture, and acts as an interface between the static and dynamic components, as well as between the program and the OS. In particular, the runtime: *(i)* initializes the guest environment, *(ii)* acts as an interface between the guest binary and the host OS, and *(iii)* provides the dynamic translation component described in Section 8.

Guest initialization. At startup, the runtime allocates a separate guest stack with environment variables and arguments. It also acts as the guest dynamic linker, initializing linker and standard library data structures, like the TLS area and the thread control block of the main thread.

Host OS interface. Arancini is a user-space translator and thus relies on the host OS to perform system calls. In cross-ISAs translation, system calls require a shim layer to match different ABIs and perform runtime management. Moreover, some system calls require additional intervention. For example, the `clone` system call, which creates a new thread, must be intercepted to set up the child's guest memory.

On-demand dynamic translation. The runtime is responsible for Arancini's dynamic translation capabilities, which trigger upon jumping to an address of which neither static nor previous dynamic translations exist. After the target block is translated, the runtime will jump to the translation.

7 Static Translation

The static translation engine translates all statically reachable code of the guest binary. It does so by lifting the guest instructions to ArancinIR, then to LLVM IR, and after optimizations, finally lowering it to host instructions. The static engine aims to cover as much of the guest binary to reduce the runtime overhead of dynamic translation, as well as to generate more efficient code, thanks to the aggressive function-level optimizations it can perform ahead of time. In the rest of this section, we use circled numbers to refer to Figure 6.

7.1 Lifting from guest to ArancinIR

Symbol discovery. We declare each guest function found in the ELF symbol table. Their definition must be resolved by statically linking against another translated object file, or dynamically linking a translated or native shared library.

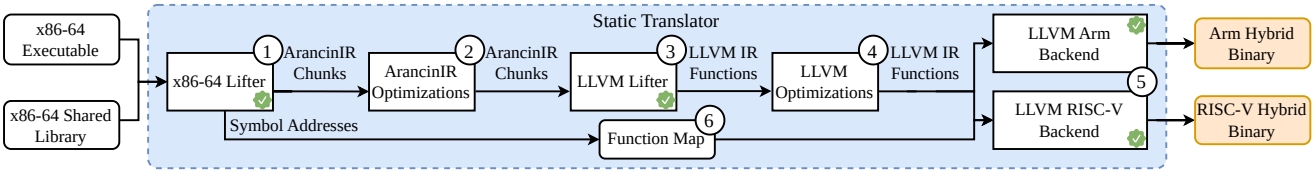

Figure 6. The static translation pipeline. Native executables and shared libraries are lifted to ArancinIR ①, optimized ②, lifted to LLVM ③, where the code is optimized again ④, and finally lowered to the respective target architecture ⑤. The function map ⑥ is used to find translated code for guest addresses at runtime.

Lifting. We lift all discovered function bodies to ArancinIR ①. Note that the mappings in Table 2 prohibit loads/stores without fences in AIMM. Hence, ArancinIR does not contain nodes that model plain guest loads/stores. We directly lift to nodes representing the memory access and fence pairs. We then perform a set of optimizations on ArancinIR ②, e.g., the flag elimination discussed in Section 4.

7.2 Lifting from ArancinIR to LLVM IR

Next we lift our optimized ArancinIR to LLVM IR ③, in order to leverage the LLVM infrastructure and its vast number of optimizations ④. We use the following mapping of abstractions between ArancinIR and LLVM IR: *ports* are mapped to values, *nodes* to instructions, and *chunks* to functions.

Function level. Each translated function contains: *(i)* a preamble that may load guest registers into host ones, *(ii)* the translation, and *(iii)* an epilogue that handles dynamic jumps and may store host registers to guest ones. Dynamic jumps are resolved in two tiers: Targets within a function are mapped in the epilogue, while tail calls outside of functions are handled by the global function map ⑥. If both lookups fail, dynamic translation is required.

Module level. All translated functions are placed in a single LLVM module. The entry point populates a function lookup table, mapping guest addresses to translated functions ⑥. This table acts as a vtable for dynamic control flow within the static translations. This permits guest dynamic calls to resolve almost directly to translated functions at runtime. Additionally, specific entries can be overwritten to target wrapped native library functions instead (see 6.1).

Optimizations and Code Generation. In LLVM IR, we merge multiple consecutive fence instructions.We replace them with a single, equivalently strong fence. Again, as guest memory access are only lifted together with their required fence, this optimization mainly addresses the following pattern: `RMOV;WMOV` ↦ `ld;Frm;Fww;st` which will be optimized to `ld;Fmm;st`. We run the fence-merging optimization during LLVM's `O2` optimization pipeline on the generated binary and produce machine code for the target architecture ⑤.

8 Dynamic Translation

In this section, we describe the dynamic translation pipeline as shown in Figure 7. The dynamic translator handles guest code discovered at runtime. It must thus be fast to avoid slowdowns. We first detail the two main translation stages:

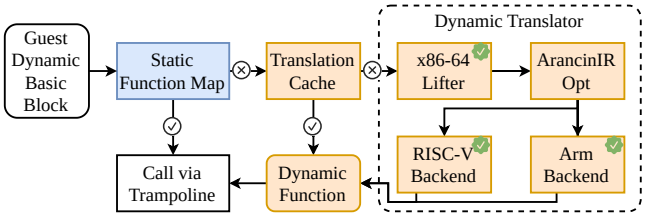

Figure 7. The dynamic translation pipeline. If code is not in the translation cache, the corresponding dynamic code is translated at runtime to the host ISA.

lifting to ArancinIR and then lowering to host ISA. We also discuss the runtime components that support the translation engine, including the caching and lookup mechanisms.

8.1 Translation Stages

Lifting to ArancinIR. We dynamically lift guest code to ArancinIR similarly to the static lifting explained in Section 7.1. The fundamental difference is the granularity of the translation, now being a single basic block, rather than an entire symbol. Each basic block, instantiates a *chunk*, and each contained guest instruction is then lifted into a *packet*.

Lowering to target ISAs. Architecture-specific backends generate host instructions from ArancinIR. We implement two backends: one each for Arm and RISC-V. Note that the implemented mappings from ArancinIR to Arm and RISC-V, defined in Table 2, guarantee correct translation of weak memory behaviors. Hence, backends do not need to perform expensive analyses to preserve the correct guest behavior.

Target-specific optimizations. If the host ISA supports status flags, or vector registers and instructions, backends use them. Our Arm backend supports both optimizations, however these specific features are not supported by RISC-V.

8.2 Runtime Integration

Workflow overview. When a jump target cannot be found in the static function map nor the translation cache, the runtime triggers dynamic translation, which also caches the result. Otherwise, existing translations are executed.

Caching and chaining basic blocks. To avoid retranslating blocks, we use a cache that stores all dynamically translated basic blocks. Additionally, when the jump instruction ending a block has a target already available in the cache, we chain blocks by patching the jump target in order to avoid unnecessary lookup operations.

Benchmark	Version	Lasagne (static)	Arancini Arm	Arancini RISC-V
histogram (hi)	pthread	y	y	y
	map-reduce	n	y	y
kmeans (km)	pthread	y*	y	y
	map-reduce	n	y	y
linear_regression (lr)	pthread	y	y	y
	map-reduce	n	y	y
matrix_multiply (mm)	pthread	y	y	y
	map-reduce	n	y	y
pca (pc)	pthread	n	y	y
	map-reduce	n	y	y
string_match (sm)	pthread	y*	y	y
	map-reduce	n	y	y
word_count (wc)	pthread	n	y	y
	map-reduce	n	y	y

Table 3. Coverage of the evaluated translators. Successfully lifted applications are marked *y*, successful unmodified ones *y**, failed translations *n*. Lasagne only supports Arm.

9 Evaluation

We evaluate Arancini around three dimensions: completeness (§ 9.1), performance (§ 9.2), and correctness (§ 9.3).

Experimental testbed (Arm and RISC-V). All x86-64 guest binaries, except when used for Lasagne, are compiled on a AMD EPYC 7713P 64-core processor, using Clang 18 and linked against Musl 1.2.3 [63]. Target and hybrid binaries are compiled on a ThunderX2-99xx 56-core processor or on a SOPHON SG2042 64-Core processor, for Arm and RISC-V respectively. For Lasagne, we use the provided artifact [75].

Baselines. Native binaries compiled to the target architecture serve as our baseline. We also compare against the state-of-the-art translators: Lasagne (static) and Risotto (dynamic). Where possible, we additionally use native functions of the host C library for all translators.

Benchmarks. We evaluate with the multi-threaded benchmark suite Phoenix [73] (see Table 3) supporting two versions: pthreads & map-reduce. We modified the pthread versions to support an adjustable number of threads, similar to the map-reduce version. We report the mean of 10 runs.

9.1 Completeness

RQ1. *Can Arancini translate a more diverse set of applications than state-of-the-art static translators?*

Arancini can translate an extensive set of applications, achieving DBT-like coverage, while translating applications that SBT cannot, as shown in Table 3. The map-reduce versions extensively use jump tables and function pointers, resulting in dynamic control flow, which Lasagne cannot translate statically. In summary, Arancini, as a HBT, achieves better coverage than a pure SBT, matching that of a DBT.

9.2 Performance

RQ2. *How performant are translations produced by Arancini, compared to state-of-the-art static and dynamic translators?*

Figure 8 shows that Arancini's translations are on par, or often faster than Risotto. When Lasagne translates a program, it performs better than the alternatives. Although Arancini also produces a static translation like Lasagne, it emulates the full guest state needed by the runtime, causing overheads.

Arancini translated binaries are 8.01× (Arm) and 4.52× (RISC-V) slower than native binaries, by geometric mean. However, linking to native libraries reduces the overheads to 6.01× and 3.81× respectively. Using native memory and string manipulation avoids translation of hard-to-optimize `rep` instructions, significantly improving performance.

Performance analysis. We dissect how static and dynamic translations are distributed in Arancini. First, we compare the number of unique addresses translated in the static and runtime phases. Unique instructions translated multiple times at runtime are only counted once for all threads.

The majority of the unique instructions are translated ahead of time, with *histogram* representing the highest percentage of dynamically translated unique instructions at 2.1%. All benchmarks exclusively translate the same eight instructions dynamically. Dynamic translation primarily triggers when guest instruction lookups fail in the middle of basic blocks, i.e., in the child thread after a `clone` system call.

Next, we use *perf* [30] to sample the shared objects in which most cycles are spent. Shared objects are either: *(i)* native libraries, *(ii)* Arancini's runtime library, *(iii)* static translations, or *(iv)* dynamically translated code.

Figure 9 confirms that statically translated code dominates the execution, with the exception of *pca*. The absolute runtime of *pca* is relatively shorter, where setup and transition code represents a higher runtime percentage. As almost no code is translated during runtime, no sample of dynamically translated code exists, using a 99Hz sampling rate.

Effectiveness of optimizations. Arancini applies two custom optimizations: Dead-flag elimination and fence merging, discussed in Section 4 and Section 7.2. Figure 10 shows their evaluation. We create three separate sets of translations, a fully optimized (default) and two with a single optimization disabled. We gain little performance with dead-flag elimination as LLVM's dead-code elimination performs a similar reduction. Although fence merging improves performance more, we found stalls from a combined fence are often comparable to those from two separate fences.

To summarize, execution times under Arancini are dominated by the statically translated workload. In our case, guest binaries are almost entirely handled statically, and the runtime is required primarily to map dynamic control flow to statically translated code rather than unknown guest code. Thus, performance is on par with static translators.

Scalability. Lastly, we evaluate the scaling of hybrid translation. In particular, fences emitted to preserve correct guest behavior can stall independent memory accesses needed by the runtime, thus potentially affecting the scaling behavior of

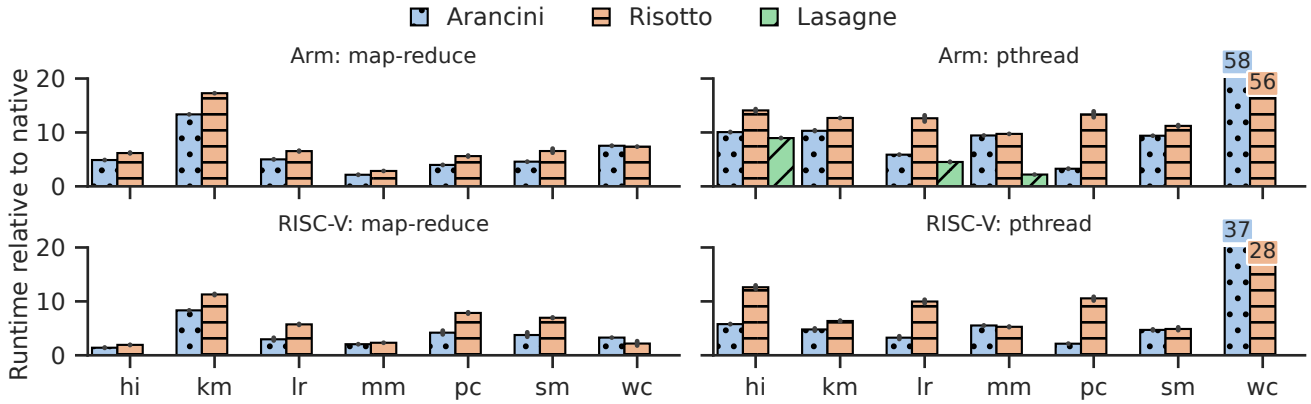

Figure 8. Comparison of end-to-end performance of multi-threaded benchmarks (16 threads), relative to a natively compiled program, on Arm and RISC-V. Missing bars for Lasagne are due its inability to lift the benchmarks. Lower is better.
Note: QEMU DBT provides incorrect translation for concurrent execution and is 6.7% slower than Risotto [37].

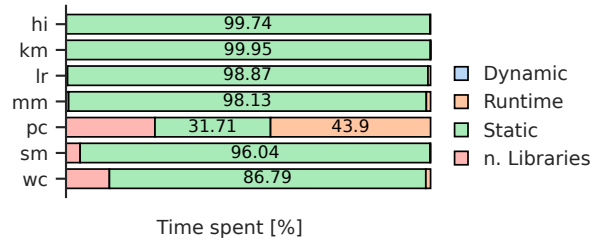

Figure 9. Distribution of executed code (pthread version).

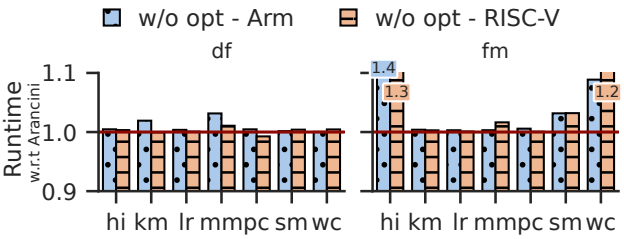

Figure 10. Runtime impact of disabling the proposed optimizations, dead-flag (df) and fence-merge (fm), over the fully optimized version of each target (red line). Lower is better.

a program. We evaluate this scalability aspect by running all benchmarks with an increasing number of threads. As shown by Figure 11, the scalability of Arancini is on par with the native binaries. Overall, neither global fences nor the runtime impact the scalability of concurrent programs. A full evaluation for Arm and RISC-V can be found in Appendix C.1.

9.3 Correctness

RQ3. *Does Arancini correctly map the strong guest memory model to the target models?*

Proven mappings. We provide proofs for the mapping schemes, as explained in § 5.5. Arancini follows those mappings when generating binaries. This results in correct and minimal fence placements in general, leading to the low overheads seen in § 9.2. As concrete examples, we translate different lock structures from the Linux kernel and C standard libraries. The atomic instructions used are listed in Table 4.

Correct mixed-sized accesses on Arm targets. Additionally `lockref` uses mixed-sized atomic accesses, which is covered by our mappings, and proven for the Arm target. The C code, guest and translated Arm assembly can be found in C.3.Overall, the mappings implemented in Arancini are proven correct, even for programs with mixed-sized concurrent accesses. This only holds when the target architecture has atomic instructions with the correct operand size.

10 Related Work

Binary translation. Traditionally, static translators work in a one-to-one fashion, i.e., translating from one ISA directly to another [10, 27, 84]. Recently, several SBTs have adopted IRs [16, 22, 24, 76, 82, 93, 94], enabling further optimizations and compilation to multiple targets. Other works have also investigated the adoption of neural-based strategies [15, 51]. Alternatively, there are also a number of dynamic translators [2, 12, 27, 29, 32, 35, 37, 38, 41, 58, 92, 96]. Microsoft's Windows-on-Arm machines dynamically emulate x86-64 binaries through the WOW64 layer [61] and use existing native shared libraries, by previously uniting x86-64 and Arm ABIs, which requires control over the entire software ecosystem.

More recently, Lasagne (SBT) [76] and Risotto (DBT) [37] support the correct translation of concurrent programs from x86-64 to Arm. Unfortunately, Lasagne is incomplete, and Risotto incurs high performance overheads compared to

Lock	Accesses	SLoC	Slowdown w.r.t native
lockref [59, 87]	cmpxchg 64 & xchg 32h	241	3.11
Array lock [39, 65]	xchg 8; xadd 8	162	3.34
CAS lock [65]	cmpxchg 32	151	3.50
MCS lock [39, 65]	cmpxchg 64; xchg 64	156	3.52
Rec MCS lock [65]	cmpxchg 64; xchg 64	147	3.56
Ticket lock [65, 69]	cmpchg 32; xadd 32	150	3.51

Table 4. Overview of lock primitives using atomic accesses, which get correctly translated by Arancini.

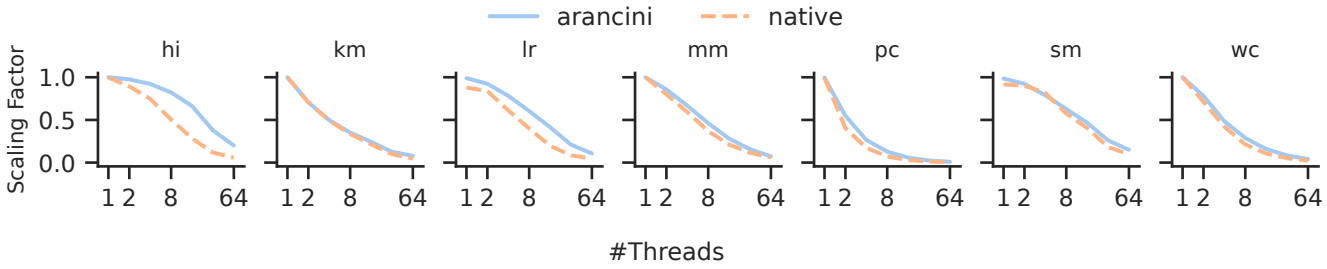

Figure 11. Scaling factor of all benchmarks with increasing number of threads, for native binaries and Arancini's hybrid binaries on Arm. A scaling factor of 1 is ideal.

static-only approaches. Moreover, both lack support for mixed-size concurrency models and translation to RISC-V.

Existing works explore the combination of static and dynamic approaches in a cross-ISA context [48, 66, 90, 96], while others use it for patching security vulnerabilities at the binary level [31]. Some works extend DBTs with static components, such as persistent code caches [91], or offline generated cache contents [90]. Others extend SBT with a runtime [44, 83, 95]. Specifically, Apple's Rosetta 2 [13] enforces weak memory correctness, by adopting a TSO execution mode in Apple's proprietary processors. In contrast, we advance the state-of-the-art by tackling the primitives of the latest architectures and memory model mismatches for concurrent programs generically in software.

Concurrency and memory models. Concurrency semantics creates complexities in program translations due to the underlying memory models. Correct compilation of high-level concurrency primitives to different architectures along with optimizations based on formal semantics is an active research area [18–20, 25, 26, 43, 56, 62, 68, 70, 77, 79, 89]. There are other approaches for translating programs from strong to weak memory models [36, 64, 86]. Different strategies have been proposed in the literature for checking the SC-robustness or stability and, in case of violation, insert fences to ensure stronger memory models [3, 6, 23, 52–55, 81]. However, using model checking to insert fences is computationally expensive and rarely scales beyond small programs.

11 Conclusion

We present Arancini, a hybrid binary translator for weak memory model architectures, thus executing x86 programs on Armv8 and RISC-V. Arancini implements formally verified mapping schemes between those architectures, where the mapping from x86 to Arm also ensures correct translation of mixed-size accesses. As a *hybrid* translator, Arancini is *more complete* than static translators, meaning it can translate more programs; while it is *more performant* than dynamic ones by up to 5× on the Phoenix benchmark suite.

11.1 Scope and Future Work

Arancini and our mapping schemes do not currently consider non-temporal accesses [28, 45]. Supporting those requires further *(i)* formal inspection of their respective models, *(ii)* modified mapping schemes, *(iii)* additional proofs, and *(iv)* implementing them in Arancini.

Arancini does not tackle the translation of x86 Advanced Vector Extensions. As we show, splitting memory accesses is unsound, translating 512-bit x86-64 accesses to Arm is unsound—as Arm lacks those wide registers, splitting is unavoidable. We foresee two methods to translate correctly: Either *(i)* the split accesses must be surrounded with a lock, or *(ii)* we require a mechanism to determine that multiple accesses to those locations never occur concurrently, making splitting sound.

Handling self-modifying code poses subtle challenges for Arancini's implementation: While writes to mapped guest code can be detected easily, invalidating all existing translations is not trivial. All threads must be stopped, and depending on the granularity of the translation, their executions may need to be rolled back. However, it can be assumed that existing self-modifying code is thread-safe, thus only one thread must undergo the expensive re-translation.

Arancini's runtime re-implements part of the guest's dynamic loader. Supporting a more diverse set of guest applications, such as ones compiled from C++, requires additional engineering to implement relocation types used by the implementations of `new` and `delete`.

Artifact and Supplementary Material. The Arancini system, along with the formal Agda proofs is publicly available as an open-source project, detailed in Appendix A. The appendices also contain additional formalization of the memory models, detailed examples of mapping mixed-size accesses, and additional evaluation results.

Acknowledgements

We thank our shepherd, Joseph Devietti, and the anonymous reviewers for their helpful comments and suggestions. This work was supported in part by an ERC Starting Grant (ID: 101077577) and the Chips Joint Undertaking (JU), European Union (EU) HORIZON-JU-IA, under grant agreement No. 101140087 (SMARTY), the Intel Trustworthy Data Center of the Future (TDCoF), and Google Research Grants. The authors acknowledge the financial support by the Federal Ministry of Research, Technology and Space of Germany in the programme of "Souverän. Digital. Vernetzt.". Joint project 6G-life, project identification number: 16KISK002.

References

[1] [n. d.]. C/C++11 mappings to processors. https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html.

[2] [n. d.]. FEX - Fast x86 emulation frontend. https://github.com/FEX-Emu/FEX.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Magnus Lång, and Tuan Phong Ngo. 2015. Precise and Sound Automatic Fence Insertion Procedure under PSO. In *Networked Systems*, Ahmed Bouajjani and Hugues Fauconnier (Eds.). Springer International Publishing, Cham, 32–47.

[4] Agda Development Team. 2024. *Agda documentation.* https://agda.readthedocs.io/

[5] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2, Article 8 (jul 2021), 54 pages. https://doi.org/10.1145/3458926

[6] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2017. Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion. *ACM Trans. Program. Lang. Syst.* 39, 2, Article 6 (May 2017), 38 pages. https://doi.org/10.1145/2994593

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

[8] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: modelling, simulation, testing, and data-mining for weak memory. *ACM Trans. Program. Lang. Syst.* 36, 2 (2014), 7:1–7:74. https://doi.org/10.1145/2627752

[9] Krste Asanovic Andrew Waterman. 2020. The RISC-V Instruction Set Manual Volume I Unprivileged ISA Document Version 20191213. https://github.com/herd/herdtools7/blob/master/herd/libdir/riscv-defs.cat.

[10] Kristy Andrews and Duane Sand. 1992. Migrating a CISC computer family onto RISC via object code translation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, USA) *(ASPLOS V)*. Association for Computing Machinery, New York, NY, USA, 213–222. https://doi.org/10.1145/143365.143520

[11] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. 2016. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *Proceedings of the 25th USENIX Conference on Security Symposium* (Austin, TX, USA) *(SEC'16)*. USENIX Association, USA, 583–600.

[12] Apple. 2021. Rosetta 2 on a Mac with Apple silicon. https://support.apple.com/fr-fr/guide/security/secebb113be1/web.

[13] Apple-Silicon. 2020. Addressing Architectural Differences in Your macOS Code. Available at https://developer.apple.com/documentation/apple-silicon/addressing-architectural-differences-in-your-macos-code.

[14] ARM. 2015. ARM Cortex-A Series Programmer's Guide for ARMv8-A. https://developer.arm.com/documentation/den0024/a/.

[15] Jordi Armengol-Estapé, Rodrigo C. O. Rocha, Jackson Woodruff, Pasquale Minervini, and Michael F. P. O'Boyle. 2024. Forklift: An Extensible Neural Lifter. arXiv:2404.16041 [cs.PL] https://arxiv.org/abs/2404.16041

[16] avast. [n. d.]. A retargetable machine-code decompiler based on LLVM. https://github.com/avast/retdec.

[17] Amazon AWS. [n. d.]. AWS Graviton Processor. https://aws.amazon.com/ec2/graviton.

[18] Maurice Bailleu, Dimitrios Stavrakakis, Rodrigo Rocha, Soham Chakraborty, Deepak Garg, and Pramod Bhatotia. 2024. Toast: A Heterogeneous Memory Management System. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques* (Long Beach, CA, USA) *(PACT '24)*. Association for Computing Machinery, New York, NY, USA, 53–65. https://doi.org/10.1145/3656019.3676944

[19] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. 2012. 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* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 509–520. https://doi.org/10.1145/2103656.2103717

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

[21] Martin Beck, Koustubha Bhat, Lazar Stričević, Geng Chen, Diogo Behrens, Ming Fu, Viktor Vafeiadis, Haibo Chen, and Hermann Härtig. 2023. AtoMig: Automatically Migrating Millions Lines of Code from TSO to WMM. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 61–73. https://doi.org/10.1145/3575693.3579849

[22] Lifting Bits. 2022. Framework for lifting x86, amd64, and aarch64 program binaries to LLVM bitcode. https://github.com/lifting-bits/mcsema.

[23] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. Checking and Enforcing Robustness against TSO. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 533–553.

[24] Ahmed Bougacha. 2022. Binary Translator to LLVM IR. https://github.com/repzret/dagger.

[25] Soham Chakraborty and Viktor Vafeiadis. 2016. Validating optimizations of concurrent C/C++ programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. Association for Computing Machinery, New York, NY, USA, 216–226. https://doi.org/10.1145/2854038.2854051

[26] Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the Concurrency Semantics of an LLVM Fragment. In *CGO '17*. IEEE, Austin, TX, USA, 100–110. https://doi.org/10.1109/CGO.2017.7863732

[27] Jiunn-Yeu Chen and Wuu Yang. 2008. A Static Binary Translator for Efficient Migration of ARM based Applications. , 36–39 pages.

[28] Kyeongmin Cho, Sung-Hwan Lee, Azalea Raad, and Jeehoon Kang. 2021. Revamping hardware persistency models: view-based and axiomatic persistency models for Intel-x86 and Armv8. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 16–31. https://doi.org/10.1145/3453483.3454027

[29] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. 2017. Cross-ISA Machine Emulation for Multicores. In *CGO'2017*. IEEE Press, Austin, TX, USA, 210–220. https://doi.org/10.1109/CGO.2017.7863741

[30] Arnaldo Carvalho De Melo. 2010. The new linux'perf'tools. In *Slides from Linux Kongress*, Vol. 18. 1–42.

[31] Chinmay Deshpande, Fabian Parzefall, Felicitas Hetzelt, and Michael Franz. 2024. Polynima: Practical Hybrid Recompilation for Multithreaded Binaries. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 1126–1141. https://doi.org/10.1145/3627703.3650065

[32] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. 2011. PQEMU: A Parallel System Emulator Based on QEMU. In *ICPADS'11*. IEEE, Tainan, Taiwan, China, 276–283. https://doi.org/10.1109/ICPADS.2011.102

[33] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. 2017. Mixed-size concurrency: ARM, POWER, C/C++11,

and SC. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 429–442. https://doi.org/10.1145/3009837.3009839

[34] Andrei Frumusanu. 2020. 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.

[35] Sheng-Yu Fu, Ding-Yong Hong, Yu-Ping Liu, Jan-Jan Wu, and Wei-Chung Hsu. 2018. Efficient and retargetable SIMD translation in a dynamic binary translator. *Software: Practice and Experience* 48, 6 (2018), 1312–1330. https://doi.org/10.1002/spe.2573

[36] Chen Gao, Xiangwei Meng, Wei Li, Jinhui Lai, Yiran Zhang, and Fengyuan Ren. 2024. CrossMapping: Harmonizing Memory Consistency in Cross-ISA Binary Translation. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 1013–1028. https://www.usenix.org/conference/atc24/presentation/gao-chen

[37] 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 the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 107–122. https://doi.org/10.1145/3567955.3567962

[38] Yu-Chuan Guo, Wuu Yang, Jiunn-Yeu Chen, and Jenq-Kuen Lee. 2016. Translating the ARM Neon and VFP instructions in a binary translator. *Software: Practice and Experience* 46, 12 (2016), 1591–1615. https://doi.org/10.1002/spe.2394

[39] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2020. *The art of multiprocessor programming*. Newnes.

[40] HiSilicon. 2024. Kunpeng 920. Available at https://www.hisilicon.com/en/products/Kunpeng/Huawei-Kunpeng/Huawei-Kunpeng-920.

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

[42] Apple Inc. 2023. Apple unveils M3, M3 Pro, and M3 Max, the most advanced chips for a personal computer. Available at https://www.apple.com/newsroom/2023/10/apple-unveils-m3-m3-pro-and-m3-max-the-most-advanced-chips-for-a-personal-computer/.

[43] Suresh Jagannathan, Vincent Laporte, Gustavo Petri, David Pichardie, and Jan Vitek. 2014. Atomicity refinement for verified compilation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 27. https://doi.org/10.1145/2594291.2594346

[44] Saagar Jha. 2020. TSOEnabler – Kernel extension that enables TSO for Apple silicon processes. https://github.com/saagarjha/TSOEnabler.

[45] Artem Khyzha and Ori Lahav. 2021. Taming x86-TSO persistency. *Proc. ACM Program. Lang.* 5, POPL, Article 47 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434328

[46] Hyungseok Kim, Soomin Kim, and Sang Kil Cha. 2025. Towards Sound Reassembly of Modern x86-64 Binaries. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1317–1333. https://doi.org/10.1145/3676641.3716026

[47] Arun Kishan. 2024. Azure Cobalt 100-based Virtual Machines are now generally available. Available at https://azure.microsoft.com/en-us/blog/azure-cobalt-100-based-virtual-machines-are-now-generally-available/.

[48] Jyun-Kai Lai and Wuu Yang. 2022. Hyperchaining for LLVM-Based Binary Translators on the x86-64 Platform. *Journal of Signal Processing Systems* 94 (09 2022), 1569–1589. https://doi.org/10.1007/s11265-022-01803-1

[49] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[50] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, San Jose, CA, USA, 75–86. https://doi.org/10.1109/CGO.2004.1281665

[51] Celine Lee, Abdulrahman Mahmoud, Michal Kurek, Simone Campanoni, David Brooks, Stephen Chong, Gu-Yeon Wei, and Alexander M. Rush. 2024. Guess & Sketch: Language Model Guided Transpilation. arXiv:2309.14396 [cs.SE]

[52] J. Lee and D. A. Padua. 2001. Hiding relaxed memory consistency with a compiler. *IEEE Trans. Comput.* 50, 8 (2001), 824–833. https://doi.org/10.1109/PACT.2000.888336

[53] Alexander Linden and Pierre Wolper. 2011. A Verification-Based Approach to Memory Fence Insertion in Relaxed Memory Systems. In *Model Checking Software*, Alex Groce and Madanlal Musuvathi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 144–160.

[54] Alexander Linden and Pierre Wolper. 2013. A Verification-Based Approach to Memory Fence Insertion in PSO Memory Systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, Nir Piterman and Scott A. Smolka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 339–353.

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

[56] Shuyang Liu, John Bender, and Jens Palsberg. 2022. Compiling volatile correctly in java. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*.

[57] LLVM Team. [n. d.]. The LLVM Compiler Infrastructure. https://llvm.org/.

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

[59] Iason Marmanis, Michalis Kokologiannakis, and Viktor Vafeiadis. 2024. Model Checking C/C++ with Mixed-Size Accesses. https://doi.org/10.5281/zenodo.13938750.

[60] Iason Marmanis, Michalis Kokologiannakis, and Viktor Vafeiadis. 2025. Model Checking C/C++ with Mixed-Size Accesses. *Proc. ACM Program. Lang.* 9, POPL, Article 75 (Jan. 2025), 21 pages. https://doi.org/10.1145/3704911

[61] Microsoft. 2024. How x86 emulation works on ARM. https://learn.microsoft.com/en-us/windows/arm/apps-on-arm-x86-emulation.

[62] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 187–196. https://doi.org/10.1145/2491956.2491967

[63] Musl. [n. d.]. musl libc. https://musl.libc.org/.

[64] 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. 2021. 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 530–545. https://doi.org/10.1145/3445814.3446748

[65] Oberhauser, Jonas and Behrens, Diogo and Oberhauser, Lilith. [n. d.]. libvsync. https://github.com/open-s4c/libvsync.

[66] Joo-On. Oni, Fawnizu Azmadi Hussin, and Nordin Zakaria. 2018. Fine-Grained Overhead Analysis Utilizing Atomic Instructions for Cross-ISA Dynamic Binary Translation on Multicore Processor. In *2018 International Conference on Intelligent and Advanced System (ICIAS)*. IEEE, Kuala Lumpur, Malaysia, 1–6. https://doi.org/10.1109/ICIAS.2018.8540622

[67] Scott Owens. 2010. Reasoning about the Implementation of Concurrency Abstractions on x86-TSO. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6183)*, Theo D'Hondt (Ed.). Springer, 478–503. https://doi.org/10.1007/978-3-642-14107-2_23

[68] Gustavo Petri, Jan Vitek, and Suresh Jagannathan. 2015. Cooking the Books: Formalizing JMM Implementation Recipes. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 445–469. https://doi.org/10.4230/LIPIcs.ECOOP.2015.445

[69] Nick Piggin. 2008. x86: FIFO ticket spinlocks. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=314cdbefd1fd.

[70] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.* 3, POPL, Article 69 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290382

[71] QEMU. [n. d.]. Atomic operations in QEMU. https://qemu.readthedocs.io/en/latest/devel/atomics.html.

[72] QEMU. [n. d.]. the FAST! processor emulator. https://www.qemu.org/.

[73] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary R. Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA*. IEEE Computer Society, Scottsdale, AZ, USA, 13–24.

[74] H. G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. http://www.jstor.org/stable/1990888

[75] Rodrigo Rocha, Dennis Sprokholt, Martin Fink, Redha Gouicem, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. 2022. Lasagne: A Static Binary Translator for Weak Memory Model Architectures (Artifact). https://doi.org/10.5281/zenodo.6408463. https://doi.org/10.5281/zenodo.6408463

[76] Rodrigo C. O. Rocha, Dennis Sprokholt, Martin Fink, Redha Gouicem, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. 2022. 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* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 888–902. https://doi.org/10.1145/3519939.3523719

[77] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 311–322. https://doi.org/10.1145/2254064.2254102

[78] Shigeyuki Sato, Taiyo Mizuhashi, Genki Kimura, and Kenjiro Taura. 2025. Efficiently Adapting Stateless Model Checking for C11/C++11 to Mixed-Size Accesses. In *Programming Languages and Systems*, Oleg Kiselyov (Ed.). Springer Nature Singapore, Singapore, 346–364.

[79] Jaroslav Ševčík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *ECOOP 2008 – Object-Oriented Programming*, Jan Vitek (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 27–51.

[80] Agam Shah. 2021. 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/.

[81] Dennis E. Shasha and Marc Snir. 1988. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Trans. Program. Lang. Syst.* 10, 2 (1988), 282–312. https://doi.org/10.1145/42190.42277

[82] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. 2012. LLBT: an LLVM-based static binary translator. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (Tampere, Finland) *(CASES '12)*. Association for Computing Machinery, New York, NY, USA, 51–60. https://doi.org/10.1145/2380403.2380419

[83] Bor-Yeh Shen, Jyun-Yan You, Wuu Yang, and Wei-Chung Hsu. 2012. An LLVM-based hybrid binary translation system. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. 229–236. https://doi.org/10.1109/SIES.2012.6356589

[84] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. 1993. Binary translation. *Commun. ACM* 36, 2 (Feb. 1993), 69–81. https://doi.org/10.1145/151220.151227

[85] Tom Spink, Harry Wagstaff, and Björn Franke. 2019. A Retargetable System-Level DBT Hypervisor. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 505–520. https://doi.org/10.1145/3302516.3307357

[86] Runzhou Tao, Jianan Yao, Xupeng Li, Shih-Wei Li, Jason Nieh, and Ronghui Gu. 2021. Formal Verification of a Multiprocessor Hypervisor on Arm Relaxed Memory Hardware. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 866–881. https://doi.org/10.1145/3477132.3483560

[87] Linus Torvalds. 2013. lockref: implement lockless reference count updates using cmpxchg(). https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bc08b449ee14.

[88] Amin Vahdat. 2024. Introducing Google Axion Processors, our new Arm-based CPUs. Available at https://cloud.google.com/blog/products/compute/introducing-googles-new-arm-based-cpu.

[89] Jaroslav Ševčík. 2011. Safe optimisations for shared-memory concurrent programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 306–316. https://doi.org/10.1145/1993498.1993534

[90] Jun Wang, Jianmin Pang, Xiaonan Liu, Feng Yue, Jie Tan, and Liguo Fu. 2019. Dynamic Translation Optimization Method Based on Static Pre-Translation. *IEEE Access* 7 (2019), 21491–21501. https://doi.org/10.1109/ACCESS.2019.2897611

[91] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. 2016. A general persistent code caching framework for dynamic binary translation (DBT). In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (Denver, CO, USA) *(USENIX ATC '16)*. USENIX Association, USA, 591–603.

[92] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. 2011. COREMU: a scalable and portable parallel full-system emulator. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) *(PPoPP '11)*. Association for Computing Machinery, New York, NY, USA, 213–222. https://doi.org/10.1145/1941553.1941583

[93] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne,

Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 133–147. https://doi.org/10.1145/3373376.3378470

[94] S. Bharadwaj Yadavalli and Aaron Smith. 2019. Raising binaries to LLVM IR with MCTOLL (WIP paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Phoenix, AZ, USA) *(LCTES 2019)*. Association for Computing Machinery, New York, NY, USA, 213–218. https://doi.org/10.1145/3316482.3326354

[95] Zhaoxin Yang, Xuehai Chen, Liangpu Wang, Weiming Guo, Dongru Zhao, Chao Yang, and Fuxin Zhang. 2024. MFHBT: Hybrid Binary Translation System with Multi-stage Feedback Powered by LLVM. In *Advanced Parallel Processing Technologies*, Chao Li, Zhenhua Li, Li Shen, Fan Wu, and Xiaoli Gong (Eds.). 310–325.

[96] Yi-Ping You, Tsung-Chun Lin, and Wuu Yang. 2019. Translating AArch64 Floating-Point Instruction Set to the x86-64 Platform. In *Workshop Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP Workshops '19)*. Association for Computing Machinery, New York, NY, USA, Article 12, 7 pages. https://doi.org/10.1145/3339186.3339192

A Artifact Appendix

A.1 Abstract

This appendix helps to reproduce the experimental and theoretical results in § 5 and § 9. Our artifact reproduces:

- **Experiments** (§ 9) – We provide a Docker image with our hybrid binary translator Arancini, along with our evaluation setup. The evaluation setup includes the source code, compiled x86_64 binaries of the Phoenix benchmark suite, and scripts to generate the plots.

  In Sec. 9 there are several experiments: Fig. 8 shows the end-to-end comparison between Arancini and two other binary translators. Fig. 10 shows the runtime impact of disabling proposed optimizations. Fig. 11 shows the scaling behavior of translations produced by Arancini.

  Although this work is evaluated on Aarch64 and RISC-V, we only provide the Docker setup for Aarch64. Replicating the RISC-V environment is preferably done manually using the source code.
- **Proofs** (§ 5.4) – Another Docker image contains mechanized Agda proofs for our mapping schemes in Fig. 2.

A.2 Artifact check-list (meta-information)

- **Program:** Hybrid binary translator Arancini
- **Proofs:** Mechanized weak memory proofs in Agda
- **Data set:** Phoenix
- **Hardware:** x86_64 (e.g. AMD EPYC 7713P), Aarch64 (e.g. ThunderX2-99xx), RISC-V (e.g. SOPHON SG2042, *optional*). For each, at least 64 CPU-cores and 32GB RAM
- **Metrics:** GeoMean of end-to-end runtime, relative to a native execution
- **How much disk space required (approximately)?:** 50GB
- **How much time is needed to prepare workflow (approximately)?:** 30min
- **How much time is needed to complete experiments (approximately)?:** 5-10h (depending on the host architecture)
- **How much time is needed to complete proofs (approximately)?:** 5min
- **Publicly available?:** yes

A.3 Description

A.3.1 How to access

- Artifact: 10.5281/zenodo.18257167
- Arancini (source): github.com/binary-translation/arancini-exploration/tree/sr/bench
- Proofs (source): github.com/sourcedennis/arancini-proofs/

A.3.2 Hardware dependencies Hardware used for our experiments: x86_64 (e.g., AMD EPYC 7713P), Aarch64 (e.g., ThunderX2-99xx), and RISC-V (e.g., SOPHON SG2042). Notably, the RISC-V system should *(i)* scale to a meaningful number of threads, and *(ii)* compile larger binaries natively. The Docker image for the proofs require any x86_64 machine.

A.3.3 Software dependencies Most importantly:

- A recent Linux distribution
- Clang 18
- musl 1.2.3

For other dependencies, including the ones to build the system and the benchmarks from scratch, refer to `flake.nix` in the source code.

A.4 Installation

Our artifact can *either* natively be built from scratch *or* be executed from the Docker image we provide.

From source. We recommend the use of the reproducible Nix package manager: nixos.org/download/. The setup requires the feature `flakes` to be enabled: nixos.wiki/wiki/flakes. A simple setup **on the host** can then be achieved through the following commands:

```
## Aarch64 or RISC-V
# get a shell with all dependencies
nix develop

# build the main executable "txlat"
# and place it under result/bin/txlat
nix build

# test
./result/bin/txlat -O hello.tx \
  -I test/hello-world/hello-static-musl
./hello.tx
```

Optionally building programs from scratch, **on the guest**:

```
## x86_64
# build the benchmarks and place them under
# phoenix-x86_64
nix build ./scripts#phoenix.x86_64-linux \
  --out-link phoenix-x86_64
```

From the Docker image. Long-running parts of the evaluation setup have already been executed in a Docker image.

To enter a shell with dependencies, guest binaries, and an already compiled Arancini, run:

```
## Aarch64
docker run -it \
   rmrssebastian/arancini-system-aarch64:latest
```

A.5 Experiment workflow

After installation, translation is then done for each shared library and executable, **on the host**. Steps #1 and #2 are already done in the Docker image.

```
## Aarch64 or RISC-V
# libc first ~40min
result/bin/txlat \
   -I phoenix-x86_64/libc.so \
   -O txlat-<host>/libc.so

# libunwind ~5min
result/bin/txlat \
   -I phoenix-x86_64/libunwind.so \
   -O txlat-<host> libunwind.so

# executable ~5min
result/bin/txlat \
   -I phoenix-x86_64/histogram \
   -O txlat-<host>/histogram \
   -l txlat-<host>/libc.so \
   -l txlat-<host>/libunwind.so

# test
txlat-<host>/histogram \
   <...>/histogram_datafiles/small.bmp
```

Proof workflow. We provide a separate Docker image to check the proofs:

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

The above command is sufficient to check whether the proofs are correct, corresponding to the correctness of the mappings in Fig. 2. To inspect the theorems and their proofs in Agda, the proofs can be rendered to HTML (in local directory `html/`) and then viewed in any web browser:

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

The resulting file `html/Main.html` is a documented entry point into the proofs.

A.6 Evaluation and expected results

Experiments (§ 9) are performed by the `bench.py` script present in both the Docker image, and the source code. It can be configured via `scripts/conf.json`: Specifically, this configuration defines directories that contain translations, native host binaries or to-be-translated guest binaries.

The following command will collect all runtimes:

```
python3 scripts/bench.py
```

All results will be stored under `bench/`, with the newest results linked to `bench/latest`.

Plotting results from this directory is done through:

```
cd scripts && python3 timeplot.py
```

Which will output Figures 8, 10 and 11

Proofs. Executing the proofs checks all our files and their dependencies (e.g., Agda's standard library). If this proof checking throws no errors, they are correct. The theorems for the mappings (Fig. 2) are defined in `MapX86toAIMM.agda`, `MapAIMMtoArmv8.agda`, and `MapAIMMtoRISCV.agda`.

A.7 Experiment customization

We encourage translation of other x86-64 binaries. Limitations of what can be translated, can be found below.

A.8 Notes

Currently Arancini's implementation has two limitations: *(i)* Only binaries linked against `musl` libc can be translated. A shell with a usable toolchain can be accessed through:

```
nix develop ./scripts#phoenix.aarch64-linux
```

(ii) Translating C++ binaries is not supported, since the relocations needed for `new` and `delete` are not implemented. Thus, programs are currently limited to C.

For a full comparison, the artifact for Lasagne [75] is needed separately. Using it is best done together with the Arancini source code. As the used guest binaries slightly differ, our binaries must be copied into the Lasagne artifact, translated and the translations subsequently copied out.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae