Lasagne: A Static Binary Translator for Weak Memory Model Architectures

Rodrigo C. O. Rocha*
University of Edinburgh
United Kingdom

Dennis Sprokholt*
TU Delft
Netherlands

Martin Fink
TU Munich
Germany

Redha Gouicem
TU Munich
Germany

Tom Spink
University of St Andrews
United Kingdom

Soham Chakraborty
TU Delft
Netherlands

Pramod Bhatotia
TU Munich
Germany

Abstract

The emergence of new architectures create a recurring challenge to ensure that existing programs still work on them. Manually porting legacy code is often impractical. Static binary translation (SBT) is a process where a program's binary is automatically translated from one architecture to another, while preserving their original semantics. However, these SBT tools have limited support to various advanced architectural features. Importantly, they are currently unable to translate *concurrent* binaries. The main challenge arises from the mismatches of the *memory consistency model* specified by the different architectures, especially when porting existing binaries to a weak memory model architecture.

In this paper, we propose Lasagne, an end-to-end static binary translator with precise translation rules between x86 and Arm concurrency semantics. First, we propose a concurrency model for Lasagne's intermediate representation (IR) and formally proved mappings between the IR and the two architectures. The memory ordering is preserved by introducing fences in the translated code. Finally, we propose optimizations focused on raising the level of abstraction of memory address calculations and reducing the number of fences. Our evaluation shows that Lasagne reduces the number of fences by up to about 65%, with an average reduction of 45.5%, significantly reducing their runtime overhead.

*The first two authors contributed equally to this work.

PLDI '22, June 13–17, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9265-5/22/06…$15.00
https://doi.org/10.1145/3519939.3523719

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Theory of computation** → *Formal languages and automata theory*.

Keywords: Binary Translation, Memory Model, Compiler

ACM Reference Format:
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 (PLDI '22), June 13–17, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3519939.3523719

1 Introduction

The landscape of hardware is shifting with the evolution of new computer architectures. Recently, processors based on the Arm and RISC-V instruction set architectures (ISAs) are emerging in consumer devices to servers running in data centers, disrupting the dominance that x86 once had [66**?**]. There are many benefits to using these modern architectures, including increased performance, better power efficiency [24], and better license stability.

However, a major challenge is ensuring that existing legacy applications can still continue to work on these new architectures. They provide different architectural features and memory consistency models. Porting an application can be as easy as recompiling for the new architecture, but older applications may not have the source-code available, or they may contain hard-coded architectural intrinsics that make a source-code based recompilation approach non-viable [**?**].

Static Binary Translation (SBT) is a process for automatically rewriting, ahead-of-time, the machine code from the original architecture to a target architecture. Because SBT works on the machine code itself, access to the original source-code is not required. Crucially, the translation must preserve the semantics of the original binary, as specified by the original architecture, whilst also optimizing the target binary in

a discernible way. Although SBT tools have gained popularity [25, 26, 75, 76], their support of several advanced architectural features is often limited. For example, Microsoft's binary lifting prototype, called mctoll, was unable to lift the programs used in our evaluation.

Furthermore, these SBT tools are also unable to translate concurrent binaries [9, 13, 74]. This is due to the mismatches of the *weak memory consistency model* in different architectures, which governs the valid orderings of memory accesses. To address this problem, the translation tools must reason about the consistency models for *correct* and *efficient* translation.

In this paper, we address the challenge of developing efficient translation between x86 and Arm concurrency semantics through LLVM's intermediate representation (IR). The x86 and Arm architectures have different *memory ordering* semantics, which results in different *memory ordering* rules. Therefore, we need a concurrency model for the LLVM primitives that enables *precise* mapping schemes between LLVM and these architectures, while also allowing for code transformations. We note that none of the existing concurrency models [18, 19, 30, 33, 52] suffice to satisfy all these requirements. Hence, to bridge this gap, we propose LIMM (LLVM IR Concurrency Memory Model). We use this model to design *precise* mapping schemes and prove their correctness.

We implement our approach in Lasagne, an end-to-end static binary translator. We extensively evaluate the effectiveness of Lasagne using the Phoenix multi-threaded benchmark suite [57]. Our evaluation shows that Lasagne reduces the number of fences by up to 65%, with an average reduction of 45.5%; thus, significantly minimizing the fence overheads. Overall, our paper makes the following key contributions[1]:

- **Static binary lifting (§ 4):** First, we build a Binary Lifting tool capable of lifting *concurrent* binaries to the LLVM IR, while supporting several challenging architectural features. Our contributions have been merged into Microsoft's mctoll[2], as these functionalities are not fully supported by existing binary lifting tools.
- **IR refinement (§ 5):** Secondly, once the binary is lifted to the LLVM IR, we propose IR refinements to enable subsequent optimizations. Our IR refinement strategies are based on peephole optimizations and pointer parameter promotion. The IR refinement not only enables the standard LLVM optimizations for the target architecture, but it also aids in a significant reduction in the number of fences.
- **LLVM IR concurrency memory model (§ 6– 8):** Lastly, we propose the LLVM IR's concurrency model, named LIMM (§ 6). Based on LIMM, we design formally verified *precise* mapping schemes and also prove the correctness of the safe transformations in Agda (§ 7). We implement these mappings and optimizations in Lasagne (§ 8).

[1] Artifact: https://doi.org/10.5281/zenodo.6408463
[2] Microsoft's mctoll: https://github.com/microsoft/llvm-mctoll

$$\begin{array}{c} X=Y=0 \\ X=1; \,||\, Y=1; \\ a=Y; \,||\, b=X; \end{array} \quad \text{(SB)} \qquad \begin{array}{c} X=Y=0; \\ X=1; \,||\, a=Y; \\ Y=1; \,||\, b=X; \end{array} \quad \text{(MP)}$$

Figure 1. Non-SC outcome $a = b = 0$ of SB program is allowed in x86 and Arm. Outcome $a = 1, b = 0$ of MP program is disallowed in x86 but allowed in Arm.

2 Background and Motivation

2.1 Concurrency in Architectures

A prevalent programming paradigm in modern multi-core architectures is *shared memory concurrency*, where concurrently running threads communicate through shared memory accesses. These architectures provide the following concurrency primitives: *(1)* load (ld) that reads from memory, *(2)* store (st) that writes to memory, *(3)* atomic read-modify-write (RMW), and *(4)* fence operations to order memory accesses.

Weak memory concurrency in architectures. Concurrent programs are usually understood by *execution interleaving*, where shared memory accesses in each thread execute in program order, and threads interleave arbitrarily. This execution model results in *sequential consistency* (SC) [34]. However, many architectures exhibit additional program behaviors which cannot be explained by interleaving alone, mainly due to out-of-order execution. These additional non-SC behaviors are known as *weak memory* behaviors.

SB shows an example comparing SC and weak memory. While both architectures exhibit certain common weak memory behaviors, their weak consistency models vary significantly. For instance, in x86, shared read-read and write-write memory access pairs are ordered, which is not the case in Arm. As a result, a program may exhibit different behaviors on different architectures. For example, the outcome $a=1,b=0$ of the MP program in Figure 1 is disallowed in x86 but allowed in Arm. These subtle differences affect the correct translations between architectures, and hence the architectures' concurrency models require careful analysis.

x86. x86 provides mov instructions to perform 'load from' (ld) and 'store to' (st) operations. x86 also has a number of operations for RMW accesses. For instance, lock cmpxchg performs atomic compare-and-exchange operation on a memory location. Finally, x86 provides the MFENCE instruction that prevents memory accesses from being reordered across it.

Arm. Arm provides regular load (ld) and store (st) accesses, and load-linked (ll) and store-conditional (sc) pairs to construct RMW i.e. RMW $\triangleq \ell$: ll;cmp;bc ℓ';sc;bc $\ell;\ell'$: where cmp and bc are compare and jump instructions [33, 52]. Arm provides multiple fences such as DMBFF (full fence), DMBLD that prevent load-load and load-store pairs from being reordered, and DMBST that only orders store-store pairs. Arm also has release and acquire accesses, which act as half fences. We handle these accesses in details in Appendix A.

$$\begin{matrix} X=1; \\ Y=1; \end{matrix} \Big\| \begin{matrix} a=Y; \\ b=X; \end{matrix} \xrightarrow{\text{mctoll}} \begin{matrix} X_{\text{NA}}=1; \\ Y_{\text{NA}}=1; \end{matrix} \Big\| \begin{matrix} a=Y_{\text{NA}}; \\ b=X_{\text{NA}}; \end{matrix} \xrightarrow{\text{opt}} \begin{matrix} Y_{\text{NA}}=1; \\ X_{\text{NA}}=1; \end{matrix} \Big\| \begin{matrix} a=Y_{\text{NA}}; \\ b=X_{\text{NA}}; \end{matrix} \xrightarrow{\text{codegen}} \begin{matrix} Y=1; \\ X=1; \end{matrix} \Big\| \begin{matrix} a=Y; \\ b=X; \end{matrix}$$

(a) x86 **(b)** Unoptimized LLVM IR **(c)** Optimized LLVM IR **(d)** Arm

Figure 2. Example of incorrect x86 to Arm translation by mctoll + LLVM. Suffix NA denotes the non-atomic accesses in LLVM IR. Outcome $a=1,b=0$ is disallowed in x86 program but allowed in the generated Arm program.

2.2 Binary Lifting

Our approach is based on static binary translation. Similar to modern compilers, the architecture of modern static binary translators also have a 3-phases structure. In both cases, their first phase (the compiler frontend and binary lifter) translate the input program to an intermediate representation (IR), e.g., LLVM IR. This IR code is then optimized and finally compiled down to its final binary format for a given architecture. There are two key benefits to this approach: First, the lifted code can be re-targeted to multiple architectures. Second, existing optimizations directly used on the lifted code.

The two fundamental principles of compilation [71] also apply to binary translation. First, binary translation must preserve the semantics of the input program, and second, the output program must be optimized in a discernible way. In this paper, we address both principles for concurrent programs.

State-of-the-art binary lifters. Lifting the source binary requires the binary translator to correctly map target to source instructions, discover global values, and reconstruct the control flow graph. Several state-of-the-art binary lifters target an intermediate representation, e.g. LLVM IR, to ease this process [9, 13, 15, 20, 27, 68, 69, 74, 76**?**]. Binary translators can operate statically or dynamically. The former allows more aggressive optimizations at the cost of being unable to handle dynamic jumps known at run time only. The latter handles those jumps but usually lifts at the basic block granularity, hindering optimization capabilities.

Limitations. Lifting full programs is a challenging and laborious task. Most existing tools are incapable of lifting the programs we use in our evaluation because they lack support for floating-point operations, SSE-based packed instructions as well as integer-based ones. The main difficulty in supporting these operations stems from the lack of abstract information in the source binary regarding types, control flow or function calls. Additionally, these lifting tools primarily target sequential programs and do not handle concurrency, i.e., they ignore the differences in memory consistency models altogether.

2.3 Motivation: Translation for Concurrent Binaries

It is well-known that any transformation (mapping or optimization) written for sequential programs may not always be correct for concurrent programs [17, 48, 63, 72]. We note that the state-of-the-art SBT tools are written for sequential programs [9, 13, 15, 68, 74]. Hence, using them to translate concurrent programs may lead to erroneous program behavior.

As a concrete example, consider the translation in Figure 2. mctoll [74] lifts the x86 program in Figure 2a to the LLVM IR in Figure 2b, where it translates the shared variable accesses in x86 to non-atomic accesses. Next, LLVM reorders the shared memory non-atomic accesses (NA) and generates the optimized IR in Figure 2c. Finally, LLVM generates the Arm program in Figure 2d that may exhibit program outcomes that were originally not allowed in x86.

The error results from the lack of reasoning about concurrency at the IR level. To do so, the IR needs a concurrency model. Thus, the combination of mctoll and LLVM raises a question: *What is the concurrency model of the IR*?

A naive answer would be to insert full fences before or after all memory accesses to preserve correctness. However, full fences are costly in terms of performance and restrict a number of optimizations. To perform correct and efficient translation of concurrent programs, we require a concurrency model for the IR which fulfills the following desired properties:

- **Precise mapping schemes.** The IR concurrency model must facilitate precise mapping schemes from source to the IR as well as from IR to the target.
- **Optimized.** The IR should allow common transformations including shared memory access reordering, elimination and redundant fence elimination. The correctness of these transformations ensures that a compiler can safely apply the respective compiler optimizations on the IR.

3 Overview

Our approach targets statically translating an existing binary from a strong to weak memory model architecture. Figure 3 shows the overall workflow of Lasagne, our end-to-end static binary translator between x86 and Arm. The x86 architecture employs a strong Total Store Ordering (TSO) model whilst Arm implements a weaker memory model [4, 5, 55].

The key aspect in supporting strong-to-weak binary translation is the strategic placement of memory fences to correctly emulate the ordering behavior of the source architecture. Our overarching goal is to support *correct* and *optimized* placement of fences, so that we emulate the source architecture faithfully, without introducing run-time overheads.

#1: Binary lifting. First we lift x86 binaries into LLVM bitcode. This is achieved by progressively raising the level of abstraction of the machine code, through multiple passes over the code and with the help of different IRs. The main challenge in binary lifting comes from having to reconstruct, from the machine, higher-level abstractions that have been lost in the compilation process. While lifting the binary to LLVM IR, it is important to identify these abstractions in order to enable

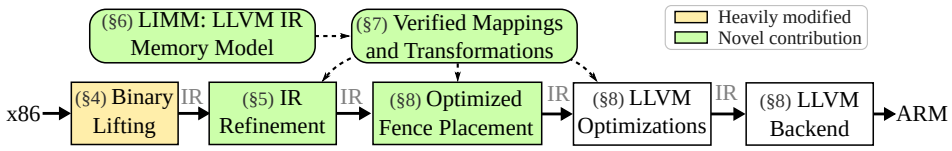

Figure 3. Lasagne overview.

more aggressive optimizations by the following stages. We address these challenges in Section 4.

#2: IR refinement. Once the binary is lifted to the LLVM IR, we perform some refinements to the code to better enable the upcoming optimizations. In this paper, we are focused on exposing pointer types throughout the lifted code, instead of handling integer-based raw memory addresses. Our approach is two-fold: First, we propose some peephole optimizations that reconstruct this information within functions; Second, we propose pointer parameter promotion, which replaces integer parameters that are used as raw memory addresses with an appropriate pointer type. We discuss the IR refinement process in Section 5.

#3: LIMM: IR concurrency model. As Lasagne translates from a strong to weak memory model, it inserts memory *fences* to achieve correctness. Excessively inserting fences degrades performance, whereas too few (or too weak) fences will lead to incorrect behavior. Approaching the minimum number of fences requires a careful understanding of concurrency models in different architectures.

We introduce LIMM (LLVM IR Memory Model), which acts as Lasagne's formal concurrency model. LIMM extends the concurrency primitives in the LLVM IR. The semantics of LLVM non-atomic accesses differ from both the corresponding x86 and Arm load and store accesses. In x86, ld-ld, ld-st, st-st access pairs are ordered, whereas non-atomic load and store accesses are always unordered. The Arm concurrency model disallows the removal of false dependencies [55] as these dependencies enforce certain orders between memory accesses. In contrast, LLVM regularly removes false dependencies in various optimizations . To allow these optimizations, LIMM does not order any accesses based on dependencies. We describe the details of LIMM in Section 6.

#4: Translation correctness in LIMM. Based on LIMM, we define precise mapping schemes for translating between architectures, and reason about the correctness of the common transformations on LIMM. More specifically, we identify the safe/unsafe reordering of independent shared memory accesses and fences. We also identify safe elimination of redundant shared memory accesses. The main challenge is to formally prove the correctness of the mapping schemes, and the safe transformations. We discuss the mapping schemes and the transformations in Section 7.

#5: Implementing LIMM translations. We implement our mapping schemes in Lasagne, which appropriately inserts fences into the refined LLVM IR. In particular, these schemes

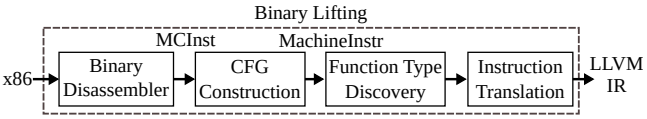

Figure 4. Lasagne's Binary Lifter based on mctoll.

demand leading or trailing fences for shared accesses. In Section 7, we describe reordering and elimination transformations, which LLVM regularly performs; these remain correct in LIMM. We discuss the further details in Section 8.

4 Binary Lifting

In this section, we describe the binary lifting component, which we develop based on Microsoft's mctoll [74]. Figure 4 shows the workflow for binary lifting, including the three different IRs used throughout this process, namely, `MCInst`, `MachineInstr`, and finally the LLVM IR. By lifting the binary to the LLVM IR, we are able to re-optimize the program, enabling us to exploit features that are specific of the target ISA [41, 47] or focus on a different objective function such as code-size reduction [21, 58–61].

First, the source binary is disassembled to an array of `MCInst`, which is the lowest-level IR in LLVM, working as an in-memory representation of the disassembled binary code. Afterwards, this monolithic array of `MCInst` is processed and control-flow graphs (CFG) are reconstructed using `MachineInstr`. These two low-level IRs are also used in the LLVM backend during code generation; however in the reverse order. Finally, the LLVM bitcode is generated, function by function, via multiple passes over the code, progressively lifting the level of abstraction, as shown in Figure 4. We detail each one of these components in the subsections below.

Our contributions to mctoll can be summarized as follows:

- We add support for floating-point arguments/return types and tail-calls.
- We add support for around 100 instructions (400 instruction variants), mainly SSE instructions, which were not supported before.
- We also add support for some of the affected flags, e.g., the parity flag.
- We implement additional x86 features such as global variables declared in header files, e.g., `stdout`.
- Finally, we fix several bugs discovered when lifting highly optimized programs, e.g., using `-O3`.

In total, we submitted 71 pull requests, 63 of which have been merged into the mctoll repository. Although we added support for almost 400 x86 instruction variants (or around 100

unique instructions) and several other x86 features, mctoll is still a work in progress.

4.1 Function Type Discovery

Before lifting a function or its calls to a strongly-typed language, such as LLVM IR, we need to know its function type, i.e., the list of parameters and return types. This function type discovery is based on the calling convention, which dictates how parameters and return types are arranged in the registers and stack memory. In this paper, we focus on the System-V application binary interface (ABI) [**?**].

Type discovery. First, we describe how we derive the types of a value in a specific register. This type discovery approach is used for both the list of parameters and return type.

Values in general-purpose registers are of integer type, with the width of the register as the width of the raised type, e.g. `EDI` → `i32`. As a result, pointers are also raised as raw 64-bit integer values, since pointers – at the machine level – are always manipulated using the general-purpose registers. We discuss this in more detail in Section 5.

For SSE registers, we need a different approach as they may hold either floating-point (FP) or packed values [53, 54]. While we can derive the integer type from the general-purpose registers alone, this is not possible for SSE registers. To derive the type of an SSE register, we need to analyze the instructions using it. We differentiate between two types of SSE instructions:

Packed instructions operate on vectors of integers or FP values. The discovered type is a 128-bit wide vector, the type and count of elements depends on the instruction, e.g. `ADDPD` (add packed double) → `<2 × double>`

Scalar instructions operate on single FP values. The discovered type is either a `float` or a `double`, e.g., `ADDSD` (add scalar double) → `double`

Parameter discovery. Detecting the parameters of a function involves a *live variable analysis* [6] of the registers' usage in the CFG. If a live register is one of the conventional parameter registers specified by the System-V ABI, and has no reaching definition, it is considered to be a parameter register. The System-V ABI allows for up to six integer and eight SSE-based parameters to be passed via registers, while additional ones will be spilled onto the stack.

Return type discovery. Discovering the return type of a function involves determining if one of the conventional return registers, `RAX` or `XMM0`, is alive at all the exiting blocks of the CFG. The return type can then be derived from such a register, as explained in Section 4.1. For functions where no return register is found, the type is set to `void`. If a function has multiple exit paths, the largest sized return type will be used.

4.2 Instruction Translation

In the last two passes, all instructions are lifted to the LLVM IR. Moreover, while traversing the CFG, we need to keep track of the values stored in the registers and the processor status flags, so operands can be correctly translated. First, we only lift non-terminator instructions. Then, the final pass lifts the terminator instructions, connecting the CFG in the LLVM bitcode.

Each `MachineInstr` can be translated to zero, one, or more LLVM instructions. Some instructions, such as copy operations, only update the internal record of which value the destination register is currently holding, generating no instruction in the lifted code. Instructions such as addition, where a simple LLVM counterpart exists, translate to a single LLVM instruction. Lastly, instructions that implicitly set processor status flags will result in more than one LLVM instruction. If any of these lifted instructions are unnecessary, they become dead code, later eliminated by traditional LLVM optimizations.

Programs may also contain *function calls*, that do not translate like instructions, as they take an arbitrary number of arguments – possibly of different types – requiring more consideration during translation.

4.2.1 Translating function calls.

In order to translate function calls, we examine the information gathered by the function type discovery and standard library header files. Similar to parameter discovery, the list of arguments passed in a function call also accounts for the conventional parameter registers specified by the System-V ABI. The LLVM values referenced by the parameter registers are passed as arguments to the function call. If the function returns a value, the tracking record of the return register is also updated accordingly.

Call to variadic functions. *Variadic* functions can have a different number of arguments passed in each callsite. All parameter registers *alive* at the callsite are passed an argument.

For SSE registers, the System-V ABI requires that the number of SSE-based arguments must be set in the `AL` register while calling a *variadic* function. Most compilers, such as LLVM, often explicitly assign a constant to the `AL` register. We try to leverage this behavior whenever possible. If we find a reaching constant value for the `AL` register, we look specifically for that amount of SSE registers. Otherwise, we fall back to the same approach used for general purpose registers.

Challenges with parameter ordering. When general purpose and SSE registers are passed as arguments to a function, we cannot reconstruct precisely the order of arguments from the original source code. This can be problematic for some *variadic* functions that depend on a specific argument order, e.g., `printf` from the C standard library. We can only preserve the order of the arguments within each group of registers.

In this paper, we assume that arguments passed via general purpose registers come before those passed via SSE registers. While this is not an issue when recompiling to ARM64 or another architecture that has a similar distinction between the two register groups, this may be an issue in the general case.

4.2.2 Handling of SSE Register Values.

As discussed in Section 4.1, SSE registers can be interpreted in several ways

depending on the instructions using them. Therefore, we need to cast the values stored in SSE registers accordingly.

For instructions operating only on the lower bits of an SSE register, zeroing the remaining bits, e.g., most scalar FP instructions, we ignore the zeroed upper bits. This is beneficial for consecutive SSE instructions that operate on the resulting value in a similar manner.

For instructions that also operate on higher bits, however, we cast the FP value to a vector type, zeroing the upper bits. Since casting happens implicitly in x86, without modifying any bit, we use bit-casts in LLVM. We have to handle three cases in particular, with *src* as the source type, *dst* as the destination type, and $|x|$ as the bit width of a given type x.

1. $|src| = |dst|$: In this case, a simple `bitcast` instruction is used to cast the value to the desired type.
2. $|src| < |dst|$: Since we assume missing bits to be zero, we first create a 128-bit wide vector of the type $<n \times src>$, where $n = \frac{|dst|}{|src|}$. Then, we insert the source value into position 0 of the created vector and bitcast the value to the destination one.
3. $|src| > |dst|$: Here, we first bitcast the source value to a bit vector of type $<n \times dst>$, where $n = \frac{|src|}{|dst|}$ and return the element at position 0.

Although we currently only support SSE instructions, our solution could be generalized to AVX2 and AVX-512. However, in this paper, we focus on SSE instructions used in the context of scalar floating-point computations.

4.2.3 Handling the stack memory.

mctoll reconstructs the stack memory of a lifted function by allocating a byte (`i8`) array. Offsets relative to the stack base pointer are translated to indexing operations on the lifted byte array. After raising all instructions in a function, the stack accesses are inserted with indices relative to the end of the stack.

5 IR Refinement

After lifting a program to LLVM IR, much information remains absent. For instance, in machine code, there is no difference between integer values and pointers, even less what *type* is pointed. However, this information *can* be represented in LLVM IR, and is invaluable to enable standard LLVM optimizations. Furthermore, this lack of information hinders our optimized fence placement algorithm. We aim to rediscover this information—in particular pointer arithmetic—after mapping to the IR, to enable optimizations.

Pointer parameters are lifted as 64-bit integer parameters and `inttoptr` instructions are used to convert integer-based addresses to pointer types just before a `load` or `store` instruction. Similarly, stack allocated memory has its base pointer converted to an integer via the `ptrtoint`.

In this section, we propose transformations that address this issue. Our approach is twofold. First, we use peephole optimizations to rewrite integer-based address computations to their pointer-based counterparts. Second, all integer parameters only used as input to `inttoptr` are modified to a pointer type, possibly also introducing pointer casts where needed.

Rule 1: Pointer casting

```
%0 = ptrtoint i8* %stacktop to i64
%RBP = inttoptr i64 %0 to i32*
=>
%RBP = bitcast i8* %stacktop to i32*
```

Rule 2: Stack offset

```
%tos = ptrtoint i8* %stacktop to i64
%0 = add i64 %tos, 16
%RBP = inttoptr i64 %0 to i32*
=>
%0 = getelementptr i8, i8* %stacktop, i64 16
%RBP = bitcast i8* %0 to i32*
```

Rule 3: Parameter offset

```
%0 = add i64 %arg, 8
%RBP = inttoptr i64 %0 to i32*
=>
%0 = inttoptr i64 %arg to i8*
%1 = getelementptr i8, i8* %0, i64 8
%RBP = bitcast i8* %1 to i32*
```

Figure 5. Peephole optimizations on pointer arithmetic instructions. `%stacktop` is a pointer to a stack allocated memory, `%arg` is an integer parameter, `%RBP` holds the resulting memory address. Note that `%RBP` could be a pointer of any type.

5.1 Exposing Pointers via Peephole Optimizations

The peephole optimizations comprise a collection of code patterns replaceable with another semantically equivalent piece of code. Figure 5 shows three examples of such patterns.

The patterns of code covered by these peephole rules mainly comprise integer-based arithmetic operations on the raw memory addresses. In this scenario, adding a number to a raw memory address is equivalent to offsetting an `i8` pointer by that amount through the `getelementptr` (GEP) instruction. The GEP instruction performs address calculation only, and does not access memory. The address calculation is based on the size of the base type, the base pointer, and the integer indices provided. This equivalence is illustrated by transformation ③ in Figure 5. In order to simplify our peephole rules, we always use `i8` as the base type of the GEP instructions, and then cast the resulting pointer to the expected pointer type.

5.2 Promoting Pointer Parameters

Since all pointer parameters are lifted as integer parameters, we need a way to identify when these integer parameters are actually used to represent a raw memory address. As illustrated by rule ③ in Figure 5, applying the proposed peephole optimizations can expose such integer parameters.

Our *pointer parameter promotion* works as follows: for each integer parameter of a given function, we collect all its uses. If all its users are `inttoptr` instructions, we mark it for a pointer type promotion, otherwise, we keep it as is. We choose the pointer type depending on all the destination pointer types of the `inttoptr` instructions. If all of them have the same destination pointer type, we promote it to that type and simply

delete all its `inttoptr` instructions. Otherwise, we promote it to an `i8` pointer, replacing all the `inttoptr` instructions to a bitcast to the appropriate destination pointer type.

6 LIMM: Concurrency Memory Model

In this section, we describe our next contribution. Lasagne translates *concurrent* programs from x86 to the IR to Arm. We intend to ensure that the translation remains correct based on the formal concurrency models. We first describe the general components of axiomatic concurrency models. We then compare the x86 and Arm axiomatic models. Finally, we propose LIMM—an axiomatic concurrency model for the LLVM-IR.

6.1 Axiomatic Model of Concurrency

In axiomatic semantics, a program is represented by a set of executions. An execution consists of a set of events and relations between these events. Our notations are similar to the cat language used for defining axiomatic models for concurrency [4].

Event. An event is generated from the execution of a shared memory access or fence. An event is represented by ⟨id,tid,lab⟩ where id is a unique identifier, tid is the thread identifier, and lab is the label of the event. A label lab = ⟨op,loc,val⟩ is a tuple where op denotes the corresponding memory access or fence operation. For memory accesses, loc denotes the corresponding memory location, while val denotes the read or written value. In case of fences, loc = val = ⊥. We respectively denote the set of read, write, and fence events by R, W, and F. Every memory location is initialized at the start of the execution, represented by a set of write events (where tid is 0). Unless otherwise mentioned, memory locations are initialized to zero.

Relation. Various binary relations connect the events in an execution. Given a binary relation S, we write $S^?, S^+, S^*, S^{-1}$ to denote its reflexive, transitive, reflexive-transitive closures, and inverse relations respectively. Relation S_{imm} denotes the immediate relation: $S_{\text{imm}}(a,b) \triangleq S(a,b) \wedge \nexists c\ S(a,c) \wedge S(c,b)$. We write $[A]$ to denote the identity relation on a set A, i.e. $[A](x,y) \triangleq x = y \wedge x \in A$. Given two relations S_1 and S_2, we denote their composition by $S_1;S_2$. In the model, an execution has the following relations between events:

- Relation program-order (po) is a strict partial order that denotes the syntactic order between the events. It is a strict *total* order on same-thread events.
- Relation reads-from (rf) relates every write event with the read events that read from it. Every read event reads from *exactly* one write event.
- Relation coherence-order (co) is a strict total order over same-location writes.
- A *successful* `RMW` results in an rmw relation between a pair of read and write events on the same location which are also in immediate-po relation; i.e. $\text{rmw} \subseteq ([\text{R}];\text{po}_{\text{imm}};[\text{W}])|_{\text{loc}}$ holds. A *failed* `RMW` results in only a single R event.

From these, we derive a number of other relations:

x86 axiom

(GHB) hb^+ is irreflexive where

$\text{ppo} \triangleq ((\text{W}\times\text{W}) \cup (\text{R}\times\text{W}) \cup (\text{R}\times\text{R})) \cap \text{po}$

$\text{implid} \triangleq \text{po};[At \cup \text{F}] \cup [At \cup \text{F}];\text{po}$

where $At \triangleq \text{dom}(\text{rmw}) \cup \text{codom}(\text{rmw})$

$\text{hb} \triangleq \text{ppo} \cup \text{implid} \cup \text{rfe} \cup \text{fr} \cup \text{co}$

Arm axiom

(external) ob is irreflexive where

$\text{ob} \triangleq (\text{obs} \cup \text{aob} \cup \text{dob} \cup \text{bob})^+$ where

$\text{obs} \triangleq \text{rfe} \cup \text{coe} \cup \text{fre}$

$\text{aob} \triangleq \text{rmw} \cup \ldots$

$\text{dob} \triangleq \text{addr} \cup \text{data} \cup \text{ctrl};[\text{W}] \cup \ldots$

$\text{bob} \triangleq \text{po};[\text{F}];\text{po} \cup [\text{R}];\text{po};[\text{F}_{\text{LD}}];\text{po} \cup [\text{W}];\text{po};[\text{F}_{\text{ST}}];\text{po};[\text{W}] \cup \cdots$

Figure 6. Distinguishing axioms in x86 and Arm. Both models satisfy sc-per-loc and atomicity axioms (Section 6.2). Full Arm model is in Appendix A.

- Relation from-read (fr) connects a pair of read and write events accessing the same memory location: $\text{fr} \triangleq \text{rf}^{-1};\text{co}$. If a read r reads-from a write w, while write w' on the same location is co-after w, then r and w' are in fr relation.
- We categorize the relations in an execution as either internal or external. If a relation S is between po-related events, then it is an internal-S relation and otherwise an external-S relation. For instance, we categorize the rf, co, fr relations in internal and external relations as follows:

$\text{rfi} \triangleq \text{rf} \cap \text{po}$	$\text{coi} \triangleq \text{co} \cap \text{po}$	$\text{fri} \triangleq \text{fr} \cap \text{po}$
$\text{rfe} \triangleq \text{rf} \setminus \text{po}$	$\text{coe} \triangleq \text{co} \setminus \text{po}$	$\text{fre} \triangleq \text{fr} \setminus \text{po}$

Execution. An execution X = ⟨E,po,rf,co,rmw⟩ is a tuple where X.E is the set of events and X.po, X.rf, X.co, X.rmw are set of po, rf, co, rmw relations between the events in X.E.

From programs to executions. A program consists of a set of initialization writes on all shared memory locations followed by a parallel composition of threads. In a program, the concurrency primitives generate the following events and relations during an execution:

- A store (st) that writes value v on a shared memory location x generates an event with label $\text{W}(x,v)$.
- A load (ld) that reads value v from a shared memory location x generates an event with label $\text{R}(x,v)$.
- A *successful* `RMW` on x that reads value v_r and writes value v_w generates events with labels $\text{R}(x,v_r)$ and $\text{W}(x,v_w)$ that are rmw-related. If the `RMW` reads value v' and *fails*, it generates a single read event with label $\text{R}(x,v')$.
- A full fence, e.g., `MFENCE` in x86 or `DMBFF` in Arm, generates an event with label F.

Consistency. Axiomatic memory models define several consistency constraints, which capture architectural properties. An execution satisfying these constraints is *consistent*.

$[[\mathbb{P}]]_M$ denotes the set of consistent executions of program $\mathbb{P}$ in memory model M.

Behavior. The behavior of a consistent execution is defined by the final values of all memory locations; the values written by the co-maximal writes (have no co-successors).

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

We use these events, relations, and definitions in all concurrency models, including those for x86 and Arm.

6.2 Comparing x86 and Arm Concurrency Models

We compare the axiomatic models of x86 and Arm [4, 5, 55]. The similarities and differences between these models guide us in defining the IR concurrency model.

Common axioms. Both the x86 and Arm architectures provide coherence and atomicity axioms. These properties are captured by the axioms described below.

Coherence. In an execution, coherence enforces *SC-per-location*: the memory accesses per memory locations are totally ordered. We capture this property as follows:

$$(\text{po}|_{\text{loc}} \cup \text{rf} \cup \text{co} \cup \text{fr})^+ \text{ is irreflexive.} \qquad \text{(sc-per-loc)}$$

Atomicity. Suppose r and w are the read and write events generated from a successful RMW. These events are in rmw relation. If there exists a write event w' between r and w on the same location, such that $\text{fre}(r,w)$ and $\text{coe}(w',w)$ hold, then the execution *violates* atomicity. Both x86 and Arm restrict atomicity violation with the (atomicity) axiom:

$$\text{rmw} \cap (\text{fre};\text{coe}) = \emptyset \qquad \text{(atomicity)}$$

Differentiating x86 and Arm. In Figure 6, we state the distinguishing axioms between x86 and Arm. Axiom (GHB) in x86 and axiom (external) in Arm enforce a global order on x86 and Arm executions, respectively.

x86. In x86, the read-read, read-write, write-write access pairs are ordered by the ppo relation. The access pairs are also ordered by the implid relation, an intermediate F event, or the rmw relation. Relation x86-happens-before (hb) is defined using ppo, implid, rfe, co, fr relations. Finally, axiom (GHB) enforces a global order on x86 executions.

Consider the x86 program in Figure 9 and its execution with the outcome $a=1, b=0$. The execution contains a $\text{ppo} \cup \text{fre} \cup \text{rfe}$ cycle which is disallowed according to the (GHB) axiom.

Arm. We follow the axiomatic model of Arm from Pulte et al. [55]. Arm defines atomic-ordered-by (aob), dependency-ordered-before (dob), barrier-ordered-by (bob), and observed-by (obs) relations to define the (external) axiom in Figure 6.

Relations aob, dob, bob order po-related events. Relation obs orders same-location events in different threads. Relation aob is based on rmw. Relation dob is defined using data, address, and control dependencies. These dependencies are captured by data, addr, and ctrl relations. Relation data, addr, and ctrl order a read to a po-successor write, read or write, and all events respectively. Relation bob is based on fences and synchronizing memory accesses. Relation obs is thread-external and relates same-location concurrent events. We use these relation to define relation ob. Finally, (external) axiom enforces a global order using the ob relation.

Unlike x86, the Arm model allows the execution with $a=1, b=0$ outcomes in the $\mathbb{P}_{\text{src}}$ program in Figure 9.

6.3 IR Concurrency Model

Primitives. We use LLVM primitives, in particular RMW_{SC} atomic accesses (i.e. RMW with `seq_cst` memory order), and the non-atomic load (ld_{NA}) and store (st_{NA}) instructions.

To order the non-atomics, LIMM uses various fences, such as Fsc (fence with `seq_cst` memory order) to enforce a full fence like `MFENCE` in x86 and `DMBFF` in Arm. LIMM introduces Frm and Fww into LLVM IR, which are similar to the `DMBLD` and `DMBST` fences in Arm. An Frm orders a load with its successor memory accesses (M refers to any memory access). Any Fww pair is ordered by an intermediate Fww fence.

Events. Given a program, we generate the following events and relations in an execution:

- For the non-atomic load and store accesses, we generate $\text{R}_{\text{NA}}(x,v)$ and $\text{W}_{\text{NA}}(x,v)$ events respectively.
- A successful $\text{RMW}_{\text{SC}}(x,v_r,v_w)$ generates a pair of $\text{R}_{\text{SC}}(x,v_r)$ and $\text{W}_{\text{SC}}(x,v_w)$ events which are rmw-related. If it reads v' and fails, it generates a single $\text{R}_{\text{SC}}(x,v')$. Relation rmw acts as a full fence similar to that of x86.
- The Fsc, Frm, Fww fences generate fence events with labels F_{SC}, F_{RM}, F_{WW} respectively.

Finally $\text{R} = \text{R}_{\text{NA}} \cup \text{R}_{\text{SC}}$ and $\text{W} = \text{W}_{\text{NA}} \cup \text{W}_{\text{SC}}$ hold in LIMM.

Relations. We define *order* (ord) and global-happen-before (ghb) relations in Figure 7. The ord relation orders po-related events. A pair of po-related events (a,b) are in ord relation in the following scenarios.

(ord_1) There is an intermediate F_{RM} event where a is a read and b is a memory access.
(ord_2) a and b are writes with an intermediate F_{WW} event.
(ord_3) a is a F_{SC} event or an event generated from RMW_{SC}.
(ord_4) Event b is an F_{SC} or a write generated from a successful RMW_{SC} access.

Note that we do not define any ordering based on dependencies in the IR. This is because LLVM may eliminate false dependencies. Such eliminations could introduce disallowed behavior, which would render the translations incorrect.

We can define a ghb relation on events across threads. On an execution graph, $\text{ghb}(a,b)$ implies a path from a to b by the combination of ord and external relations rfe, coe, fre.

Axioms. We now define the consistency axioms in Figure 7. The (sc-per-loc) and (atomicity) axioms are also present in the x86 and Arm memory models. The (GOrd) axiom ensures a global order between events.

$(\mathsf{po}|_{\mathsf{loc}} \cup \mathsf{rf} \cup \mathsf{fr} \cup \mathsf{co})$ is acyclic. (sc-per-loc)
$\mathsf{rmw} \cap (\mathsf{fre};\mathsf{coe}) = \emptyset$. (atomicity)
ghb is irreflexive where (GOrd)

$$\mathsf{ghb} \triangleq (\mathsf{ord} \cup \mathsf{rfe} \cup \mathsf{coe} \cup \mathsf{fre})^+ \text{ where}$$
$$\mathsf{ord} \triangleq [\mathsf{R};\mathsf{po};[\mathsf{F_{RM}}];\mathsf{po};[\mathsf{R} \cup \mathsf{W}]] \quad (\mathsf{ord}_1)$$
$$\cup\ [\mathsf{W};\mathsf{po};[\mathsf{F_{WW}}];\mathsf{po};[\mathsf{W}]] \quad (\mathsf{ord}_2)$$
$$\cup\ [\mathsf{F_{SC}} \cup \mathsf{R_{SC}} \cup \mathsf{codom}(\mathsf{rmw})];\mathsf{po} \quad (\mathsf{ord}_3)$$
$$\cup\ \mathsf{po};[\mathsf{F_{SC}} \cup \mathsf{W_{SC}} \cup \mathsf{dom}(\mathsf{rmw})] \quad (\mathsf{ord}_4)$$

Figure 7. LIMM Concurrency Model

x86	IR
ld	$\mathsf{ld_{NA}}$;Frm
st	Fww;$\mathsf{st_{NA}}$
RMW	$\mathsf{RMW_{SC}}$
MFENCE	Fsc

(a) x86 to IR

IR	Arm
$\mathsf{ld_{NA}}$	ld
$\mathsf{st_{NA}}$	st
$\mathsf{RMW_{SC}}$	DMBFF;RMW;DMBFF
Frm	DMBLD
Fww	DMBST
Fsc	DMBFF

(b) IR to Arm

x86		IR		Arm
ld	→	$\mathsf{ld_{NA}}$;Frm	→	ld;DMBLD
st	→	Fww;$\mathsf{st_{NA}}$	→	DMBST;st
RMW	→	$\mathsf{RMW_{SC}}$	→	DMBFF;RMW;DMBFF
MFENCE	→	Fsc	→	DMBFF

(c) x86 to IR to Arm

Figure 8. Verified mappings from x86 to Arm by the IR.

7 Translation Correctness

We use LIMM from the previous section to correctly reason about concurrency in the IR. Our main objective is to define a precise mapping from x86 to Arm, which goes through the IR. We also define precise mapping from Arm to x86 through the IR in Appendix B. Additionally, we reason about the IR to IR optimizing transformations on LIMM. We mechanize the correctness proofs in Agda.

7.1 Mapping Correctness

We propose the mapping schemes of the concurrency primitives between x86 and Arm through the primitives in IR. For each mapping scheme, we prove its correctness, as described in Theorem 7.1 and show that the scheme is precise.

Theorem 7.1 (Mapping Correctness). *Let* $\mathsf{M_s} \longrightarrow \mathsf{M_t}$ *be a mapping scheme which generates target program* $\mathbb{P}_t$ *from the source program* $\mathbb{P}_s$. *The scheme is correct if for each consistent target execution* $\mathsf{X_t} \in [[\mathbb{P}_t]]_{\mathsf{M_t}}$ *there exists a consistent source execution* $\mathsf{X_s} \in [[\mathbb{P}_s]]_{\mathsf{M_s}}$ *such that* $\mathsf{Behav}(\mathsf{X_t}) = \mathsf{Behav}(\mathsf{X_s})$.

Definition 7.2 (Precise Mapping Scheme). A correct mapping scheme is precise if for each fence used 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.

x86 to Arm Mappings. We obtain the x86 to Arm mappings in two steps: (1) x86 to IR and (2) IR to Arm in Figures 8a and 8b.

$$\begin{array}{l} X = Y = 0; \\ X = 1; \\ Y = 1; \end{array} \Big\| \begin{array}{l} a = Y; \\ b = X; \end{array}$$

(a) x86

$$\begin{array}{l} X = Y = 0; \\ X_{\mathsf{NA}} = 1; \\ \mathsf{Fww}; \\ Y_{\mathsf{NA}} = 1; \end{array} \Big\| \begin{array}{l} a = Y_{\mathsf{NA}}; \\ \mathsf{Frm}; \\ b = X_{\mathsf{NA}}; \end{array}$$

(b) IR

$$\begin{array}{l} X = Y = 0; \\ Y = 1; \\ \mathtt{DMBST}; \\ X = 1; \end{array} \Big\| \begin{array}{l} a = Y; \\ \mathtt{DMBLD}; \\ b = X; \end{array}$$

(c) Arm

Figure 9. x86, IR, Arm versions of MP program in x86 to IR to Arm translations by the proposed mapping schemes.

x86 to IR. The load and store in the IR is weaker than those of x86. So we map an x86 load to an IR load with a trailing Frm. The Frm orders a load with successor memory accesses. Similarly, an x86 store is mapped to an IR store with a leading Fww. The Fww orders the store with the predecessor store. An x86 RMW is mapped to an $\mathsf{RMW_{SC}}$ in the IR. A successful atomic-update acts as a full fence in both x86 and the IR. Finally, an MFENCE maps to an Fsc fence.

Theorem 7.3. *The mapping scheme in Figure 8a is precise.*

We prove Theorem 7.1 for the mapping scheme in Figure 8a. Now, we show that the Frm and Fww fences are required. In Figure 9a and Figure 9b, we show thean x86 program and the generated IR. The IR program disallows the outcome $a = 1, b = 0$ similar to the x86 program. The IR does not provide any weaker fences than Frm and Fww. Without any of these two fences, the outcome would be allowed and the mapping would be incorrect. Hence, the x86 to IR mapping scheme is precise.

IR to Arm. We map an IR load to Arm ld and IR store to Arm st. IR $\mathsf{RMW_{SC}}$ maps to an Arm RMW primitive, around which we insert leading and trailing DMBFF fences. Finally, we map Frm, Fww, Fsc fences in IR to DMBLD, DMBST, DMBFF accesses.

Theorem 7.4. *The mapping scheme in Figure 8b is precise.*

We prove the correctness of the mapping scheme. In the mapping, the leading and the trailing DMBFF fences are required and no weaker fence can be used as shown in Figure 10. Hence the mapping scheme is precise.

x86 to IR to Arm. In Figure 8c, we compose the x86 to IR and IR to Arm mappings. The resulting x86 to IR to Arm mapping is also precise. Consider the x86 and Arm programs in Figure 9a and Figure 9c. The DMBLD and DMBST Arm fences are required to preserve correctness. For the DMBFF fences with RMW, we can follow a similar example as shown in Figure 10. Hence the overall mapping is precise.

7.2 Correctness of Optimizing Transformations

We study the correctness of various transformations on LIMM. For the correct transformations, we show that the transformations preserve the following correctness theorem.

Theorem 7.5 (Transformation Correctness). *Let* $\mathbb{P}_s \longrightarrow \mathbb{P}_t$ *be a transformation on a memory model M. The translation is correct if for each consistent target execution* $\mathsf{X_t} \in [[\mathbb{P}_t]]_M$ *there exists a consistent source execution* $\mathsf{X_s} \in [[\mathbb{P}_s]]_M$ *such that* $\mathsf{Behav}(\mathsf{X_t}) = \mathsf{Behav}(\mathsf{X_s})$.

$$\begin{array}{l} X=Y=0; \\ X_{\mathrm{NA}}=1; \;\|\; Y_{\mathrm{NA}}=1; \\ \mathtt{RMW}_{\mathrm{SC}}(Y,0,2); \;\|\; \mathtt{RMW}_{\mathrm{SC}}(X,0,2); \end{array} \rightarrow \begin{array}{l} X=Y=0; \\ X=1; \;\|\; Y=1; \\ \mathtt{DMBFF}; \;\|\; \mathtt{DMBFF}; \\ \mathtt{RMW}(Y,0,2); \;\|\; \mathtt{RMW}(X,0,2); \\ \mathtt{DMBFF}; \;\|\; \mathtt{DMBFF}; \end{array}$$

Disallowed outcome $X=Y=2$.

$$\begin{array}{l} X=Y=0; \\ \mathtt{RMW}_{\mathrm{SC}}(X,0,2); \;\|\; \mathtt{RMW}_{\mathrm{SC}}(Y,0,2); \\ a=Y_{\mathrm{NA}}; \;\|\; b=X_{\mathrm{NA}}; \end{array} \rightarrow \begin{array}{l} X=Y=0; \\ \mathtt{DMBFF}; \;\|\; \mathtt{DMBFF}; \\ \mathtt{RMW}(X,0,2); \;\|\; \mathtt{RMW}(Y,0,2); \\ \mathtt{DMBFF}; \;\|\; \mathtt{DMBFF}; \\ a=Y; \;\|\; b=X; \end{array}$$

Disallowed outcome $a=b=0$.

Figure 10. Role of DMBFF fences in IR to Arm mapping. In Arm, the intermediate DMBFF fences restrict the outcomes. Any weaker (or no) fence would allow these outcomes in Arm and the translations would be incorrect.

$\downarrow a \setminus b \rightarrow$	R_{NA}	W_{NA}	R_{SC}	$\mathsf{R}_{\mathrm{SC}}\cdot\mathsf{W}_{\mathrm{SC}}$	F_{RM}	F_{WW}	F_{SC}
R_{NA}	✓	✓	✓	✗	✗	✓	✗
W_{NA}	✓	✓	✓	✗	✓	✗	✗
R_{SC}	✗	✗	✗	✗	✓	✓	✓
$\mathsf{R}_{\mathrm{SC}}\cdot\mathsf{W}_{\mathrm{SC}}$	✗	✗	✗	✗	✓	✓	✓
F_{RM}	✗	✗	✗	✓	=	✓	✓
F_{WW}	✓	✗	✓	✓	✓	=	✓
F_{SC}	✗	✗	✗	✓	✓	✓	=

(a) Reorderings $a\cdot b \rightsquigarrow b\cdot a$.

$$\begin{array}{ll} \mathsf{R}(X,v)\cdot\mathsf{R}(X,v') \rightsquigarrow \mathsf{R}(X,v) & \text{(RAR)} \\ \mathsf{W}(X,v)\cdot\mathsf{R}(X,v) \rightsquigarrow \mathsf{W}(X,v) & \text{(RAW)} \\ \mathsf{W}(X,v)\cdot\mathsf{W}(X,v') \rightsquigarrow \mathsf{W}(X,v') & \text{(WAW)} \\ \mathsf{R}(X,v)\cdot\mathsf{F}_o\cdot\mathsf{R}(X,v') \rightsquigarrow \mathsf{R}(X,v)\cdot\mathsf{F}_o & \text{(F-RAR)} \\ \mathsf{W}(X,v)\cdot\mathsf{F}_\tau\cdot\mathsf{R}(X,v) \rightsquigarrow \mathsf{W}(X,v)\cdot\mathsf{F}_\tau & \text{(F-RAW)} \\ \mathsf{W}(X,v)\cdot\mathsf{F}_o\cdot\mathsf{W}(X,v') \rightsquigarrow \mathsf{F}_o\cdot\mathsf{W}(X,v') & \text{(F-WAW)} \end{array}$$

(b) Eliminations where $o\in\{\mathrm{RM},\mathrm{WW}\}$ and $\tau\in\{\mathrm{SC},\mathrm{WW}\}$.

Figure 11. Reordering and elimination transformations on LIMM. In reorderings a and b memory accesses are on different locations and independent. $a\cdot b$ denotes that a and b are the labels of events that are related by $\mathsf{po}_{\mathsf{imm}}$. R_{SC} on its own represents a *failed* $\mathtt{RMW}_{\mathrm{SC}}$ read, while '$\mathsf{R}_{\mathrm{SC}}\cdot\mathsf{W}_{\mathrm{SC}}$' corresponds to a *successful* $\mathtt{RMW}_{\mathrm{SC}}$.

Reorderings. We study the correctness of the reordering of an adjacent shared memory access or fence pair a and b. In Figure 11a, we mark the safe (✓) and unsafe (✗) reorderings. The non-atomic accesses can be reordered freely as performed by LLVM. Non-atomic accesses *cannot* reorder with an $\mathtt{RMW}_{\mathrm{SC}}$ operation, as that requires reordering with both events in the rmw, which is disallowed by the reordering rules in Figure 11a. A store can reorder with a successor Frm and a load can reorder with a predecessor as well as a successor Fww. Any pair of fences can reorder safely.

We prove Theorem 7.5 to establish the correctness of the safe reorderings. We show that a reordering does not remove any ord relation from the target while defining the corresponding source execution.

Memory access eliminations. Figure 11b enlists the safe redundant access elimination transformations for read-after-read (RAR), read-after-write (RAW), and write-after-write (WAW) transformations. A RAR or RAW transformation eliminates the following read accesses and uses the value read or written by the first access. In the WAW transformation, the first write is redundant and can be eliminated safely. In these three LLVM transformations, the shared memory accesses are adjacent. In the next three transformations, the memory accesses are non-adjacent, with fences between the memory accesses.

Speculative load introduction. LIMM also supports speculative load introduction where a shared memory load access is hoisted outside the conditional. The read value is used only when the conditional holds. This transformation is regularly performed in LLVM optimizations e.g. 'SimplifyCFG'.

The transformation is correct on LIMM. A target execution contains the event corresponding to the speculative load which is absent in the source execution. However, in this case, the speculative load value remains unused and does not affect the program behavior.

Fence merging. We can safely merge a fence with an adjacent same or stronger fence. It is also safe to strengthen an Frm or Fww fence to a full fence Fsc. So given a pair of adjacent Frm and Fww fences, we can strengthen and merge them to create one Fsc, that is, $\mathsf{F}_{\mathrm{RM}}\cdot\mathsf{F}_{\mathrm{WW}} \rightarrow \mathsf{F}_{\mathrm{SC}}\cdot\mathsf{F}_{\mathrm{SC}} \rightarrow \mathsf{F}_{\mathrm{SC}}$.

Proof strategy. We prove the theorems for the correct transformations in the following steps. Given a M_{t}-consistent execution X_{t} of $\mathbb{P}_{\mathsf{tgt}}$, we (1) define a source execution X_{s} from $\mathbb{P}_{\mathsf{src}}$. Following the mapping scheme, the memory accesses in X_{s} have corresponding accesses in X_{t}. Similarly, the mapping ensures the rf, co, rmw relations in X_{s} correspond one-to-one to those in X_{t}. Then, (2) we relate the X_{s} and X_{t} relations that are used in M_{s} and M_{t} and show that X_{s} satisfies the axioms in M_{s}. Finally, we (3) show the $\mathsf{X}_{\mathsf{t}}.\mathsf{co}$ and $\mathsf{X}_{\mathsf{s}}.\mathsf{co}$ relations match and hence X_{t} and X_{s} have identical behaviors. In mapping schemes the source and target models differ and for the transformations on the IR both M_{s} and M_{t} are LIMM model.

We mechanize the proofs for the mapping schemes and transformations in about 12,000 lines of Agda [2].

7.3 Adopting LLVM Semantics

LLVM performs various optimizations based on undefined behaviors [31, 35]. However, compiled x86 programs have a defined behavior. Hence, performing these optimizations on a lifted program may result in unsound translation. Considering this issue, Lasagne assumes that mctoll's lifting is correct and produces LLVM programs that are free from undefined behavior. Lasagne also ensures sound translation for racy programs as LIMM allows the optimizations.

Benchmark	Abbrv.	# Functions	LoC
histogram	HT	4	171
kmeans	KM	7	235
linear_regression	LR	2	120
matrix_multiply	MM	3	179
string_match	SM	5	205

Table 1. Phoenix multi-threaded benchmark suite.

8 Implementing LIMM Translations

Lasagne is implemented on top of Microsoft's mctoll binary lifting tool and the LLVM compiler framework, both open-source projects. Most of our contributions have already been accepted and incorporated in the main mctoll repository.

Precise fence placement. We enforce x86 to IR mapping from Figure 8a on the lifted code. We perform the fence insertion in two steps.

1. For every load and store, we explore the use-def chain of their pointer operand. In this exploration, we ignore bitcast and `getelementptr` operations, looking for a potential stack allocation. If the access is performed on a stack address, then no fence is inserted. Otherwise, the access is conservatively treated as a shared memory access and fences are inserted following the mapping scheme from Figure 8a.
2. We merge pairs of fences in the same basic block if there is no instruction in between that may access the memory.

LLVM optimizations. After fence placement, we apply the LLVM optimizations. These optimizations are crucial to eliminate unnecessary code produced in the binary lifting process. We evaluate the most impactful optimizations in Section 9.4.

Code generation. We implement IR to Arm mapping scheme (Figure 8b) in the LLVM backend to generate Arm code.

9 Evaluation

9.1 Experimental Setup

Experimental testbed. All x86 binaries are compiled on a machine with a quad-core Intel Xeon CPU E5-2650, 64 GiB of RAM, running Ubuntu 18.04.3 LTS. All Arm binaries are compiled on a machine with a 16-core Arm Cortex-A72, 32 GiB of RAM, running Ubuntu 18.04.5 LTS.

Benchmarks. We base our evaluation on the multi-threaded Phoenix benchmark suite [57], detailed in Table 1. The Phoenix benchmark suite contains scalar floating-point computations implemented as SSE instructions. We omit two programs from the suite as mctoll lifts them incorrectly, resulting in a segmentation fault in one of them and an infinite loop in the other. Our initial investigation suggests that this issue arises from the way mctoll handles some stack accesses through the base pointer. Fixing this would have required to rewrite most of mctoll's stack handling code.

Methodology. To minimize the effect of measurement noise, we repeat all experiments 25 times. We report the average values and their 95% confidence intervals.

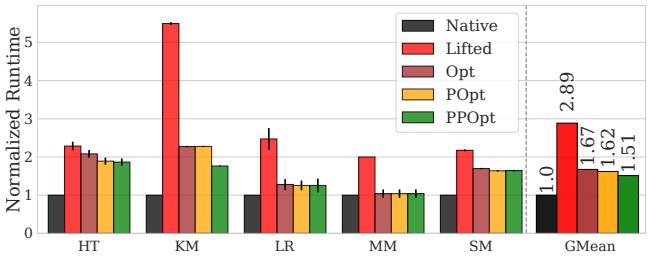

Figure 12. Performance with various level of optimizations normalized w.r.t. *Native* execution. Lower is better.

Baselines. We compare five versions of Arm programs:

- *Native* is a direct compilation from the C source code to an Arm binary with standard compiler optimizations.
- *Lifted* is a translation from an x86 binary to Arm via Lasagne, without re-optimizing the lifted LLVM IR. This version also excludes our IR refinement strategies, and the fence merging rules. It contains the *unoptimized* fence placement algorithm, in order to preserve program correctness.
- *Opt* extends the *Lifted* version by re-optimizing the lifted LLVM IR before re-compiling it down to Arm.
- *(Proposed+Opt)* (*POpt*) extends the *Opt* version by applying the fence merging rules proposed in Section 7.
- *Peephole+Proposed+Opt*(*PPOpt*) extends the *POpt* version with the IR refinement strategies proposed in Section 5. This version includes all the components described in Figure 3.

9.2 Overall Runtime Performance

Figure 12 shows our main performance results comparing all five versions described in Section 9.1. We report the normalized run time with respect to the *Native* version.

As expected, the *Native* version is the fastest one. This version has the benefit of compiling directly from the source code to the final target architecture. Meanwhile, the other versions must be translated in a conservative manner, following the semantics of the x86 architecture, thus naturally introducing some run time overheads. The unoptimized *Lifted* version is the slowest among all evaluated versions since it includes a significant amount of unnecessarily lifted code representing processor flag computations and other x86-specific features. All this unnecessary overhead is optimized away in the *Opt* version. Finally, we have the two versions that include the novel optimizations proposed in this paper, namely, *POpt* and *PPOpt*. The version fully optimized by Lasagne, *PPOpt*, has the best run time among all other translated variants, with an average normalized run time of 1.51, compared to an average of 2.89 of the unoptimized *Lifted* version.

Lasagne offers a statistically significant improvement for the `histogram` (`HT`) and `kmeans` (`KM`) benchmarks. For `KM`, in particular, the *Opt* version has a run time of around 3.85 seconds while the fully optimized *PPOpt* version has a run time of around 2.95 seconds, i.e., a speedup of around 1.3×.

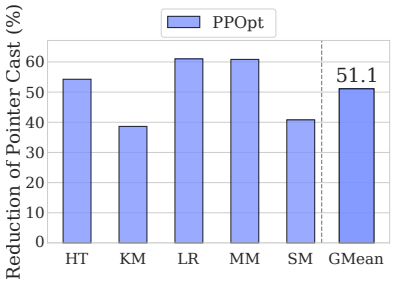

Figure 13. Percentage of pointer casting instructions removed relative to the unoptimized lifted code.

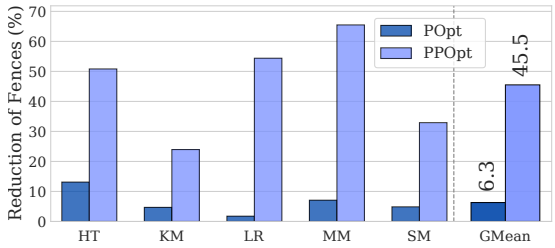

Figure 14. Fence reduction relative to the unoptimized lifted code with naive fence placement strategy.

9.3 Understanding the Impact of Each Optimization

Peephole optimizations. Figure 13 shows the percentage of `inttoptr` and `ptrtoint` instructions that can be optimized away by our IR refinement strategies. Our results show that about half, 51.1% on average, of these integer-pointer casting operations are removed. Most of the remaining `inttoptr` instructions belong to two different scenarios: (*i*) an integer parameter that could not be promoted to a pointer type; (*ii*) some level of pointer indirection, e.g., a raw memory address loaded from memory or returned by a function call which is later converted to a pointer type.

Optimized fence placement. Figure 14 shows how the IR refinement translates to fewer fences in the lifted LLVM bitcode. Our fence merging rules alone are capable of reducing the total number of fences by an average of 6.3%. However, the IR refinement enables Lasagne to reduce the total number of fences by an average of 45.5%. The IR refinement better exposes pointers used by load and store instructions, allowing our fence placement algorithm to avoid adding fences to operations involving the stack memory.

Figure 15 isolates the overhead reduction achieved by simply removing unnecessary fences in the unoptimized *Lifted* version. In this evaluation, we are excluding the impact of reducing the number of fences on other LLVM optimizations. The goal is to analyze the performance improvement of reducing the number of fences alone. Our results show that our fence merging rules has a significant impact on the `histogram` benchmark program while the IR refinement has a bigger impact on the `kmeans` benchmark program.

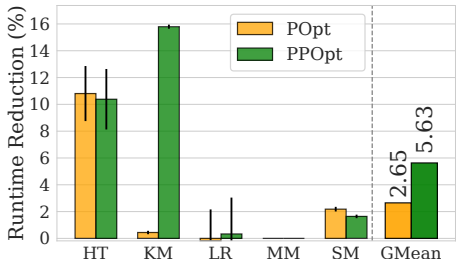

Figure 15. Performance improvement achieved by reducing the number of fences on the unoptimized lifted code.

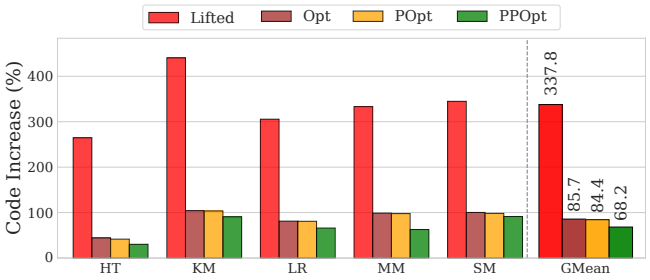

Figure 16. Code size increase, in terms of LLVM instructions, relative to the native compilation in Arm.

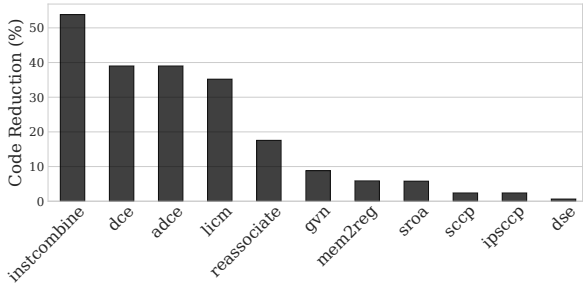

Figure 17. Code reduction on the `kmeans` benchmark with each optimization pass in isolation.

9.4 Overall Impact on Code Size

We analyze the difference in code size of the lifted LLVM bitcode with respect to the native version in Figure 16. All versions of lifted LLVM bitcode are larger than their native counterpart. The existing LLVM optimizations alone are capable of greatly reducing the size of the lifted code, with an 84.4% increase on average. However, the fully optimized version *PPOpt* is the smallest among all the translated versions, with a 68.2% increase on average. This reduction is a result of the combined effort of removing integer-pointer casting operations, inserting fewer fences, and the existing LLVM optimizations.

We also study the individual impact of optimization passes on the number of instructions on the `kmeans` program in Figure 17. We apply each optimization in isolation on the lifted LLVM bitcode after applying our IR refinement and the optimized fence placement. We report only the most impactful ones, as some of the optimizations either have no impact or result in a small code size bloat. However, some of the unreported optimizations might still be present in the

standard optimization pipeline, since they could improve run time performance. The four most impactful optimizations are: `instcombine`, LLVM's general peephole optimizations; `dce` and `adce`, basic and aggressive dead-code elimination; and `licm`, loop invariant code motion. These four optimizations alone reduce the total number of instructions, in the lifted LLVM bitcode, by at least 35%.

10 Related Work

Concurrency semantics. Correct and optimized compilation of concurrency in programming languages is a well studied problem [5, 11, 12, 19, 30, 33, 51, 52, 62, 65]. These models differ from LIMM in concurrency primitives and semantics. Considering the primitives, there is a semantic gap between the non-SC fences in C/C++/LLVM and Arm. Podkopaev et al. [52] propose an intermediate semantic model IMM that models C/C++ primitives. IMM use dependency-based ordering, which disallows many dependency-breaking LLVM optimizations. As our proposed model (LIMM) has no dependency-based ordering, we can safely apply those optimizations. What limits IMM for our use-case is two-fold: (1) Implementing the IMM-specified dependency-based-ordering checking is non-trivial. (2) LLVM optimizations have to be restricted to preserve correctness in the presence of dependencies.

In C/C++ models [12, 18, 19, 30, 33], data races result in undefined behavior . In LLVM, read-write race has defined behavior where the racy read returns undef and write-write races result in undefined behavior [18]. LIMM is closer to hardware memory models and has no undefined behavior. Kang et al. [30] is an operational model and [19] uses event structure to reason about multiple executions together whereas LIMM follows per-execution based semantics.

Program transformations under weak memory models have also been vastly explored [17–19, 28, 30, 37, 48, 63, 64, 72]. We analyze transformation correctness on IR concurrency model and use the same model to enable mappings between two different memory models, namely, x86 and Arm. We show the differences between x86 and Arm memory models, and then propose and prove correct an precise mapping scheme between them through proposed IR model.

Memory semantics enforcement. Robustness or stability based approaches check, by exploring executions, if a given program is SC-robust/stable against weaker models, inserting fences otherwise [1, 3, 14, 16, 32, 36, 38–40, 45, 50, 67]. Instead, we define a mapping scheme for translation tools that is valid for all programs and enforces x86 model instead of SC.

Binary lifting. A number of static binary lifters target the LLVM IR [9, 13, 15, 68, 74] for analysis and transformation. Previous work provided correct translations for SIMD [25, 26] or floating point instructions [26, 76]. However, these translation tools do not support concurrency.

McSema is a static binary translator that uses LLVM as its intermediate representation. Although McSema significantly covers x86 features, it is unable to correctly handle concurrency. Our solution could also be implemented in other static binary translators such as McSema. However, we chose mctoll because it is fully open-source, whereas McSema requires third-party commercial components.

Dynamic binary translation. Binary lifting is also used in a dynamic context, with cross-architecture binary or system emulation [8, 20, 23, 26, 27, 44, 73, 75]. ArMOR [44] proposes a framework to specify, compare and translate between memory consistency models, in a over-conservative manner. However, it lacks support for dependence-based orderings and `RMW` accesses in Arm. Pico [20] follows the ArMOR translation for POWER to x86 translation. However, Pico does not provide any formal guarantee of correctness. Lasagne differs in its static approach, but we also analyze dependence-based orderings and `RMW` accesses, while also providing a precise mapping from x86 to ARMv8 .

Rosetta 2 is a commercial tool for Apple Silicon [8] that translates x86 programs to the Arm ISA, where x86 ordering is enforced by Apple hardware [29]. Rosetta uses both static and dynamic translation. We are unable to compare with Rosetta 2, as it is not open-source and very little details have been made public. Concurrency-wise, Rosetta 2 handles the memory model mismatch by implementing both Arm and x86-TSO models in hardware, whereas Lasagne is a purely software-based solution. Moreover, Rosetta is platform-dependent (OS X and Apple M1 chips), preventing us from running our benchmarks targeting Linux ELF.

Peephole optimizations. Peephole optimization is a well-known technique used by optimizing compilers. Peephole optimizations identify certain code patterns and replace them with more efficient pieces of code [22, 46, 70]. Compilers may apply peephole transformations at different levels, for machine independent as well as for machine specific optimizations. Efforts have also been made to prove the correctness of peephole optimizations [42, 49].

11 Conclusion

In this paper, we present Lasagne, a static binary translator for weak memory model architectures. Lasagne is able to lift x86 binaries to LLVM IR and then compile it to Arm while enforcing the x86 memory ordering model. We provide formally verified mappings from x86 to LLVM IR to Arm and transformations on the IR, as well as peephole optimizations that drastically reduce the resulting binary's size and enable LLVM optimizations. We evaluate Lasagne and show that it generates efficient code in terms of size and performance.

Acknowledgments

We thank S. Bharadwaj Yadavalli for reviewing our patches to mctoll. We thank James Bornholt for shepherding our paper. This work was supported by a UK RISE Grant.

References

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Magnus Lång, and Tuan Phong Ngo. 2015. Precise and Sound Automatic Fence Insertion Procedure under PSO. In *NETYS (Lecture Notes in Computer Science, Vol. 9466)*. 32–47.

[2] Agda Development Team. 2021. *Agda 2.6.2 documentation*. https://agda.readthedocs.io/en/v2.6.2/

[3] 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 (2017), 6:1–6:38.

[4] Jade Alglave and Luc Maranget. 2022. herd7 consistency model simulator. http://diy.inria.fr/www/.

[5] 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.

[6] F. E. Allen and J. Cocke. 1976. A Program Data Flow Analysis Procedure. *Commun. ACM* 19, 3 (mar 1976), 137. https://doi.org/10.1145/360018.360025

[56]]porting-apple-silicon Apple. [n. d.]. Porting Your macOS Apps to Apple Silicon. https://developer.apple.com/documentation/apple-silicon/porting-your-macos-apps-to-apple-silicon.

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

[9] avast. 2022. A retargetable machine-code decompiler based on LLVM. https://github.com/avast/retdec.

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

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

[12] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ concurrency. In *POPL'11*. ACM, 55–66. https://doi.org/10.1145/1926385.1926394

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

[14] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. 2013. Checking and Enforcing Robustness against TSO. In *ESOP 2013*. 533–553.

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

[16] Soham Chakraborty. 2021. Robustness between Weak Memory Models. In *FMCAD'21*. 173–182.

[17] Soham Chakraborty and Viktor Vafeiadis. 2016. Validating optimizations of concurrent C/C++ programs. In *CGO'16*. ACM, 216–226. https://doi.org/10.1145/2854038.2854051

[18] Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the Concurrency Semantics of an LLVM Fragment. In *CGO '17*. IEEE, 100–110.

[19] Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding Thin-Air Reads with Event Structures. 3, POPL (2019). https://doi.org/10.1145/3290383

[20] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. 2017. Cross-ISA Machine Emulation for Multicores. In *CGO'2017*. IEEE Press, 210–220.

[21] Thaís Damásio, Vinícius Pacheco, Fabrício Goes, Fernando Pereira, and Rodrigo Rocha. 2021. Inlining for Code Size Reduction. In *SBLP'21* (Joinville, Brazil). Association for Computing Machinery, New York, NY, USA, 17–24.

[22] Jack W. Davidson and Christopher W. Fraser. 1980. The Design and Application of a Retargetable Peephole Optimizer. *ACM Trans. Program. Lang. Syst.* 2, 2 (April 1980), 191–202.

[23] 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*. 276–283. https://doi.org/10.1109/ICPADS.2011.102

[24] 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.

[25] 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

[26] 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

[27] 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 *CGO'12*. 104–113.

[28] Alan Jeffrey and James Riely. 2016. On Thin Air Reads Towards an Event Structures Model of Relaxed Memory. In *LICS 2016*. 759–767.

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

[30] Jeehoon Kang, Hur, Chung-Kil, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *POPL'17*. ACM.

[31] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. 2015. A formal C memory model supporting integer-pointer casts. In *PLDI 2015*. ACM, 326–335.

[32] Ori Lahav and Roy Margalit. 2019. Robustness against release/acquire semantics. In *PLDI 2019*. 126–141. https://doi.org/10.1145/3314221.3314604

[33] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *PLDI 2017*. 618–632. https://doi.org/10.1145/3062341.3062352 Technical Appendix Available at https://plv.mpi-sws.org/scfix/full.pdf.

[34] 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

[35] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming undefined behavior in LLVM. In *PLDI 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 633–647. https://doi.org/10.1145/3062341.3062343

[36] J. Lee and D. A. Padua. 2001. Hiding relaxed memory consistency with a compiler. *IEEE Trans. Comput.* 50, 8 (2001), 824–833.

[37] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: Global Optimizations in Relaxed Memory Concurrency. In *PLDI 2020*. 362–376.

[38] Alexander Linden and Pierre Wolper. 2011. A Verification-Based Approach to Memory Fence Insertion in Relaxed Memory Systems. In *SPIN'11*. 144–160.

[39] Alexander Linden and Pierre Wolper. 2013. A Verification-Based Approach to Memory Fence Insertion in PSO Memory Systems. In *TACAS*.

[40] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. 2012. Dynamic Synthesis for Relaxed Memory Models. In *PLDI '12*. 429–440.

[41] Yu-Ping Liu, Ding-Yong Hong, Jan-Jan Wu, Sheng-Yu Fu, and Wei-Chung Hsu. 2019. Exploiting SIMD Asymmetry in ARM-to-X86 Dynamic Binary Translation. *ACM Trans. Archit. Code Optim.* 16, 1, Article 2 (feb 2019), 24 pages.

[42] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. In *PLDI'15* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 22–32.

[56]]system-v-2021 H.J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger, and Mark Mitchell. [n. d.]. *System V Application Binary Interface*. https://gitlab.com/x86-psABIs/x86-64-ABI/-/jobs/1438137053/artifacts/file/x86-64-ABI/abi.pdf

[44] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. 2015. ArMOR: Defending against Memory Consistency Model Mismatches in Heterogeneous Architectures. In *ISCA'15*. 388–400.

[45] Roy Margalit and Ori Lahav. 2021. Verifying Observational Robustness against a C11-Style Memory Model. *Proc. ACM Program. Lang.* 5, POPL, Article 4 (2021). https://doi.org/10.1145/3434285

[46] W. M. McKeeman. 1965. Peephole Optimization. *Commun. ACM* 8, 7 (July 1965), 443–444. https://doi.org/10.1145/364995.365000

[47] Charith Mendis, Ajay Jain, Paras Jain, and Saman Amarasinghe. 2019. Revec: Program Rejuvenation through Revectorization. In *CC'19* (Washington, DC, USA). Association for Computing Machinery, New York, NY, USA, 29–41.

[48] 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 *PLDI'13*. ACM, 187–196.

[49] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. 2016. Verified Peephole Optimizations for CompCert. In *Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 448–461. https://doi.org/10.1145/2908080.2908109

[50] Jonas Oberhauser, R. Chehab, Diogo Behrens, Ming Fu, A. 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. *ASPLOS'21* (2021).

[51] Gustavo Petri, Jan Vitek, and Suresh Jagannathan. 2015. Cooking the Books: Formalizing JMM Implementation Recipes. In *ECOOP 2015*, Vol. 37. 445–469. https://doi.org/10.4230/LIPIcs.ECOOP.2015.445

[52] 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 (2019). https://doi.org/10.1145/3290382

[53] Vasileios Porpodas, Rodrigo C. O. Rocha, Evgueni Brevnov, Luís F. W. Góes, and Timothy Mattson. 2019. Super-Node SLP: Optimized Vectorization for Code Sequences Containing Operators and Their Inverse Elements. In *CGO 2019* (Washington, DC, USA). IEEE Press, Piscataway, NJ, USA, 206–216.

[54] Vasileios Porpodas, Rodrigo C. O. Rocha, and Luís F. W. Góes. 2018. VW-SLP: Auto-vectorization with Adaptive Vector Width. In *PACT '18*. ACM, New York, NY, USA, Article 12, 15 pages.

[55] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *PACMPL* 2, POPL (2018), 19:1–19:29. https://doi.org/10.1145/3158107

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

[57] 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, 13–24.

[58] Rodrigo C. O. Rocha, Pavlos Petoumenos, Björn Franke, Pramod Bhatotia, and Michael O'Boyle. 2022. Loop Rolling for Code Size Reduction. In *CGO'22*. 217–229.

[59] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, Kim Hazelwood, and Hugh Leather. 2021. HyFM: Function Merging for Free. In *LCTES'21* (Virtual, Canada). Association for Computing Machinery, New York, NY, USA, 110–121.

[60] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2019. Function Merging by Sequence Alignment. In *CGO'19* (Washington, DC, USA). IEEE Press, Piscataway, NJ, USA, 149–163.

[61] Rodrigo C. O. Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, and Hugh Leather. 2020. Effective Function Merging in the SSA Form. In *PLDI'20* (London, UK). Association for Computing Machinery, New York, NY, USA, 854–868.

[62] Susmit Sarkar, Kayvan Memarian, Scott Owens, Mark Batty, Peter Sewell, Luc Maranget, Jade Alglave, and Derek Williams. 2012. Synchronising C/C++ and POWER. In *PLDI'12*. ACM, 311–322.

[63] Jaroslav Sevcík. 2011. Safe optimisations for shared-memory concurrent programs. In *PLDI 2011*. 306–316.

[64] Jaroslav Sevcík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *ECOOP 2008*. 27–51.

[65] Jaroslav Sevcik and Peter Sewell. 2016. C/C++11 mappings to processors. https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html.

[66] 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/.

[67] 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

[68] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. 2012. LLBT: An LLVM-Based Static Binary Translator. In *CASES 2012*. 51–60. https://doi.org/10.1145/2380403.2380419

[69] Tom Spink, Harry Wagstaff, and Björn Franke. 2019. A Retargetable System-Level DBT Hypervisor. In *USENIX Annual Technical Conference*. USENIX Association, 505–520.

[70] Andrew S. Tanenbaum, Hans van Staveren, and Johan W. Stevenson. 1982. Using Peephole Optimization on Intermediate Code. *ACM Trans. Program. Lang. Syst.* 4, 1 (Jan. 1982), 21–36.

[71] Linda Torczon and Keith Cooper. 2007. *Engineering A Compiler* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[72] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. In *POPL'15*. ACM, 209–220.

[73] 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 *PPOPP'11*, Calin Cascaval and Pen-Chung Yew (Eds.). 213–222.

[74] S. Bharadwaj Yadavalli and Aaron Smith. 2019. Raising Binaries to LLVM IR with MCTOLL (WIP Paper). In *LCTES 2019*. 213–218. https://doi.org/10.1145/3316482.3326354

[75] Jiunn yeu Chen, Wuu Yang, Charlie Su, and Wei Chung Hsu. 2008. A static binary translator for efficient migration of ARM-based applications. In *Workshop on Optimizations for DSP and Embedded Systems*.

[76] Yi-Ping You, Tsung-Chun Lin, and Wuu Yang. 2019. Translating AArch64 Floating-Point Instruction Set to the X86-64 Platform. In *ICPP*.

A x86 and Arm Concurrency Models

A.1 x86 Model [4, 5].

Given a shared memory location x,

- `st` that writes value v on x generates an event with label $W(x,v)$.
- ld that reads value v from x generates an event with label $R(x,v)$.
- `RMW` operations e.g. `LOCK CMPXCHG` or `XCHG` instructions perform atomic read-modify-write operations (`RMW`). A successful `RMW` on x that reads value v' and writes value v generates a pair of events e and e' with labels $R(x,v')$ and $W(x,v)$ respectively such that $(e,e') \in [R];rmw;[W]$ holds. A failed `RMW` generates an $R(x,v')$ event.
- An `MFENCE` generates an event with label F.

Relations. x86 defines following relations.

- Relation *preserved-program-order* (ppo).

$$ppo \triangleq ((W\times W)\cup(R\times W)\cup(R\times R))\cap po$$

- Relation implid for fences.

$$implid \triangleq po;[dom(rmw)\cup F]\cup[codom(rmw)\cup F];po$$

- Relation *happens-before* (hb).

$$hb \triangleq ppo\cup implid\cup rfe\cup fr\cup co$$

Consistency Constraints. x86 consistency constraints are as follows.

- $(po|_{loc}\cup rf\cup fr\cup co)$ is acyclic. (sc-per-loc)
- $rmw\cap(fre;coe)=\emptyset$ (atomicity)
- hb is acyclic. (GHB)

A.2 ARMv8 Model [55]

- Load (ld) or load-linked (ll) that reads value v from x generates an event with label $R(x,v)$.
- Store (st) or store-conditional (sc) that writes value v on x generates an event with label $W(x,v)$.
- Acquire-read (ld_A) or acquire-load-linked (ll_A) that reads value v from x generates an event with label $A(x,v)$.
- AcquirePC-read (`LDAPR`) that reads value v from x generates an event with label $Q(x,v)$.
- Release-write (st_L) or release-store-conditional (sc_L) that writes value v on x generates an event with label $L(x,v)$.
- Atomic read-modify-write operations are as follows.

$$RMW \triangleq \mathbb{P} : ll;mov;teq\ \mathbb{Q};sc;teq\ \mathbb{P};\mathbb{Q}:$$
$$RMW_A \triangleq \mathbb{P} : ll_A;mov;teq\ \mathbb{Q};sc;teq\ \mathbb{P};\mathbb{Q}:$$
$$RMW_L \triangleq \mathbb{P} : ll;mov;teq\ \mathbb{Q};sc_L;teq\ \mathbb{P};\mathbb{Q}:$$
$$RMW_{AL} \triangleq \mathbb{P} : ll_A;mov;teq\ \mathbb{Q};sc_L;teq\ \mathbb{P};\mathbb{Q}:$$

  - A successful `RMW` generates [R];rmw;[W] and if an `RMW` fails then it generates an R event.
  - A successful RMW_A generates [A];rmw;[W] and if an RMW_A fails then it generates an A event.
  - A successful RMW_L generates [R];rmw;[L] and if it fails then it generates an R event.
  - A successful RMW_{AL} generates [A];rmw;[L] and if it fails then it generates an A event.
- Full fence `DMBFF` generates a F event.
- Load fence `DMBLD` generates a F_{LD} event.
- Store fence `DMBST` generates a F_{ST} event.
- Control fence `ISB` generates a `ISB` event.

Relations. ARMv8 defines various relations.

Relations among events: $A\subseteq R,Q\subseteq R,L\subseteq W$.

Dependency relations are same as ARMv7 except `ISB` is an event in ARMv8.

Relation coherence-after (ca) orders read or write to a fr-after or co-after write.

$$ca \triangleq fr \cup co$$

Relation observed-by(obs) constitute of thread external fre, fre, and coe relations.

$$obs \triangleq rfe \cup coe \cup fre$$

Relation atomic-ordered-by(aob) is derived from rmw and rfi relations.

$$aob \triangleq rmw \cup [codom(rmw)];rfi;[A\cup Q]$$

Relation dependency-ordered-before(dob) captures dependency based ordering among a pair of events in a thread

$$\begin{aligned} dob \triangleq\ & addr \cup data \\ & \cup ctrl;[W] \\ & \cup (ctrl\cup(addr;po));[ISB];po;[R] \\ & \cup addr;po;[W] \\ & \cup (ctrl\cup data);coi \\ & \cup (addr\cup data);rfi \end{aligned}$$

Relation barrier-ordered-by(bob) orders events by fences or results from stronger memory accesses.

$$\begin{aligned} bob \triangleq\ & po;[F];po \\ & \cup [L];po;[A]; \\ & \cup [R];po;[F_{LD}];po \\ & \cup [A\cup Q];po \\ & \cup [W];po;[F_{ST}];po;[W] \\ & \cup po;[L] \\ & \cup po;[L];coi \end{aligned}$$

Relation *ordered-before* (ob) is a transitive closure of obs, aob, dob, and bob relations.

$$ob \triangleq (obs\cup dob\cup aob\cup bob)^+$$

Consistency Constraints An well-formed ARMv8 execution X is consistent when:

- $(X.po|_{loc}\cup X.ca\cup X.rf)$ is acyclic. (internal)
- X.ob is irreflexive. (external)
- $X.rmw\cap(X.fre;X.coe)=\emptyset$ (atomicity)

B Arm to x86 Mapping

We get Arm to x86 mapping by combining Arm to IR and IR to x86 mappings discussed in fig. 18a and fig. 18b respectively.

B.1 Arm to IR

Following the mapping scheme in fig. 18a an Arm load access is mapped to a load along with a trailing Frm fence. An Arm store is mapped to a store in IR and an st_L is mapped to an IR store with leading Fww and trailing Fsc fences. Any `RMW` in Arm is mapped to an `RMW` in IR. We map Arm full fence `DMBFF` to Fsc in IR. The weaker Arm fences generate no fence in the IR.

B.2 IR to x86

Following the mapping scheme in fig. 18b the load, store, `RMW`, and Fsc primitives in the IR are mapped to x86 load, store, `RMW`, and full fence accesses. The other fences in the IR generate no x86 access.

B.2.1 Arm to IR to x86. In fig. 18c we combine the mapping schemes of fig. 18a and fig. 18b. The resulting mappings are also precise.

Arm	IR
$ld/ld_Q/ld_A$	ld_{NA};Frm
st	st_{NA}
st_L	Fww;st_{NA};Fsc
$RMW/RMW_{(A\|L\|AL)}$	RMW_{SC}
DMBFF	Fsc
DMBLD/ISB	skip
DMBST	Fww

(a) Arm to IR

IR	x86
ld_{NA}	ld
st_{NA}	st
RMW_{SC}	RMW
Fsc	MFENCE
Frm/Fww	skip

(b) IR to x86

Arm		IR		x86
$ld/ld_Q/ld_A$	→	ld_{NA};Frm	→	ld
st	→	st_{NA}	→	st
st_L	→	Fww;st_{NA};Fsc	→	st;MFENCE
$RMW/RMW_{(A\|L\|AL)}$	→	RMW_{SC}	→	RMW
DMBFF	→	Fsc	→	MFENCE

(c) Arm to IR to x86

Figure 18. Verified Mappings from Arm to x86 through IR.

The IR to x86 mapping introduce no fence for the memory access mappings. So the mapping scheme is precise. We show that the Arm to IR mapping scheme is also precise.

B.3 Examples for precise fences in Arm to IR mapping scheme

We explain the leading and/or trailing fences introduced along with the memory access in the mapping scheme.

Fww. Consider the following example for the Fww fence in st_L mapping:

$$\begin{array}{l} X=Y=0; \\ X=1; \\ Y_L=1; \end{array} \left\| \begin{array}{l} a=Y; \\ \texttt{DMBFF}; \\ b=Y; \end{array} \right. \quad \rightarrow \quad \begin{array}{l} X=Y=0; \\ X_{NA}=1; \\ \text{Fww}; \\ Y_{NA}=1; \\ \text{Fsc}; \end{array} \left\| \begin{array}{l} a=Y_{NA}; \\ \text{Frm}; \\ \texttt{DMBFF}; \\ b=Y_{NA}; \end{array} \right.$$

The source Arm program disallows outcome $a=1,b=0$. The target IR program disallows the outcome. Without the intermediate Fww fence the outcome would be allowed. Hence the leading Fww fence is essential for st_L mapping.

Fsc. Consider the following example for the Fsc fence in st_L mapping:

$$\begin{array}{l} X=Y=0; \\ X_L=1; \\ a=Y_A; \end{array} \left\| \begin{array}{l} Y_L=1; \\ b=X_A; \end{array} \right. \quad \rightarrow \quad \begin{array}{l} X=Y=0; \\ \text{Fww}; \\ X_{NA}=1; \\ \text{Fsc}; \\ a=Y_{NA}; \\ \text{Frm}; \end{array} \left\| \begin{array}{l} \text{Fww}; \\ Y_{NA}=1; \\ \text{Fsc}; \\ a=X_{NA}; \\ \text{Frm}; \end{array} \right.$$

The source Arm program disallows outcome $a=0,b=0$. The target IR program disallows the outcome. Without the intermediate Fsc fences the outcome would be allowed. Hence the trailing Fsc fence is essential for st_L mapping.

Frm. Consider the following example for the Frm fence in ld mapping:

$$\begin{array}{l} X=Y=0; \\ a=X; \\ Y=a*0+1; \end{array} \left\| \begin{array}{l} b=Y; \\ X=b*0+1; \end{array} \right. \quad \rightarrow \quad \begin{array}{l} X=Y=0; \\ a=X_{NA}; \\ \text{Frm}; \\ Y_{NA}=a*0+1; \end{array} \left\| \begin{array}{l} b=Y_{NA}; \\ \text{Frm}; \\ X_{NA}=b*0+1; \end{array} \right.$$

The source program disallows $a=b=1$ outcome in Arm. To disallow the same in the target program in the IR the Frm is required in Arm to IR mapping.

Note that Arm (version 8) does not remove false dependency [55]. In IR we eliminate false dependence as follows. In that case the fences preserve the required orders for correctness.

$$\begin{array}{l} X=Y=0; \\ a=X_{NA}; \\ \text{Frm}; \\ Y_{NA}=a*0+1; \end{array} \left\| \begin{array}{l} b=Y_{NA}; \\ \text{Frm}; \\ X_{NA}=b*0+1; \end{array} \right. \quad \rightarrow \quad \begin{array}{l} X=Y=0; \\ a=X_{NA}; \\ \text{Frm}; \\ Y_{NA}=1; \end{array} \left\| \begin{array}{l} b=Y_{NA}; \\ \text{Frm}; \\ X_{NA}=1; \end{array} \right.$$

C Counter-examples of Unsafe Reorderings

In fig. 11a we display safe reorderings on LIMM. For the safe reorderings, we provide proofs.

We provide counter-examples for some of the unsafe reorderings.

RM-fence. An Frm *cannot* reorder with a preceding read instruction. Consider the following example:

$$\begin{array}{c} X=Y=0 \\ \begin{array}{l} a=X_{\text{NA}}; \\ \mathsf{Frm}; \\ Y_{\text{NA}}=1; \end{array} \Bigg\| \begin{array}{l} b=Y_{\text{NA}}; \\ X_{\text{NA}}=b; \end{array} \end{array} \quad \rightarrow \quad \begin{array}{c} X=Y=0 \\ \begin{array}{l} \mathsf{Frm} \\ a=X_{\text{NA}}; \\ Y_{\text{NA}}=1; \end{array} \Bigg\| \begin{array}{l} b=Y_{\text{NA}}; \\ X_{\text{NA}}=b; \end{array} \end{array}$$

The target program allows the output $a=b=1$, while the source program disallows this.

WW-fence. An Fww *cannot* reorder with a preceding write instruction. Consider the following example:

$$\begin{array}{c} X=Y=0 \\ \begin{array}{l} X_{\text{NA}}=1; \\ \mathsf{Fww}; \\ Y_{\text{NA}}=2; \end{array} \Bigg\| \begin{array}{l} a=Y_{\text{NA}}; \\ \mathtt{DMBFF}; \\ b=X_{\text{NA}}; \end{array} \end{array} \quad \rightarrow \quad \begin{array}{c} X=Y=0 \\ \begin{array}{l} \mathsf{Fww}; X_{\text{NA}}=1; \\ Y_{\text{NA}}=2; \end{array} \Bigg\| \begin{array}{l} a=Y_{\text{NA}}; \\ \mathtt{DMBFF}; \\ b=X_{\text{NA}}; \end{array} \end{array}$$

The target program allows the output $a=2,b=0$, which the source does not.

RMW - RMW. It is *unsafe* to reorder two RMW operations, even if they access different locations. Consider the following program.

$$\begin{array}{c} X=Y=0; \\ \begin{array}{l} \mathtt{RMW}_{\text{SC}}(X,0,1) \\ \mathtt{RMW}_{\text{SC}}(Y,0,1) \end{array} \Bigg\| \begin{array}{l} a=Y_{\text{NA}} \\ \mathsf{Frm} \\ b=X_{\text{NA}}; \end{array} \end{array} \quad \rightarrow \quad \begin{array}{c} X=Y=0; \\ \begin{array}{l} \mathtt{RMW}_{\text{SC}}(Y,0,1) \\ \mathtt{RMW}_{\text{SC}}(X,0,1) \end{array} \Bigg\| \begin{array}{l} a=Y_{\text{NA}} \\ \mathsf{Frm} \\ b=X_{\text{NA}}; \end{array} \end{array}$$

Outcome $a=1,b=0$ is disallowed in the source program but allowed in the target program after the reordering of the $\mathtt{RMW}_{\text{SC}}$ accesses.