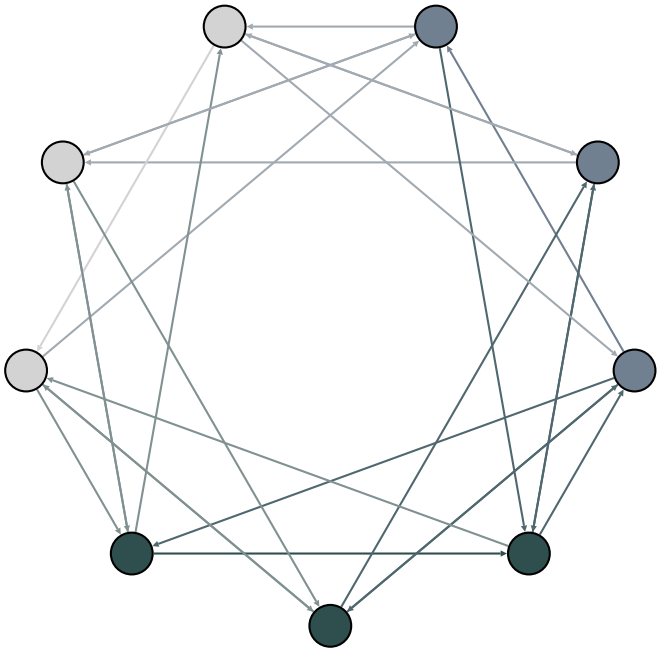
Fencing off unwanted behavior: Improving and evaluating the Fency static analysis tool

Master's Thesis



Pieter van den Ham

Fencing off unwanted behavior: Improving and evaluating the Fency static analysis tool

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Pieter van den Ham
born in Beverwijk, the Netherlands

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: A visualization of a Memory Pair Graph.

Fencing off unwanted behavior: Improving and evaluating the Fency static analysis tool

Author: Pieter van den Ham
Student id: 5173477

Abstract

Computer architectures with weak memory models, such as ARMv8 and ARMv7, allow memory accesses to be reordered in many situations. Therefore, weak memory models may cause a program to exhibit more behavior than a strong memory model, such as x86. Fency is a static analysis tool that inserts memory fences to ensure that a program exhibits the same behavior when run on a weaker memory model. However, Fency lacks important features such as function call support, does not use LLVM's alias analysis algorithms, and inserts too many fences when targeting ARMv8.

We expand Fency with a new dependency tracking analysis, integrate it with LLVM's alias analysis infrastructure, and improve its usability. We show that while the new alias analysis fixes a vital soundness issue, it does not reduce the number of fences Fency inserts. Additionally, our evaluation of the dependency tracking analysis shows that it can eliminate some redundant fences. Finally, we run Fency on larger C/C++ programs, which we made possible by reimplementing Fency as a module pass.

Thesis Committee:

Chair: Prof. dr. K. G. Langendoen, Faculty EEMCS, TU Delft
Committee Member: Dr. S. S. Chakraborty, Faculty EEMCS, TU Delft
Committee Member: D. G. Sprokholt MSc, Faculty EEMCS, TU Delft

Preface

This master's thesis marks the end of over a year of research. It has been quite a journey — one with long winding roads, hairpin turns, dead-ends, and a steep incline. Even though my interest in compilers helped me overcome some of the more minor hurdles, completing this journey would have been impossible without the support of my supervisors, my girlfriend, my friends, and my family.

First and foremost, I want to thank my supervisors, Soham Chakraborty and Dennis Sprokholt, for devoting their time to teaching me an incredibly complex topic. They were always available for questions, gave excellent suggestions, and helped me shape the direction of my thesis. They have shown an incredible amount of patience, understanding, and empathy. I could not have gotten better supervision during this journey. For this, I am forever grateful.

I want to thank Jens and Chris for proofreading my thesis. Their suggestions ensured that my ramblings turned into something somewhat comprehensible. Additionally, the many (beach) volleyball sessions with Jens definitely helped with decompressing after staring at assembly code all day. Thank you.

I want to thank my girlfriend, parents, brother, and sister for supporting me in this effort despite not understanding what I was precisely working on due to my failure to explain it adequately. It is because of their encouragement and motivational speeches that I have finished the thesis by now.

Finally, I want to thank you, the reader, for reading this thesis. I hope you will find its content valuable and educational.

Pieter van den Ham
Delft, the Netherlands
November 30, 2022

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer architectures with *weak memory models* are becoming increasingly prevalent. Weak memory models allow optimizations that reduce the latency of memory accesses [1]. Well-known computer architectures that employ a weak memory model are ARMv8 (AArch64) [2], ARMv7 [3], and Power [4].

The x86 architecture has a stronger memory model than ARMv8 [5]. Consequently, a program initially written for x86 can unexpectedly exhibit more behavior when run on ARMv8. This behavior, when unaccounted for, can lead to subtle bugs that may manifest only under specific circumstances.

1.1 Weak memory models

Traditionally, the behavior of concurrent programs is explained by modeling the execution as a sequence of interleaving threads. However, computer architectures with weak memory models exhibit behavior that cannot be explained by simple thread interleaving.

Consider the program in Figure 1.1. This program uses two threads T_1 and T_2 to write a value to *shared variables* X and Y, after which each thread attempts to read the other thread's shared variable. Thus, T_1 reads Y into local register r_1 and T_2 reads X into local register r_2. All shared variables and registers are zero-initialized.

(a) Pseudocode

T_1	T_2
$X \leftarrow 1$	$Y \leftarrow 1$
$r_1 \leftarrow Y$	$r_2 \leftarrow X$

(b) Equivalent x86 assembly code

T_1	T_2
`mov [X], 1`	`mov [Y], 1`
`mov r1, [Y]`	`mov r2, [X]`

Figure 1.1: A simple concurrent program with shared variables

The author of this program might reasonably expect three possible outcomes when running it. Figure 1.2 shows three possible executions of this example program, resulting in three distinct outcomes.

(a) Interleaved

$X \leftarrow 1$
$Y \leftarrow 1$
$r_1 \leftarrow Y$
$r_2 \leftarrow X$

Results in: $r_1 = 1 \wedge r_2 = 1$

(b) T_2 runs before T_1

$Y \leftarrow 1$
$r_2 \leftarrow X$
$X \leftarrow 1$
$r_1 \leftarrow Y$

Results in: $r_1 = 1 \wedge r_2 = 0$

(c) T_1 runs before T_2

$X \leftarrow 1$
$r_1 \leftarrow Y$
$Y \leftarrow 1$
$r_2 \leftarrow X$

Results in: $r_1 = 0 \wedge r_2 = 1$

Figure 1.2: Three possible executions of the simple concurrent program and their corresponding outcomes

- $r_1 = 1 \wedge r_2 = 1$ (Figure 1.2a). This is the outcome we can expect whenever the instructions of each thread are interleaved.
- $r_1 = 1 \wedge r_2 = 0$ (Figure 1.2b). This happens whenever T_2 runs to completion before T_1.
- $r_1 = 0 \wedge r_2 = 1$ (Figure 1.2c). This happens whenever T_1 runs to completion before T_2.

Most importantly, the author of this program may assume $r_1 = 0 \wedge r_2 = 0$ to be an impossible outcome. This is a reasonable assumption because this outcome would imply a happens-before cycle. The author expects $X \leftarrow 1$ to happen before $r_1 \leftarrow Y$. Since $r_1 = 0$, $r_1 \leftarrow Y$ must have happened before $Y \leftarrow 1$. However, $r_2 = 0$ at the end of the execution, so $r_2 \leftarrow X$ must have happened before $X \leftarrow 1$. This is impossible because $Y \leftarrow 1$ must have happened before $r_2 \leftarrow X$, which creates a cycle. Figure 1.3 visualizes this cycle.

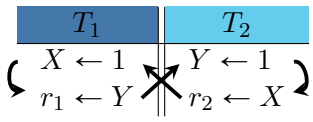


Figure 1.3: Execution order with happens-before cycle for $r_1 = 0 \wedge r_2 = 0$

In reality, there are multiple reasons why outcome $r_1 = 0 \wedge r_2 = 0$ can happen when a modern processor runs the program. Two examples are:

- **Store buffering and load buffering.** An architecture might store writes to a CPU-local buffer before flushing the entire buffer to RAM. A CPU with store buffers is more performant since writing values to a store buffer is orders of magnitude faster than writing values to RAM immediately. If a processor has this feature *and* T_1 and T_2 are scheduled to run on different CPU cores, write instructions $X \leftarrow 1$ and $Y \leftarrow 1$ might end up in a store buffer which prevent T_2 and T_1 from reading the updated values. Load buffering is analogous to store buffering. A CPU that has load-buffering reads from a CPU-local buffer before accessing RAM.
- **Instruction reordering.** An architecture might allow the CPU to execute multiple read-/write instructions in parallel, execute some instructions speculatively or even swap instructions altogether to avoid stalling the instruction pipeline while waiting for data to return from RAM. If the processor that we run our example program on allows out-of-order execution, it might decide to execute $r_1 \leftarrow Y$ and $r_2 \leftarrow X$ before $X \leftarrow 1$ and $Y \leftarrow 1$.

A computer architecture may implement one or more of these kinds of optimizations, making it increasingly harder to reason about concurrent memory accesses. Therefore, many hardware designers publish a memory model.

> **Definition:** A memory model (also called a consistency model) is a set of guarantees that a platform provides programmers when working with memory shared between different threads or processor cores. A memory model defines when two memory accesses may be *reordered*. When two memory accesses are reordered, they appear to have been executed in a different order than originally defined by the program (the second memory access "overtakes" the first memory access).
>
> We consider four types of reordering: Store-Store, Store-Load, Load-Store, and Load-Load. A memory model may allow none, some, or all of these types of reordering. For example, a load access can overtake an earlier store access in a memory model that allows Store-Load reordering.
>
> Memory accesses are never allowed to overtake *fence* instructions. Thus, a program-

mer or compiler can prevent instruction reordering by inserting a fence between the two memory accesses. A fence can order one or more of the types of reordering.

Sequential Consistency. One example of a strong memory model is Sequential Consistency (SC). SC never allows any instruction reordering, and it enforces a total order on the executed instructions. Intuitively, this means a program run on the SC memory model appears as if it had been run on a single CPU.

x86-TSO. The memory model that Intel has implemented for its x86 processors, x86-TSO, is a *weaker* memory model than SC: x86-TSO allows Store-Load reordering in some instances. This means that an earlier store can be reordered with a later load if the load does not depend on the store [6].

To improve the performance of the x86 architecture, x86-TSO is a slightly weakened version of SC. Even though x86-TSO is weaker than SC, it is still considered a strong memory model.

ARMv8. ARMv8 is the first weak memory model that we will consider. The ARMv8 architecture allows all four types of memory reordering [2].

ARMv7. ARMv7 is an even weaker architecture than ARMv8. For example, besides allowing all four types of reordering, it also allows different cores to observe the same writes in a different order.

1.2 M-K robustness

x86-TSO does not allow Load-Store reordering, but ARMv8 does. This difference in memory models can result in surprising behavior. Consider the simple concurrent program in Figure 1.4a.

T_1	T_2
$r_1 \leftarrow X$	$r_2 \leftarrow Y$
$Y \leftarrow 1$	$X \leftarrow 1$

(a)

T_1	T_2
$r_1 \leftarrow X$	$r_2 \leftarrow Y$
F_{LS}	F_{LS}
$Y \leftarrow 1$	$X \leftarrow 1$

(b)

Figure 1.4: Two concurrent programs. (a) without Load-Store fences, (b) with Load-Store fences.

Two threads, T_1 and T_2, attempt to read from and subsequently write to two shared variables X and Y.

Assuming that both X and Y are initialized to 0, after running the program on an x86 system, r_1 and r_2 can equal 0 or 1, but they can never *both* equal 1. However, if we run this program on ARMv8 then, due to ARMv8's Load-Store reordering, $r_1 = 1 \wedge r_2 = 1$ *is* a possible outcome: the program exhibits more behavior (i.e., more possible outcomes) when run on ARMv8.

To ensure that the program does not exhibit more behavior on ARMv8, we have to prevent the Load-Store reordering from happening. The load-store fences ensure $r_1 = 1 \wedge r_2 = 1$ cannot happen on ARMv8.

Several existing techniques can help identify these subtle differences in program behavior. Chakraborty[5] published one such technique: M-K robustness analysis.

Definition: A program is M-K robust when running it on weaker memory model K yields the same output as running it on stronger memory model M. The program in Figure 1.4a is not x86-ARMv8 robust since running it on ARMv8 can result in different output than

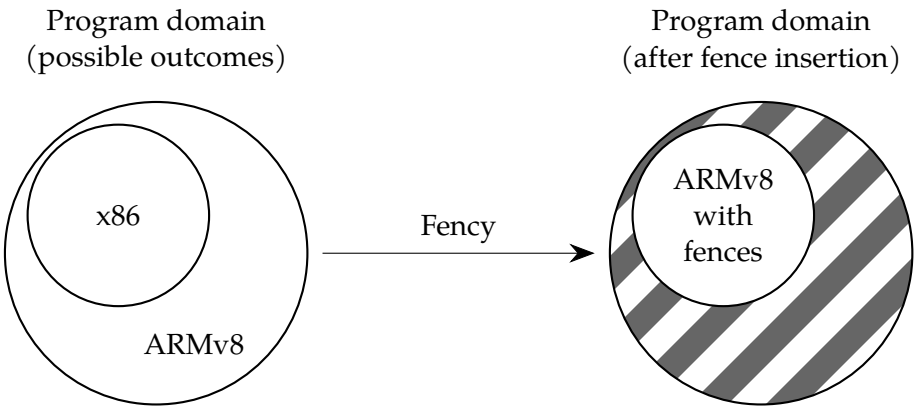


Figure 1.5: A schematic overview of how Fency enforces x86-ARMv8 robustness. ARMv8 allows more behavior, and thus more potential outcomes. Fency restricts the behavior to match the behavior of x86.

> running it on x86. The program in Figure 1.4b *is* x86-ARMv8 robust since it will always yield the same output on both architectures.

Chakraborty built a prototype that implements M-K robustness analysis. This prototype was dubbed Fency, a static analysis tool built on top of the LLVM compiler framework that uses M-K robustness analysis to find subtle differences in program behavior. It automatically inserts the appropriate fences, so the program behaves as if running on memory model M. See Figure 1.5 for a schematic overview of how Fency impacts the potential outcomes of a program after inserting fences.

From now on, we will refer to the prototype that Chakraborty built as "Fency v1".

1.3 Fency v1: issues and limitations

1.3.1 Architectural limitations

In the LLVM framework, there are two types of compiler passes:

- Function passes, which operate on a single function at a time
- Module passes, which operate on an entire module. A module is analogous to a translation unit in C or C++ [7]. It may contain the *definition* of one or more functions, and it may include the *declaration* of functions defined in other translation units.

Fency v1 is implemented as a function pass. This results in two significant limitations.

The `main` function always has to be defined last

Fency v1 needs to access the analysis results of earlier functions. However, this is not possible when using a function pass since a function pass generally should not and cannot access information about other functions. As a band-aid fix for this issue, Fency v1 uses global state to access the analysis results of previously defined functions, which implicitly causes it to rely on the function definition order.

Fency v1 does not implement the fence insertion algorithm as described in [5]

In LLVM, it is only possible to insert instructions into the function on which the function pass is currently operating. Fency v1 only needs to insert fences between memory access pairs that are part of a cycle in the Memory Pair Graph [5]. However, because Fency v1 has

to wait until it encounters the `main` function before it can the Memory Pair Graph, it cannot do so. Instead, Fency v1 currently inserts fences between **all** non-robust pairs, regardless of whether they are part of a cycle. As a result, Fency v1 may insert more fences in some cases.

To fix these two issues, Fency v1 needs to be reimplemented as a module pass.

1.3.2 Dependency analysis

In some cases, the memory barriers that Fency v1 inserts are redundant because of certain ordering guarantees by the architecture. Consider the following ARMv8 assembly code snippet:

```
1 ldr     w8, [x8]  ; load data from location [x8] into register w8
2 dmb ishld         ; fence
3 str     w8, [x9]  ; store data from register w8 into location [x9]
```

In this short example, the second store instruction depends on the data loaded by the first load instruction (via the `w8` register). This is significant because the ARMv8 architecture never reorders two instructions with a data dependency [2]. Unfortunately, Fency v1 lacks this knowledge about dependencies, and thus it still inserts a fence between these two instructions. There are approximately 8 similar situations where dependency analysis could lead to fewer fences.

1.3.3 Alias analysis

How many fences Fency v1 inserts largely depends on the accuracy of the instruction alias analysis implementation. An instruction alias analysis algorithm attempts to determine whether two instructions access the same memory location. The accuracy of this algorithm directly impacts the underlying data structure that Fency v1 uses for its M-K robustness analysis. The alias analysis implementation is also essential in ensuring that Fency v1 is sound.

A sound alias analysis implementation is conservative: it must default to the most conservative result when it does not know whether an instruction points to a particular memory location. For example, when checking whether two instructions *potentially* access the same location, an alias analysis algorithm must default to "yes" if it cannot prove that the two instructions *never* alias. This may be the case in the following C++ snippet:

```
1 void threadA(void *arg) {
2     std::atomic<int> *x = static_cast<std::atomic<int>*>(arg);
3     x->store(42);
4 }
```

In this case, alias analysis may not be able to determine where the `arg` variable originates from. Therefore, when asked whether `x->store(42)` may access the same memory location as another memory access, the alias analysis algorithm must conservatively decide that this may be the case.

Similarly, when asked whether two instructions *always* access the same location, a sound alias analysis algorithm must report "no" if the instructions only sometimes access the same location.

Unfortunately, Fency's current alias analysis implementation is unsound: it is not conservative when it cannot find the exact memory location that an instruction accesses. Besides fixing this soundness issue, a more accurate alias analysis implementation might decrease the size of Fency's underlying data structure and thereby improve its runtime.

1.3.4 Function call support

As Fency v1 is still a prototype implementation, it lacks support for function calls. Consequently, every function call must be inlined in the source code. Not only is it impossible

to inline every function call for some programs, such as recursive programs, but it is also a significant barrier when attempting to use Fency v1 on larger programs.

Consider the following C++ snippet:

```
1  std::atomic<int> X;
2  std::atomic<int> Y;
3
4  void threadA(void *arg) {
5      int r = f();
6      g(r);
7  }
8
9  int f() {
10     return X.load();
11 }
12
13 void g(int a) {
14     Y.store(a);
15 }
```

Currently, Fency v1 will ignore the function calls to `f` and `g`, even though those functions contain memory accesses to shared variables. Fency v1 must include the called functions in its control flow graph to properly support function calls.

Including multiple functions in the control flow graph leads to an issue, however: Fency v1 will need to support inter-procedural reachability analysis since it needs to check whether two instructions can reach each other.

1.4 Research question

We will attempt to answer the following research question:

How does dependency-ordered before (DOB) analysis and alias analysis affect Fency's ability to insert fences?

1.5 Contributions

To answer the research question, we will reimplement Fency. This new version, Fency v2, fixes the issues and limitations described earlier, implements a new DOB analysis pass and implements a new alias analysis algorithm. To summarize, this thesis makes the following contributions:

- We fix some limitations by reimplementing Fency as a module pass
- We reduce the number of inserted fences for ARMv8 by implementing dependency-ordered-before analysis
- We improve Fency's correctness and runtime by replacing the old alias analysis implementation with a more accurate implementation
  - The new implementation fixes an important soundness issue in Fency v1
  - The new implementation reduces the size of the underlying data structure for some programs
- We improve Fency's usability by implementing function call support

- This enables us to run Fency on larger C and C++ programs (the CDS Checker benchmarks [8])

Chapter 2 provides some background knowledge on memory models, M-K robustness analysis, the LLVM compiler framework, and Fency's inner workings. Chapter 3 describes the improvements that have been made to Fency. Chapter 4 evaluates these improvements, Chapter 5 discusses related work and Chapter 6 contains a conclusion and final remarks.

Chapter 2

Background

Section 2.1 introduces the theory that underpins the Fency static analysis tool: axiomatic memory models. Section 2.2 then formalizes M-K robustness. Finally, Section 2.3 discusses how M-K robustness is implemented in Fency.

2.1 Axiomatic memory models

The concurrency semantics of x86 and ARM are formalized by *axiomatic memory models*[9, 10, 11, 12, 13, 14].

We can build an *execution graph* from the program in Figure 2.1a and the outcome $r_1 = 0 \wedge r_2 = 0$ by noticing that the reads must have obtained their values from the initializing writes (all shared variables are implicitly initialized to zero). The nodes in this graph represent a particular memory event, such as a Read or a Write. The edges define some relation between the memory events. Note how the graph contains a cycle: $\mathbf{W}(X,1) \xrightarrow{\text{po}} \mathbf{R}(Y,0) \xrightarrow{\text{fr}} \mathbf{W}(Y,1) \xrightarrow{\text{po}} \mathbf{R}(X,0) \xrightarrow{\text{fr}} \mathbf{W}(X,1)$ As we will see later, SC does not allow any cycles in its executions; thus this particular execution is not allowed by SC.

2.1.1 Motivation

A formal specification for a hardware memory model allows engineers to verify whether a particular execution of a program is possible on physical hardware. Before these formal models were developed, hardware designers often published informal models of their architectures. For example, Intel originally used litmus tests, such as the one in Figure 2.1a, to describe to x86 memory model. However, specifying an architecture in litmus tests was

(a) Pseudocode

T_1	T_2
$X \leftarrow 1$	$Y \leftarrow 1$
$r_1 \leftarrow Y$	$r_2 \leftarrow X$
Outcome: $r_1 = 0 \wedge r_2 = 0$	

(b) Execution graph

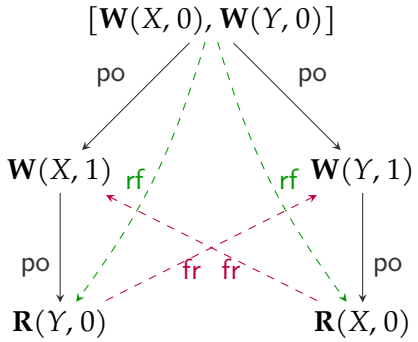


Figure 2.1: Store buffering litmus test (left) and one possible execution (right)

found to be a non-optimal solution since the litmus tests are sometimes incomplete, ambiguous, or even inaccurate [6, 15].

2.1.2 Definitions

Events

Memory events capture the memory operations that happen during a particular execution. Each event has a label, such as **R** for read events and **W** for write events. Some events are parameterized with an associated memory location and a value.

For example, $\mathbf{R}(X, 0)$ denotes a read event where the value 0 was read from location X. $\mathbf{W}(X, 1)$ denotes a write event where the value 1 was written to location X. $\mathbf{F_{full}}$ describes a full fence.

Relations

A relation captures certain information *between* events. Edges in the execution graph represent relations.

For example, the po relation captures the syntactic order of two memory events. In other words, $\mathbf{W} \xrightarrow{\mathsf{po}} \mathbf{R}$ captures the order in which the two events were defined in the original program source.

To describe an axiomatic model, we use several properties that we can derive from a binary relation. Let $\mathsf{R} \subseteq \mathbf{E} \times \mathbf{E}$ describe a binary relation R:

- $\mathsf{dom}(\mathsf{R})$ refers to its domain and $\mathsf{codom}(\mathsf{R})$ refers to its range. In other words, $\mathsf{dom}(\mathsf{R}) \subseteq \mathbf{E}$ is the "left-hand side" or the origin of the arrow, and $\mathsf{codom}(\mathsf{R}) \subseteq \mathbf{E}$ is the "right-hand side" or the target of the arrow.
- R^{-1} is the *inverse* closure of R, i.e. the set of all pairs $(\mathbf{E}_2, \mathbf{E}_1)$ such that $(\mathbf{E}_1, \mathbf{E}_2)$ is an element of R.
- $\mathsf{R}^?$ is the *reflexive* closure of R. Formally, $\mathsf{R}^? = \mathsf{R} \cup (\mathbf{e}, \mathbf{e}) : \mathbf{e} \in [\mathbf{E}]$.
- R^+ is the *transitive* closure of R. To create a transitive relation from R, we must add $(\mathbf{a}, \mathbf{c})$ to R if R contains $(\mathbf{a}, \mathbf{b})$ and $(\mathbf{b}, \mathbf{c})$, where $\mathbf{a}, \mathbf{b}, \mathbf{c} \in [\mathbf{E}]$.
- R^* is the *reflexive-transitive* closure of R. The reflexive-transitive closure must satisfy $\mathsf{R} \subseteq \mathsf{R}^*$ where R^* is both reflexive and transitive.
- R_ℓ captures the subset of R related pairs that access the same memory location. In other words, if $\mathbf{W}(X, 1) \xrightarrow{\mathsf{R}} \mathbf{R}(X, 0)$, then $(\mathbf{W}(X, 1), \mathbf{R}(X, 0)) \in \mathsf{R}_\ell$, because they both access the same location X.
- $\mathsf{R}_{\neq\ell}$ is the subset of R where each pair accesses *different* memory locations.
- $\mathsf{imm}(\mathsf{R})$ defines the immediate R relation.
- $[\mathbf{E}]$ is the identity relation on set $\mathbf{E}$.
- $\mathsf{R}; \mathsf{P}$ is the composition of relations R and relation P. For example, composite relation $[\mathbf{W}]; \mathsf{po}_\ell; [\mathbf{W}]$ captures all same-location po-related write events.

We define two more relations aside from po:

- rf (read-from) relates a write event to a read event. It describes how the read event obtained its value. In Figure 2.1b, $\mathbf{W}(X, 0) \xrightarrow{\mathsf{rf}} \mathbf{R}(X, 0)$ describes how this particular read event obtained its value.

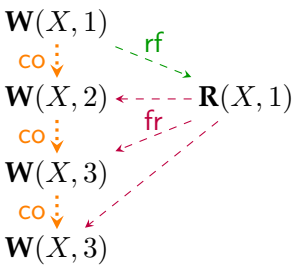


Figure 2.2: An illustration of the rf, co and fr relations

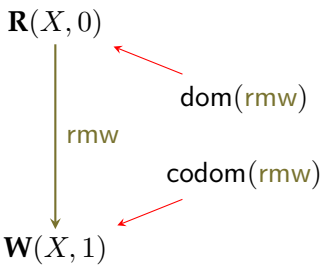


Figure 2.3: Domain and codomain of a binary relation.

- co (coherence-order) relates two same-location write events, such as $\mathbf{W}(X,1) \xrightarrow{\mathsf{co}} \mathbf{W}(X,2)$.

From these two relations we can derive several other commonly-used relations:

- fr (from-read) relates a read event $\mathbf{R}(X,i)$ to a same-location write event $\mathbf{W}_{after}(X,j)$ if $\mathbf{W}(X,i) \xrightarrow{\mathsf{rf}} \mathbf{R}(X,i)$ and $\mathbf{W}(X,i) \xrightarrow{\mathsf{co}} \mathbf{W}_{after}(X,j)$. In this case, $\mathbf{W}_{after}$ hit the memory *after* the write from which $\mathbf{R}$ obtained its value. Formally, $\mathsf{fr} \triangleq \mathsf{rf}^{-1};\mathsf{co}$. Figure 2.2 illustrates the rf, fr and co relations.
- rmw (read-modify-write[1]) relates a read event to a po-immediate same-location write event: $\mathsf{rmw} \triangleq \mathsf{imm}(\mathsf{po}) \cap ([\mathbf{R}] \times [\mathbf{W}])_\ell$.

It is helpful to differentiate normal loads and stores from rmw loads and stores. Therefore, we define the set of loads and the set of stores as:

$$\mathbf{Ld} \triangleq \mathbf{R} \backslash \mathsf{dom}(\mathsf{rmw}) \qquad \mathbf{St} \triangleq \mathbf{W} \backslash \mathsf{codom}(\mathsf{rmw})$$

Note: dom(rmw) and codom(rmw) refer to the set of rmw related reads and writes respectively (see Figure 2.3).

A read-modify-update operation can fail. A failed RMW generates an **Ld** event.

We also differentiate between external (inter-thread) and internal (intra-thread) relations for rf, fr and co:

$$\begin{array}{lll} \mathsf{rfe} \triangleq \mathsf{rf} \backslash \mathsf{po} & \mathsf{coe} \triangleq \mathsf{co} \backslash \mathsf{po} & \mathsf{fre} \triangleq \mathsf{fr} \backslash \mathsf{po} \\ \mathsf{rfi} \triangleq \mathsf{rf} \cap \mathsf{po} & \mathsf{coi} \triangleq \mathsf{co} \cap \mathsf{po} & \mathsf{fri} \triangleq \mathsf{fr} \cap \mathsf{po} \end{array}$$

We can now define eco (extended-coherence-order) as $\mathsf{eco} \triangleq (\mathsf{rfe} \cup \mathsf{coe} \cup \mathsf{fre})^+$. This relation captures all same-location inter-thread communication.

Axioms

An axiomatic model defines one or more axioms that an execution graph must satisfy to be valid. All the models that Fency uses include at least the *coherence* and *atomicity* axioms.

Coherence. The coherence axiom asserts that there exists an order in which all write events happened (i.e., it enforces a *total order*). This axiom prevents causal loops, such as the cycle in Figure 2.4.

$$\mathsf{acyclic}(\mathsf{po}_\ell \cup \mathsf{rf} \cup \mathsf{co} \cup \mathsf{fr}) \qquad \text{(coherence)}$$

[1] An RMW (also referred to as atomic compare-and-swap) operation atomically compares a value to an expected value and writes a new value if they match. An RMW operation will fail if the read value does not match the expected value.

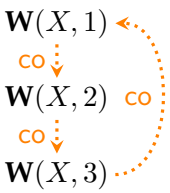


Figure 2.4: This execution is forbidden by the coherence axiom.

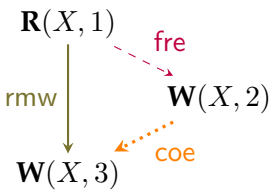


Figure 2.5: This execution is forbidden by the atomicity axiom.

Atomicity. The atomicity axiom asserts that rmw operations are atomic. An execution would violate atomicity if there is an intermediate write between the rmw related read and write events. Figure 2.5 shows an execution that is forbidden by the atomicity axiom.

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

2.1.3 An axiomatic model for Sequential Consistency

The axiomatic model for sequential consistency consists of two axioms:

$$\mathsf{acyclic}(\mathsf{po} \cup \mathsf{rf} \cup \mathsf{co} \cup \mathsf{fr}) \qquad \text{(sc)}$$

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

Since $\mathsf{po}_\ell \subseteq \mathsf{po}$, the SC axiom also encompasses the coherence axiom.

Example

Consider again the store buffering litmus test from Figure 2.1b. We can now see that, because the axiomatic model for Sequential Consistency does not allow $(\mathsf{rf} \cup \mathsf{fr} \cup \mathsf{po})$ cycles, this execution is not allowed.

2.1.4 An axiomatic model for x86-TSO

Relations

For the x86-TSO model we define three additional relations:

- xppo – x86 never reorders two write-write, read-write or read-read events. xppo incorporates this into our model by relating two write events, two read events or a read event to a write event, where both events are also related by po. xppo does not relate a write to a po-subsequent read event since x86 does allow reordering write-reads.

$$\mathsf{xppo} \triangleq ((\mathbf{W} \times \mathbf{W}) \cup (\mathbf{R} \times \mathbf{W}) \cup (\mathbf{R} \times \mathbf{R})) \cap \mathsf{po}$$

- implied – implied encodes that both a rmw and an F act as a full fence in the x86 architecture. dom(rmw) and codom(rmw) refer to the read and respectively the write event related by a rmw.

$$\mathsf{implied} \triangleq \mathsf{po}; [\mathsf{dom}(\mathsf{rmw}) \cup \mathbf{F}] \cup [\mathsf{codom}(\mathsf{rmw}) \cup \mathbf{F}]; \mathsf{po}$$

- xhb – xhb combines the relations that order two memory accesses in x86.

$$\mathsf{xhb} \triangleq \mathsf{xppo} \cup \mathsf{implied} \cup \mathsf{rfe} \cup \mathsf{fr} \cup \mathsf{co}$$

(a) Pseudocode

T_1	T_2
$r_1 \leftarrow X$	$r_2 \leftarrow Y$
$Y \leftarrow 1$	$X \leftarrow 1$

Outcome: $r_1 = 1 \wedge r_2 = 1$

(b) Execution graph

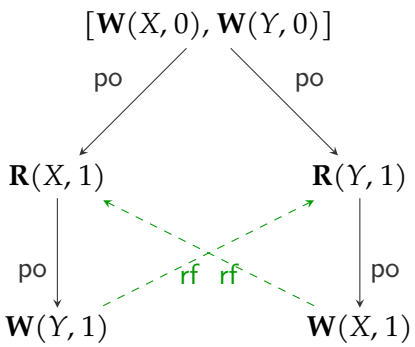


Figure 2.6: One execution of the load buffering litmus test (left) and the corresponding execution graph (right)

Axioms

We can now define the x86-TSO model as follows:

$$\text{acyclic}(\text{po}_\ell \cup \text{rf} \cup \text{fr} \cup \text{co}) \qquad \text{(sc-per-loc)}$$

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

$$\text{acyclic}(\text{xhb}) \qquad \text{(GHB)}$$

Example

Consider the load-buffering litmus test in Figure 2.6a.

Note how $\mathbf{R}(X,1) \overset{\text{po}}{\rightarrow} \mathbf{W}(Y,1) \overset{\text{rf}}{\rightarrow} \mathbf{R}(Y,1) \overset{\text{po}}{\rightarrow} \mathbf{W}(X,1) \overset{\text{rf}}{\rightarrow} \mathbf{R}(X,1)$ creates a cycle. Because $\mathbf{R} \overset{\text{po}}{\rightarrow} \mathbf{W} \subseteq \text{xppo}$ (read accesses are never reordered with subsequent write accesses in x86), we can conclude that this execution is not allowed by x86-TSO due to the GHB axiom.

2.1.5 An axiomatic model for ARMv8

Events

In addition to a Read, Write and Fence event, we define the following events for the ARMv8 model:

- Store-release events $\mathbf{L}(x,v) \subseteq \mathbf{W}$ to model store-release instructions such as `stlr`
- Load-acquire events $\mathbf{A}(x,v) \subseteq \mathbf{R}$ to model load-acquire instructions such as `ldar`
- AcquirePC events $\mathbf{Q}(x,v) \subseteq \mathbf{R}$ to model ARM's `ldapr` instruction
- Store barrier events $\mathbf{F_{ST}}$ for `dmb st` instructions
- Load barrier events $\mathbf{F_{LD}}$ for `dmb ld` instructions

Relations

In the ARMv8 model, we introduce the notion of *dependencies* between memory access events.

- $\text{data} \subseteq \mathbf{R} \times \mathbf{W}$ relates a read event to a write event if there is a data dependency from the read operation to the write operation.

- addr $\subseteq \mathbf{R} \times (\mathbf{R} \cup \mathbf{W})$ relates a read event to a read or write event if there is an address dependency from the read operation to the subsequent read or write operation.
- ctrl $\subseteq \mathbf{R} \times \mathbf{E}$ relates a read operation to a subsequent memory access event if there is a control dependency from the read operation to the subsequent memory access event

aob (atomic-orderded-by) captures how a rmw orders two memory accesses. Two accesses are also ordered when an acquire or acquirePC operation reads its value from the write operation of the rmw.

$$\mathsf{aob} \triangleq \mathsf{rmw} \cup [\mathsf{codom}(\mathsf{rmw})]; \mathsf{rfi}; [\mathbf{A} \cup \mathbf{Q}]$$

dob (dependency-ordered-before) captures the various ways in which a dependency between two instructions causes them to be ordered.

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

bob (barrier-ordered-by) relation captures how a fence orders two instructions.

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

Finally, obs $\subseteq$ eco (observed-by) captures inter-thread communication.
These relations are aggregated in the ordered-before $\mathsf{ob} \triangleq (\mathsf{obs} \cup \mathsf{dob} \cup \mathsf{aob} \cup \mathsf{bob})^+$ relation.

Constraints

The ARMv8 model has the following constraints:

$$\mathsf{acyclic}(\mathsf{po}_\ell \cup \mathsf{rf} \cup \mathsf{fr} \cup \mathsf{co}) \qquad \text{(sc-per-loc)}$$

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

$$\mathsf{irreflexive}(\mathsf{ob}) \qquad \text{(ordered-before)}$$

Example

Consider the execution of a litmus test in Figure 2.7. In this execution, there is a cycle $\mathbf{W}(X[1],1) \xrightarrow{\mathsf{rf}} \mathbf{R}(X[1],1) \xrightarrow{\mathsf{addr}} \mathbf{R}(Y[1],0) \xrightarrow{\mathsf{fr}} \mathbf{W}(Y[1],1) \xrightarrow{\mathsf{rf}} \mathbf{R}(Y[1],1) \xrightarrow{\mathsf{addr}} \mathbf{R}(X[1],0) \xrightarrow{\mathsf{fr}} \mathbf{W}(X[1],1)$. This cycle violates the (ordered-before) constraint of the axiomatic model - and thus this execution is not allowed in ARMv8.

(a) Pseudocode

T_1	T_2	T_3	T_4
$X[1] = 1$	$a = X[1]$ $b = Y[a]$	$c = Y[1]$ $d = X[c]$	$Y[1] = 1$

Outcome: $a = c = 1 \wedge b = d = 0$

(b) Execution graph

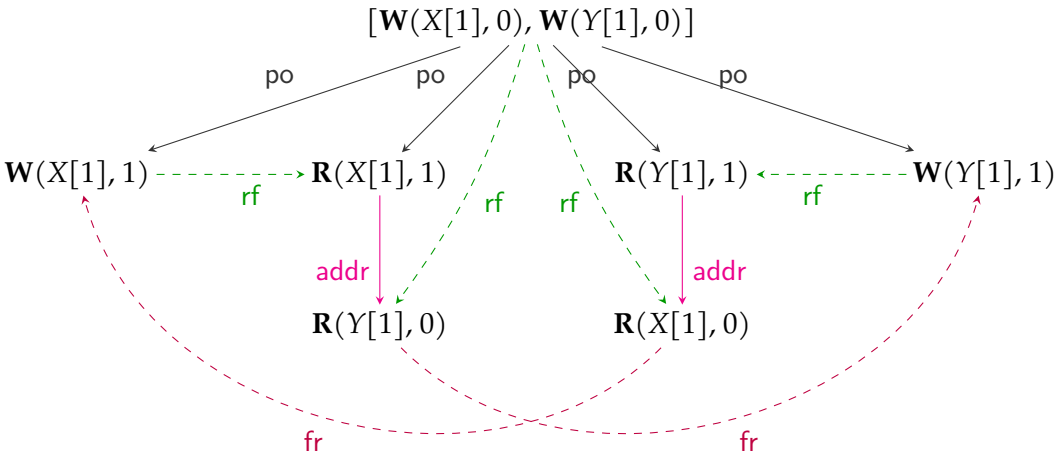


Figure 2.7: One execution of the Independent Reads of Independent Writes (IRIW) litmus test (top) and the corresponding execution graph (bottom)

2.1.6 An axiomatic model for ARMv7

The axiomatic model that Fency uses for ARMv7 can be found in Chakraborty's paper [5]. In ARMv7, memory accesses consist of an *init* and *commit* step [12, 5] Even though the ARMv7 axiomatic model does define dependencies, they are not sufficient to enforce M-K robustness.

2.2 M-K Robustness

This section is based off of research performed by Chakraborty [5]. We will explore how M-K robustness ensures that a program behaves as if it were run on stronger memory model M when run on weaker memory model K.

Robustness cycles

Whenever a program violates M-K robustness then one of its execution graphs will contain a cycle. This cycle must contain a po edge. If it would *not* have a po edge, it would mean that the cycle consists of rfe, fre and coe edges. However, this is impossible since such a cycle would violate coherence (Equation coherence). All models that we are considering satisfy the coherence axiom. Therefore, a cycle that violates M-K robustness always contains po edges.

We denote the po edges that can participate in a robustness cycle as epo (external program-order). Intuitively, epo edges relate two po-related events that "participate in external communication".

$$\mathsf{epo} \triangleq \mathsf{po} \cap (\mathsf{codom}(\mathsf{eco}) \times \mathsf{dom}(\mathsf{eco})) \qquad \text{(external program-order)}$$

M-K robustness is violated when there is a cycle of $(\text{epo}; \text{eco})^+$ edges, where one or more epo edges are not M-K robust. An *M-K robustness condition* determines when an epo edge is M-K robust.

SC-x86 robustness

A program that is SC-x86 robust exhibits SC behavior even when executed on the x86 architecture. To check whether a program is SC-x86 robust, we need to find all $(\text{epo}; \text{eco})^+$ cycles (if any) and look at its epo edges.

For SC-x86 robustness, an epo edge is considered robust when it satisfies the SC-x86 robustness condition. The SC-x86 robustness condition is defined as [5]:

$$\text{xppo} \cup \text{po}_\ell \cup \text{implied}; \text{po}^? \qquad \text{(SC-x86 robustness)}$$

Figure 2.8 illustrates all the possible robust epo edges that can be derived from the condition above. If an epo edge matches one of the pictured cases, then the epo edge is considered SC-x86 robust.

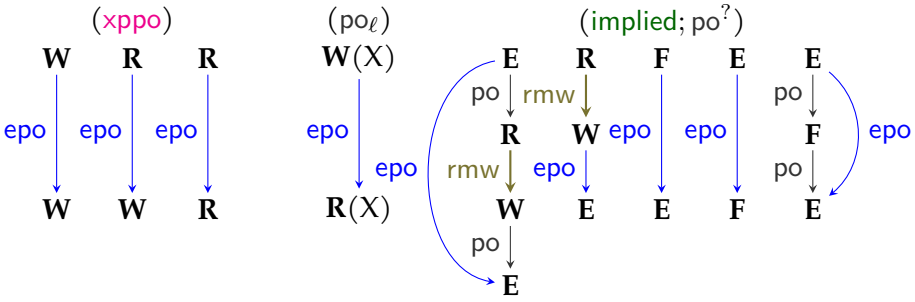


Figure 2.8: All epo edges that are sufficiently ordered according to the SC-x86 robustness condition

Conversely, if the epo edge is not robust, we need to "strengthen" it by inserting a fence. Consider the execution graph of the store buffering litmus test in Figure 2.9.

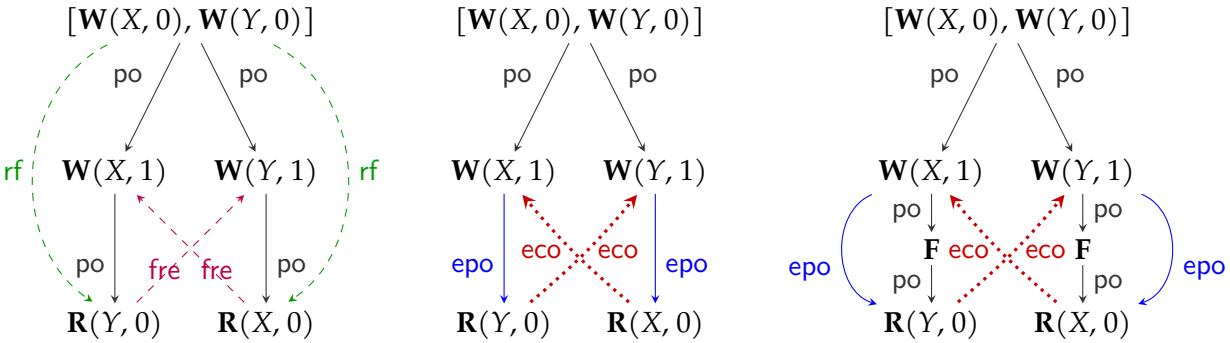


Figure 2.9: An execution graph for the store buffering litmus test (left), its corresponding $(\text{epo}; \text{eco})^+$ cycle (middle), and the strengthened epo edges (right)

The graph on the left depicts the original execution graph. When we replace the fre edges with eco edges, we can see an $(\text{epo}; \text{eco})^+$ cycle emerge. This cycle contains two epo edges:

- $\mathbf{W}(X, 1) \xrightarrow{\text{epo}} \mathbf{R}(Y, 0)$
- $\mathbf{W}(Y, 1) \xrightarrow{\text{epo}} \mathbf{R}(X, 0)$

Neither of these epo edges matches the robustness condition (po_ℓ does not apply because they are not same-location write-read pairs). Therefore, this program is not SC-x86 robust.

Figure 2.9 (right) depicts how inserting a fence strengthens the epo edge. By inserting the fence between the write-read pairs, the epo edges satisfy implied; $\mathsf{po}^?$. The program is now SC-x86 robust.

SC-ARMv8 robustness

The SC-ARMv8 robustness condition can be defined as [5]:

$$\mathsf{po}_\ell \cup (\mathsf{aob} \cup \mathsf{dob} \cup \mathsf{bob})^+ \qquad \text{(SC-ARMv8 robustness)}$$

Similarly to when we had to ensure SC-x86 robustness earlier, we have to insert two fences into the store buffering program in Figure 2.9 to strengthen its epo edges. With the fences inserted, the epo edges now satisfy bob (barrier-ordered-by); thus, the resulting program is also SC-ARMv8 robust.

x86-ARMv8 robustness

The x86-ARMv8 robustness condition is very similar to the SC-ARMv8 robustness condition. However, x86 allows write-read reordering [6]. Therefore, write-read pairs are x86-ARMv8 robust, even though they are not SC-ARMv8 robust.

This results in the following robustness condition for x86-ARMv8 [5]:

$$\mathsf{po}_\ell \cup (\mathsf{aob} \cup \mathsf{dob} \cup \mathsf{bob})^+ \cup \mathsf{wr} \qquad \text{(x86-ARMv8 robustness)}$$

In this equation, wr is a special write-read relation, defined as:

$$\mathsf{wr} \triangleq ([W]; \mathsf{po}_{\neq\ell}; [R]) \backslash (\mathsf{po}; \mathsf{rmw}; \mathsf{po})$$

wr captures $\mathsf{po}_{\neq\ell}$-related write-read pairs and eliminates any write-reads pairs that are also rmw-related.

If we consider the store buffering litmus test in Figure 2.9, we can see that the epo edges satisfy wr. Therefore, the execution is *already* x86-ARMv8 robust, and we do not have to insert any fences.

However, the execution of the load buffering litmus test in Figure 2.6 is *not* x86-ARMv8 robust: none of the epo edges match the x86-ARMv8 robustness condition and are thus not x86-ARMv8 robust. To make the load buffering litmus test x86-ARMv8 robust, we have to strengthen the epo edges by inserting two fences.

M-ARMv7 robustness

Fency supports SC-ARMv7, x86-ARMv7, and ARMv8-ARMv7 robustness. The robustness condition of these robustness configurations, along with some examples, can be found in [5].

2.3 Fency

To integrate M-K robustness checking into a static analysis tool, we have to perform four distinct steps [5].

The first step is collecting and constructing all concurrent control flow graphs. There are multiple approaches that we can take to do this. Fency uses a simple, naive approach where it analyzes the `main` function body for thread entry points.

The second step is constructing the memory pair graph (MPG). To reduce its size, Fency omits pairs from the MPG that do not have a conflicting access.

The third step is finding non-robust memory access pairs. For every pair that is part of a cycle in the memory pair graph, Fency checks whether the pair is ordered according to the robustness condition of the desired M-K robustness.

The final step is enforcing the desired M-K robustness. For every unordered pair that Fency found, it tries to construct an appropriate fence to order the pair.

The following sections will describe this process in more detail.

2.3.1 Helper functions

```
infix fn mustAlias(i: MemoryAccess, j: MemoryAccess) -> bool;
```

`mustAlias` is an infix helper function that returns whether two memory accesses *always* access the same memory location. If `i` and `j` only sometimes access the same location, or if Fency cannot know which locations will be accessed, this function will return `false`.

```
infix fn mayAlias(i: MemoryAccess, j: MemoryAccess) -> bool;
```

`mayAlias` is an infix function that checks whether two memory accesses *potentially* access the same location. If two memory accesses only sometimes access the same variable, this function must return `true`. Similarly, if Fency cannot know which locations will be accessed, this function must return `false`.

```
infix fn canPotentiallyReachWithoutPassingThrough(
    i: MemoryAccess,
    j: MemoryAccess,
    Fences: List<MemoryAccess>
) -> bool;
```

This function tries to find a path from `i` to `j` without passing through any of the memory accesses in `Fences`. This function is conservative: if it cannot prove the absence of a path, it will return `true`.

```
infix fn canPotentiallyReach(i: MemoryAccess, j: MemoryAccess) -> bool:
    return canPotentiallyReachWithoutPassingThrough(i, j, [])
```

`canPotentiallyReach` checks whether there *potentially* is a control flow path from memory access `i` to `j`. This function defers to `canPotentiallyReachWithoutPassingThrough` for the actual analysis.

```
infix fn alwaysPassesThroughAnyOf((i, j): (MemoryAccess, MemoryAccess),
                                  Fences: List<MemoryAccess>) -> bool:
    return !canPotentiallyReachWithoutPassingThrough(i, j, Fences)
```

`alwaysPassesThroughAnyOf` checks whether there is a fence in between a pair of memory accesses. This function must also be conservative. If Fency cannot prove whether a fence does or does not separate the pair, this function must return `false`. This function is implemented by checking whether there is a path from `i` to `j` that does *not* pass through any fences. If this is the case, then `alwaysPassesThroughAnyOf` must return false.

```
infix fn neverPassesThroughAnyOf((i, j): (MemoryAccess, MemoryAccess),
                                 Fences: List<MemoryAccess>) -> bool;
    for Fence in Fences:
        if (i canPotentiallyReach Fence) && (Fence canPotentiallyReach j):
```

```
            return false

    return true
```

`neverPassesThroughAnyOf` attempts to prove that there are no code paths where a fence separates `i` and `j`.

2.3.2 Collecting and constructing all concurrent Control Flow Graphs

Before we can apply robustness analysis, we first need to identify all concurrently running functions. Fency identifies these functions by scanning the LLVM IR module for calls to `pthread_create`. The third argument to `pthread_create` specifies the *thread entry point*.

After identifying a thread entry point, we can recursively construct concurrent Control Flow Graphs (CFG).

Each control flow graph contains zero or more memory access instructions. By checking whether one memory access can reach another, we can find all *potential* epo edges (memory access pairs).

The following pseudocode function describes how Fency collects all potential epo edges:

```
fn findMemoryAccessPairs(CFG) -> Set<(MemoryAccess, MemoryAccess)>:
  MemoryAccessPairs = Set()

  for MemoryAccess1 in CFG:
    for MemoryAccess2 in CFG:
      if MemoryAccess1 == MemoryAccess2:
        continue

      if MemoryAccess1 canPotentiallyReach MemoryAccess2:
        MemoryAccessPairs += (MemoryAccess1, MemoryAccess2)

  return MemoryAccessPairs
```

It includes any memory access pair that can potentially reach one another. Since `canPotentiallyReach` gives us a conservative over-approximation, we are sure to include all epo edges that might violate M-K robustness.

2.3.3 Constructing the Memory Pair Graph

During this step, Fency attempts to capture all potential eco edges. The Memory Pair Graph (MPG) is a graph where each node is a memory access pair, and each edge is a potential eco edge.

```
fn constructMPG(CFGs) -> MemoryPairGraph:
  Nodes: Set<(MemoryAccess, MemoryAccess)> = Set()
  for CFG in CFGs:
    for Pair in findMemoryAccessPairs(CFG):
      if hasConflictingAccess(CFGs, Pair):
        Nodes += Pair

  Edges = {}
  for Pair1@(A, B) in Nodes:
    for Pair2@(C, D) in Nodes:
      if Pair1.CFG != Pair2.CFG && B mayAlias C:
        Edges[Pair1] += Pair2

  return MemoryPairGraph(Nodes, Edges)
```

We can safely ignore memory accesses that do not *conflict* with other memory accesses because these accesses do not create eco relations. A read conflicts with a same-location write, and a write conflicts with a same-location read. As with `canPotentiallyReach`, it is crucial for `hasConflictingAccess` to be conservative - i.e., assume that a memory access has a conflicting access unless proven otherwise.

2.3.4 Finding non-robust memory access pairs

After constructing the Memory Pair Graph, Fency attempts to find all non-robust pairs that violate M-K robustness.

```
fn findNonRobustPairs(MPG: MemoryPairGraph,
                      Robust: ((MemoryAccess, MemoryAccess)) -> bool)
                      -> List<(MemoryAccess, MemoryAccess)>:
  NonRobustPairs = Set()
  for Pair in MPG:
    if onCycle(Pair) and !Robust(Pair):
      NonRobustPairs += Pair

  return NonRobustPairs
```

`onCycle` checks whether a pair is part of a $(\text{epo}; \text{eco})^+$ cycle. The `Robust` parameter is a function that decides whether a pair is M-K robust. The implementation of this robustness condition will be explained further in the next section.

2.3.5 Implementing the Robustness condition

The Robustness condition defines when an unordered memory access pair violates M-K robustness. Conceptually, the Robustness condition can be defined as:

```
fn isRobust(M: MemoryModel, K: MemoryModel,
            Pair: (MemoryAccess, MemoryAccess)):
  return isOrdered(K, Pair) || !isOrdered(M, Pair))
```

Intuitively, this means that a memory access pair is M-K robust if either:

- The pair is already ordered by weaker memory model K
- The pair is also unordered in stronger memory model M

For example, a write-read pair is x86-ARMv8 robust because, while this particular pair is unordered in ARMv8, it is also unordered in x86.

The SC-x86 Robustness condition can be expressed as follows:

```
fn x86::isSCRobust((i, j): (MemoryAccess, MemoryAccess)) -> bool:
    Fences = CFG.findAll(X86::MFENCE)

    return isRead(i) || isWrite(j)
        || (i mustAlias j)
        || ((i, j) alwaysPassesThroughAnyOf Fences)
```

In x86, all access pairs except write-read pairs are ordered. Write-read pairs that access the same memory or are fenced off are also ordered.

Note how the pseudocode is almost a direct translation of the robustness condition given in Section 2.2.

The other five robustness conditions for the remaining M-K robustness analyses can be found in Appendix A.

2.3.6 Enforcing M-K robustness

Now that we have identified all non-robust memory access pairs, we now know where we need to insert fences to restore M-K robustness.

```
fn restoreMKRobustness(NonRobustPairs: Set<MemoryAccessPair>,
                       Robust: ((MemoryAccess, MemoryAccess)) -> bool):
  for Pair in NonRobustPairs:
    if !Robust(Pair):
      Fence = createAppropriateFence(K, Pair)
      Fence.insertBetween(Pair)
```

This simple algorithm ensures that all non-robust pairs are adequately ordered by a fence. We repeat the `Robust` condition here to check that the pair is still non-robust (it may have been ordered by a fence inserted in a previous iteration).

`createAppropriateFence` depends on the target architecture. It is roughly defined as:

```
fn createAppropriateFence(K: Arch, (i, j): MemoryAccessPair) -> Fence:
  match (K) {
      x86 => return X86::MFENCE,
      ARMv7 => return ARMv7::DMB,
      ARMv8 => {
          if isWrite(i) && isRead(j):
            return ARMv8::DMB_ISH
          if isWrite(i) && isWrite(j):
            return ARMv8::DMB_ISHST
          else:
            return ARMv8::DMB_ISHLD
      }
  }
```

For x86 and ARMv7, we always insert `MFENCE` and `DMB` barriers, respectively. For ARMv8, we can check the type of memory accesses we need to order to determine the most lightweight fence possible. To order two writes, we only need a store fence (`dmb ishst`). We return a load fence to order two reads or a read-write (`dmb ishld`). For a write-read, we need a full fence (`dmb ish`).

This algorithm does not attempt to optimize fence placement: the actual implementation Fency uses always places a fence just before the second memory access of a pair, which will undoubtedly result in sub-optimal fence placement. It is possible to implement an algorithm that attempts to find a better position for the fences, as has been done by, for example, Bouajjani et al. in [16].

Chapter 3

Fency v2

We made numerous improvements to the original implementation of Fency.

The first improvement focuses specifically on the ARMv8 platform. Fency did not track instruction dependencies yet. By implementing a compiler pass that tracks these dependencies, we can prevent some unnecessary fences whenever we are targeting the ARMv8 platform.

Another improvement that we implemented in Fency 2.0 is alias analysis. LLVM provides existing alias analysis algorithms that we can leverage to prove that instructions never alias.

The third improvement is that we implemented function call support. This impacted two areas within Fency: building the control flow graph and the reachability analysis. The control flow graph has to be extended recursively with any functions that are called from the thread entry point. Whenever two instructions reside in different functions, we use LLVM's call graph to check whether one function can be reached from another.

The final improvement extends Fency's usability. In the first version of Fency, all functions had to be defined before the `main` function. By switching to a ModulePass instead of a FunctionPass we could rid Fency of this restriction.

3.1 Adding instruction dependency analysis

Before we look at how dependency tracking was implemented in Fency, we will look at a small example where Fency inserted unnecessary fences.

Recall that the dob relation is defined as:

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

Two terms in this definition are problematic when implementing static analysis: rfi and coi. These relations are established at runtime and cannot possibly be known without executing a program.

Therefore, instead of (addr∪data); rfi we will use (addr∪data); lrs). lrs (local-read-successor) was coined by Alglave et al. in [14]. It fully replaces rfi in their axiomatic model because it fits better with their "per-thread reasoning". We will look at lrs more closely later in this section.

Unfortunately, there is no similar solution for coi. Since coi is not statically analyzable, we leave it out of the dependency tracking implementation.

We will now look at the different terms of the dob relation.

data – data dependencies

Data dependencies, denoted by the data relation, are inter-instruction dependencies where the second instruction of two po-related instructions depends on data computed by the first instruction. The dependee is always a store instruction, while the dependent is always a load instruction.

A simple example of a data dependency is a load instruction immediately followed by a store instruction that stores the loaded value back into memory. We will use a similar notation to what LLVM uses for its MIR (Machine IR). While Fency runs, MIR is still in static single-assignment form (SSA). The registers in the following examples are virtual registers.

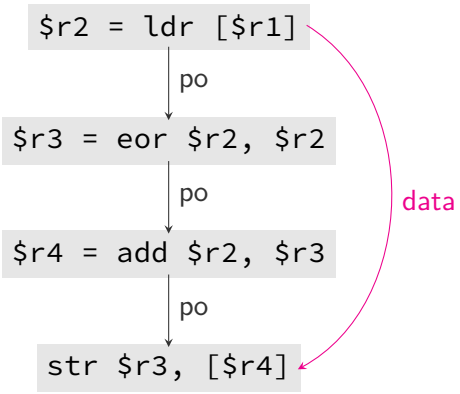
In this simple example, there is a data dependency from the load to the store instruction.

```
1   $r2 = ldr [$r1]
2   str $r2, [$r3]
```

Data dependencies are carried through any intermediate instructions, except for other memory operations [12], as is the case in the following ARMv8 assembly snippet:

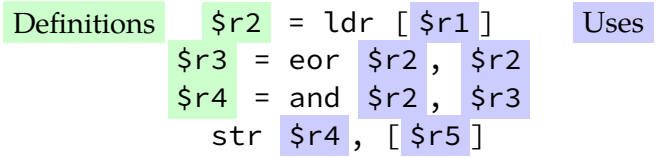
```
1   $r2 = ldr [$r1]
2   $r3 = eor $r2, $r2
3   $r4 = add $r2, $r3
4   str $r4, [$r5]
```

If we annotate this snippet with po edges and a data edge, this snippet will look like this:

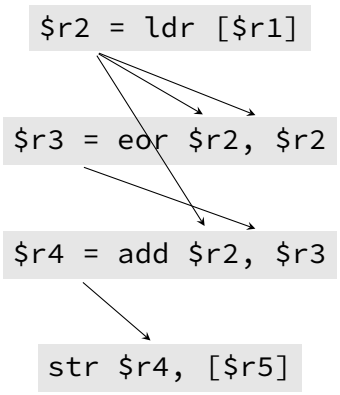


Even though the final store instruction does not *directly* depend on the earlier load, ARMv8 still considers the store to be dependent on the load.

Therefore, to support situations where a store operation indirectly depends on a load, we have to search through the *def-use* chain. Registers occurring on the left-hand side of the equals-sign are register *definitions*. Registers on the right-hand side of an assignment are register *uses*.



LLVM keeps track of all register definitions and uses. The result is a graph-like structure.



Each edge in this def-use chain originates at a register definition and points to a use of the defined register.

To find a data dependency, we have to traverse the def-use chain backwards, starting from the data operand of the store instruction (`$r4`).

Note that a single store instruction can depend on multiple load instructions, as in the following example.

```
1   $r2 = ldr [$r1]
2   $r4 = ldr [$r3]
3   $r5 = add $r2, $r4
4   str $r5, [$r6]
```

In this example, the store instruction depends on both load instructions. However, the two load instructions do *not* depend on one another - they can still be reordered.

Collecting all data dependencies requires us to collect all store instructions first. Then, for each store instruction, we find all load instructions that the data operand depends on.

addr – address dependencies

Address dependencies, which we capture in the addr relation, are similar to data dependencies. There are two main differences:

1. Data dependencies only exist between a Load and a Store. Address dependencies can be formed between a Load and a Store or two Load instructions.
2. Instead of looking at the data operand of a load or store instruction, we look at the address operand.

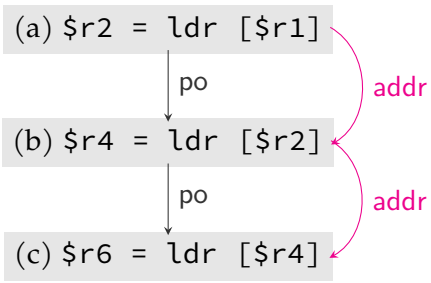
Consider the following two snippets:

```
1   $r2 = ldr [$r1]
2   $r4 = ldr [$r2]
```

```
1   $r2 = ldr [$r1]
2   str $r4, [$r2]
```

In both cases, there is an address dependency from the first load to the second memory access.

Like data dependencies, address dependencies are carried through other instructions, except for other loads and stores [12].



Load (c) depends on load (b), which in turn depends on load (a). Note that there is no addr relation from (a) to (c) because address dependencies do not carry through other loads and stores.

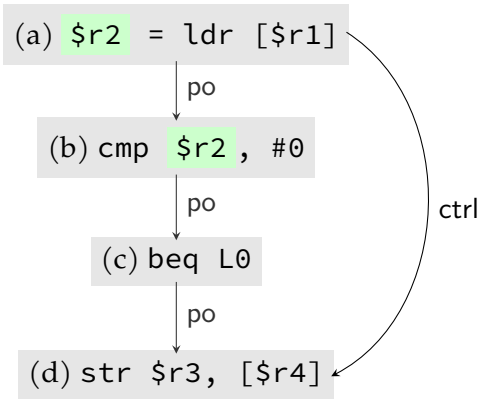
Collecting all address dependencies is done similarly to collecting the data dependencies, the difference being that, instead of only considering store instructions, we now also need to consider load instructions.

ctrl; [**W**] – control dependencies into a store instruction

The ARMv8 architecture also orders load and store instructions that are ctrl-related.

A control dependency between a load instruction and a store instruction arises when the result of the load instruction is depended on by a conditional branch instruction that po-precedes the store instruction [12], as is the case in the following snippet:

```
1   $r2 = ldr [$r1]
2   cmp $r2, #0
  ; compare $r2 with value 0
3   beq L0
  ; branch to L0 if $r2 == 0
4   str $r3, [$r4]
5 L0:
```



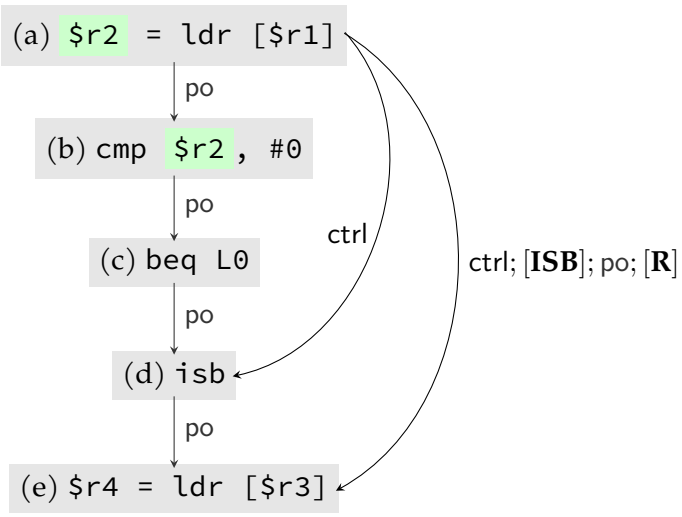
In this example, there is a ctrl relation between load instruction (a) and store instruction (d) because the load instruction is used to compute branch (c), which po-precedes (d).

Collecting all control dependencies is more involved than collecting address or data dependencies. First, we must collect all conditional branch instructions. Then, we should check whether those conditional branch instructions depend on any loads. If a conditional branch depends on a load instruction, we check whether there are any store instructions that po-succeeds the load instruction. Finally, we must ensure that the store instruction is in a different basic block than the load instruction.

ctrl; [**ISB**]; po; [**R**] – ISB as a control fence

The Instruction Synchronization Barrier (ISB) is able to order two load instructions when it itself is in ctrl relation with an earlier load [5, 2].

```
1    $r2 = ldr [$r1]
2    cmp $r2, #0
3    beq L0
4    isb
5    $r4 = ldr [$r3]
6  L0:
```



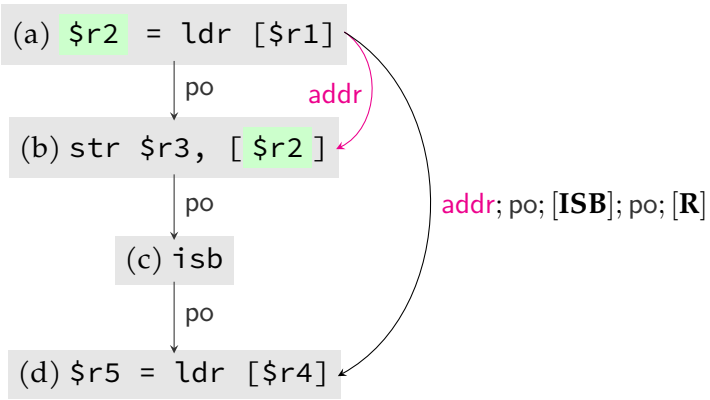
In this case, load instruction (a) is in ctrl; [**ISB**]; po; [**R**] relation with load instruction (e). This relation sufficiently orders the two load instructions.

Collecting these dependencies is done similarly to regular ctrl dependencies. As an extra step, we must check whether a load instruction is in po relation with the `ISB` instruction.

addr; po; [**ISB**]; po; [**R**] **– Address dependency with ISB**

ISB instructions can also order two read instructions when the first read instruction is addr-related to another memory access, which comes po-before the ISB. The ISB has to be po related to the second read instruction.

```
1    $r2 = ldr [$r1]
2    str $r3, [$r2]
3    isb
4    $r5 = ldr [$r4]
```



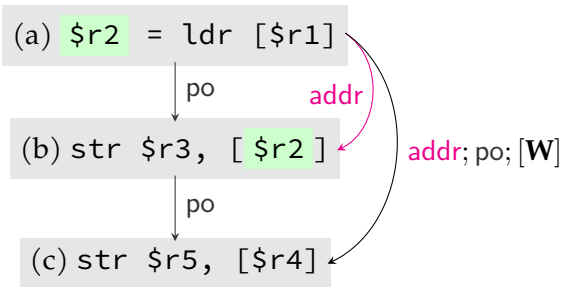
The ISB instruction (c) ensures that load instruction (d) is not reordered with load instruction (a). Note that instruction (b) can be either a load instruction or a store instruction; the only requirement is that it has an address dependency on (a).

This composite relation requires us to collect all addr-related dependencies first. We can then filter the ISB instructions that are in po-relation with the codomain of addr. Similarly, we then filter the load instructions that are po-related to the filtered ISB instructions.

addr; po; [**W**] **– Address dependency that orders a subsequent write**

The ARMv8 architecture orders a Read-Write (RW) pair where the read is addr-related to a memory access that po-precedes a the write instruction [5, 12].

```
1    $r2 = ldr [$r1]
2    str $r3, [$r2]
3    str $r5, [$r4]
```



The addr; po; [**W**] relation ensures that write instruction (c) cannot be reordered with load instruction (a).

This relation also depends on the addr-related dependencies. After finding these dependencies, we can check whether any store instruction is po-related to the codomain of an address dependency.

(addr ∪ data); lrs – address or data dependency with local-read-successor

In [14] Alglave et al. removed rfi from their ARMv8 axiomatic model and replaced it with lrs. They define lrs as

$$\text{lrs} \triangleq [\mathbf{W}]; \text{po}_\ell \backslash (\text{po}_\ell; [\mathbf{W}]; \text{po}_\ell); [\mathbf{R}]$$

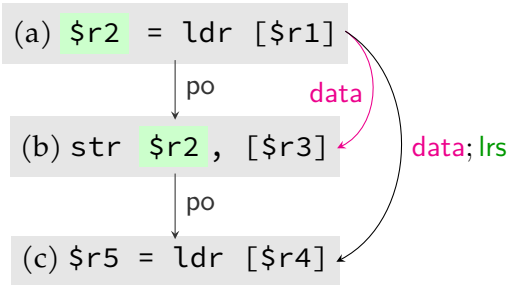
The set minus operation in the middle ensures that there is no intervening write between the po_ℓ-related read and write.

Alglave et al. acknowledge that their definition of lrs is a subset of rfi, but they prove that it does not make their axiomatic model inconsistent with the ARMv8 hardware. We can, therefore, safely adopt their notion of lrs. Since lrs is determined solely from po_ℓ, it is more appropriate to use in a static analysis tool than rfi. At the same time, its dependency on po_ℓ brings its own challenges.

po_ℓ requires Fency to be able to prove that two memory accesses always access the same memory location (i.e., they must always *alias*).

The relation (addr ∪ data); lrs orders two load instructions when the first load instructions is addr or data related to a store and this store is lrs-related to the second load instruction.

```
1    $r2 = ldr [$r1]
2    str $r2, [$r3]
3    $r4 = ldr [$r3]
```



In this example, the data; lrs relation ensures that load instruction (a) cannot be reordered with load instruction (c). Note that this relation can only be established if we can prove that `$r3` and `$r4` always point to the same memory location.

3.2 Improving Fency's alias analysis

In 2.3.1, we defined two functions that Fency uses to check whether two instructions access the same memory location. Fency uses `potentiallyAccessesSameMemoryAs` to construct the memory pair graph, and it uses `alwaysAccessesSameMemoryAs` to implement the M-K Robustness conditions.

3.2.1 Potential same-location accesses

To check whether two instructions potentially access the same memory location, we can leverage LLVM's existing alias analysis infrastructure. More specifically, Fency uses the `MachineInstr::mayAlias` API, which contains a comprehensive (but conservative) algorithm to check whether two machine instructions may access the same memory location. Under the hood, `MachineInstr::mayAlias` delegates most of its alias analysis to `AliasAnalysis::alias()`.

Target-specific alias analyis

Sometimes, a target can immediately tell us whether two memory locations can never overlap. For example, in ARMv8, memory accesses are "trivially disjoint" if the base address is equal, but the offset of the lower memory access plus the width of the access does not overlap with the offset of the higher memory access. The most simple case is with disjoint array accesses:

```
1 int SharedX[2];
2
3 void main() {
4     int x1 = SharedX[0];
5     int x2 = SharedX[1];
6 }
```

The base address is the same in both memory accesses, but the offset for `x1`, which is 0, plus the width, which is 4 bytes, is equal to the offset for `x2` (4 bytes), meaning that these accesses are trivially disjoint.

This target-specific information is provided to us via the `TargetInstrInfo` class.

Machine instruction memory locations

A machine instruction may affect zero or more memory locations. This results in quadratic complexity when checking whether two machine instructions overlap since we need to check any combination of all affected memory locations.

```
1 fn mayAlias(A: MachineInstr, B: MachineInstr) -> bool:
2     for MemoryOperandA in A.memory_operands():
3         for MemoryOperandB in B.memory_operands():
4             if (!AliasAnalysis::noAlias(MemoryOperandA, MemoryOperandB)):
5                 return true
6     return false
```

The machine instructions we are primarily interested in for our analysis affect 2 or 3 memory locations. However, ARM's `ldm` instruction can affect many more memory locations at the same time. To reduce the impact of this quadratic behavior, LLVM imposes a limit on how many memory operands it will check, conservatively defaulting to `true` when this limit is exceeded.

LLVM's alias analysis infrastructure

`MachineInstr::mayAlias` delegates the actual alias analysis to `AliasAnalysis::noAlias()`. This API is part of the alias analysis subsystem. LLVM exposes multiple alias analysis implementations. The basic alias analysis implementation is always available. This aggressive, function-local analysis can deduce a lot of information:

- Whether two variables point to distinct allocations
- An allocation never aliases the null pointer

- Struct fields can never alias
- Different array indices cannot alias
- It knows about common C functions - many of them never access memory
- It knows when pointers always point to constants
- Function-local stack allocations cannot alias if they don not escape from the current function

3.2.2 Proving that memory accesses always access the same location

Unfortunately, LLVM does not provide us with `MachineInstr::mustAlias()`. To implement Fency's `alwaysAccessesSameLocation` we need to directly use the lower-level `AliasAnalysis::alias()` interface.

`alwaysAccessesSameLocationAs` is often used by the Robustness conditions to check whether two memory accesses are M-K robust. For example, in x86, a pair consisting of a write and a subsequent read is generally not robust unless they access the same location.

Therefore, we implement `alwaysAccessesSameLocationAs` as follows:

```
1 infix fn alwaysAccessesSameLocationAs(A: MachineInstr,
2                                       B: MachineInstr) -> bool:
3     for MemoryOperandA in A.memory_operands():
4         for MemoryOperandB in B.memory_operands():
5             if (AliasAnalysis::mustAlias(MemoryOperandA, MemoryOperandB)):
6                 return true
7
8     return false
```

If any memory operand pair of the two instructions always alias, we return true.

3.2.3 Soundness

It is important that `potentiallyAccessesSameMemoryAs` and `alwaysAccessesSameMemoryAs` are conservative when necessary to preserve Fency's soundness. If Fency would not be sound, it would insert too few fences, which would cause it to output a program that is not actually M-K robust.

The previous version of Fency, referred to as simply "v1", was unsound because it would ignore memory accesses that were not accessing a global variable.

For example, it would incorrectly report this variation of the load buffering program from the introduction to be x86-ARMv8 robust.

```
1  std::atomic<int> SharedX;
2  std::atomic<int> SharedY;
3
4  void* threadA(void *arg) {
5    std::atomic<int> *PtrToSharedX = static_cast<std::atomic<int>*>(arg);
6    int localX = PtrToSharedX->load(std::memory_order_relaxed);
7    SharedY.store(1, std::memory_order_relaxed);
8
9    return nullptr;
10 }
11
12 void* threadB(void *arg) {
13   std::atomic<int> *PtrToSharedY = static_cast<std::atomic<int>*>(arg);
14   int localY = PtrToSharedY->load(std::memory_order_relaxed);
15   SharedX.store(1, std::memory_order_relaxed);
```

```
16
17   return nullptr;
18 }
```

Instead of sharing data between threads with a global atomic variable, we pass a pointer to the atomic variable as an argument to the thread. Because the atomic loads do not access the global variables directly, v1 will ignore them. V2 is much more conservative: even though LLVM cannot determine *which* memory location is accessed with the atomic load, it will assume the worst-case scenario and mark the atomic load as conflicting with the atomic write in the other thread. This results in Fency conservatively and correctly inserting two fences to order the read-write pairs.

3.3 Supporting function calls

Any sufficiently complex program will distribute its logic across multiple functions. If we want to use Fency to analyze these more complex, realistic programs, we need to correctly implement inter-procedural analysis for `canPotentiallyReach` and `canAlwaysReach`.

3.3.1 The call graph

LLVM builds a call graph for each IR module. The call graph is a directed graph, possibly cyclic, which stores information about which function can potentially call some other function.

Every function is represented as a node in the graph. Edges between nodes indicate that one function may call another function. The call graph represents a conservative superset of caller-callee relationships.

3.3.2 Checking whether an access potentially reaches another access

`canPotentiallyReach` is a conservative reachability analysis function that Fency uses to collect all potential memory access pairs. To preserve Fency's soundness, `canPotentially-Reach` must be conservative when it is unable to prove whether two memory accesses can *never* reach one another.

When Fency has to check whether two memory accesses from different functions (but within the same control flow graph) can potentially reach one another, it does a depth-first search in the call graph.

```
1 fn hasCallGraphPath(From: MemoryAccess, To: MemoryAccess) -> bool:
2     Start = CallGraph.find(From.function())
3     End = CallGraph.find(To.function())
4
5     for Node in depth_first_iterator(Start):
6         if Node == End:
7             return true;
8
9     return false
```

The call graph also contains a special "external" node. An external node represents all calls to functions defined outside of the current IR module. Any edges to this node indicate that a function directly or indirectly calls an external function. Any edges from the external node to a function indicate that this function might be called from an external source, for example, if the function contains an indirect (polymorphic) function call.

For now, Fency does not support call graphs with external nodes and ignores them.

3.3.3 Checking whether an access always reaches another access

Fency is currently unable to analyze whether a memory access can always reach another memory access in a different function. It uses dominator trees for its function-local analysis. Unfortunately, inter-procedural dominator trees are not available in LLVM.

Chapter 4

Evaluation

In this chapter, we attempt to evaluate the improvements described in Chapter 3. This chapter has been split into three sections. In Section 4.1, we will compare the new Fency version ("v2"), with the new alias analysis implementation, to the original prototype ("v1"). We will separately evaluate the new dependency ordered-before analysis in Section 4.2. Finally, in Section 4.3, we will run Fency v2 on several C11 benchmark programs, which is now possible due to function call support.

4.1 Comparing Fency v2 with Fency v1

Fency v2 is an entirely new implementation. To evaluate the effectiveness of this new implementation, we will compare Fency v1 with Fency v2.

4.1.1 Method

To compare Fency v2 with Fency v1 and evaluate its performance, we will run both versions on 33 programs. Fency v1 was previously evaluated on this set of programs in [5].

All programs consist of 70-450 lines of LLVM IR. The concurrent control flow graphs in these programs do not consist of multiple functions (i.e., there are no function calls). This makes them analyzable by Fency v1, which does not support function calls.

For each of the six supported M-K robustness analyses, SC-x86, SC-ARMv8, x86-ARMv8, SC-ARMv7, x86-ARMv7, and ARMv8-ARMv7, we collected:

- The size of the MPG (number of nodes and edges), which gives an indication of the program size
- The number of detected non-robust pairs. Fewer is better.
- The number of inserted fences. Fewer is better.
- Whether Fency v1 or v2 consider the entire program M-K robust
- The time it took for the analysis

The following sections will show and discuss the results for each supported M-K robustness configuration.

4.1.2 Results

The full results are available in Appendix B. In the discussion, we will highlight several of the most interesting results from the set.

M-K	NRP	F	Time (ms)
SC-x86	-4/0	-2/0	0/2553
SC-ARMv8	-2/18	-6/4	0/342
x86-ARMv8	-1/18	-5/4	0/357
SC-ARMv7	-2/16	-1/0	-5448/44
x86-ARMv7	0/16	-1/0	-5426/43
ARMv8-ARMv7	0/16	-1/0	-5291/53

Table 4.1: The aggregated results across all programs of the difference between Fency v1 and v2. NRP: Number of non-robust pairs. "F": Number of fences inserted. Each cell contains the minimum and maximum value.

Table 4.1 contains the aggregated results across all programs. Each cell contains the minimum and maximum value across all 33 programs. For example, the best result for SC-ARMv8 is -6 fences, while in the worst case Fency v2 inserts +4 more fences than Fency v1.

4.1.3 Discussion

We will now discuss the results for each M-K robustness configuration.

SC-x86

Program	Nodes	Edges	NRP	Fences
cilk-sc	96/104/**+8**	3109/4169/**+1060**	0/0/0	0/0/0
lamport-ra	9/13/**+4**	72/156/**+84**	0/0/0	0/0/0
peterson-ra-b	245/245/0	15468/15468/0	9/45/**+36**	6/4/**-2**
ticketlock	50/50/0	882/990/+108	0/4/+4	0/2/+2
ticketlock4	100/100/0	3564/5940/+2376	0/8/+8	0/4/+4

Table 4.2: Highlighted results for SC-x86

cilk-sc. Fency v2 creates a Memory Pair Graph with 8 more nodes and 1060 more edges than Fency v1. Fency v2 creates a larger MPG because Fency v2 includes a memory access to the thread-local variable that Fency v2 ignores. This one memory access reaches 8 other accesses, resulting in an extra 8 memory access pairs. The original 96 memory access pairs have to create 8 more edges to the new pairs, resulting in $96 \times 8 = 768$ more edges. The new pairs have 36 outgoing edges each (on average), which results in an additional $8 * 36 = 292$ more edges. The new edges do not influence the number of inserted fences because the new pairs are all SC-x86 robust.

lamport-ra. Fency v2 adds 4 nodes to the MPG for this program. These 4 nodes all reference the thread argument and thus result in 84 additional edges. Similarly to cilk-sc, these edges do not result in additional fences because all pairs are SC-x86 robust.

peterson-ra-b. Fency v1 and v2 create the same MPG. Interestingly, Fency v2 is able to prove the robustness of more pairs. This may be due to a bug in Fency v1's instruction reachability analysis. All four pairs are Write-Read pairs separated by an RMW. An RMW acts like a fence on x86 [5]. Because these four pairs are robust, Fency v2 inserts 2 fewer fences.

SC-ARMv8

Program	Nodes	Edges	NRP	Fences
cilk-sc	232/244/+12	19375/42425/+23050	73/76/**+3**	8/9/**+1**
lamport-sc	9/13/**+4**	72/156/**+84**	0/2/**+2**	0/1/**+1**
lb-2	0/2/+2	0/2/+2	0/2/+2	0/2/+2
rcu-offline	193/235/+42	6290/30011/+23721	43/55/+12	14/8/**-6**
spinlock	74/90/+16	3618/7538/+3920	8/12/+4	4/6/**+2**

Table 4.3: Highlighted results for SC-ARMv8

cilk-sc. Fency v2 creates more nodes because of the accessed thread argument. However, even though this results in significantly more edges, only three nodes are non-robust pairs on a cycle. The only difference in fence placement between Fency v1 and v2 is a `dmbld` fence to order instructions involving the thread argument.

lamport-sc. Fency v2 inserts one additional fence to order two instructions involving the thread argument.

lb-2. This program is a variation of the load-buffering litmus test. Instead of using shared variables to communicate between threads, the threads communicate using the thread argument, which contains a pointer to an atomic int. Because Fency v2 cannot know whether the pointer points to the same object, it conservatively inserts two fences. The load-buffering litmus test indeed needs two fences; thus, Fency v2 is correct.

rcu-offline. Fency v2 inserts six fences less than Fency v1. The fences that were omitted mainly attempted to order two instructions in a loop. However, these instructions always access the same memory location according to Fency v2's alias analysis. Therefore, we can safely omit these fences.

spinlock Fency v2 inserts more fences than v1 in this program because it orders instructions accessing the thread argument.

x86-ARMv8

Fency v1 and v2 perform similarly for x86-ARMv8. The programs where their outputs differ are the same programs where the output for SC-ARMv8 differs. This difference is primarily because of Fency v2's conservative approach regarding thread arguments.

SC-ARMv7

Program	Nodes	Edges	NRP	Fences
peterson-ra-b	162/162/0	6592/17710/+11118	4/2/**-2**	2/1/**-1**
rcu-offline	217/235/+18	7093/30011/+22918	15/29/**+14**	11/10/**-1**

Table 4.4: Highlighted results for SC-ARMv7

peterson-ra-b. Fency v2 inserts one less fence compared to Fency v1. Fency v2 is able to prove that a particular store-store pair always passes through a DMB fence. It is unclear why Fency v1 does not come to the same conclusion. This might be related to what happened for peterson-ra-b for SC-x86.

rcu-offline. Fency v2 inserts one fence less compared to v1. In this case, Fency must prove that a store instruction always accesses the same memory location. Fency v2 concludes this is the case, while Fency v1 conservatively inserts a fence. It is unclear whether the fence is necessary: the program contains many of hard-to-trace memory accesses, making it complex to answer this query conclusively.

x86-ARMv7

Program	Nodes	Edges	NRP	Fences
dekker-sc	102/102/0	3752/3762/0	28/28/0	4/3/-1

Table 4.5: Highlighted results for x86-ARMv7

The results for x86-ARMv7 are largely similar to SC-ARMv7.

dekker-sc. Fency v2 inserts one less fence because the non-robust pair is not part of a cycle.

ARMv8-ARMv7

With the exception of dekker-sc, Fency v1 and Fency v2 output the same amount of fences for ARMv8-ARMv7.

4.2 Evaluating the impact of DOB analysis on Fency's fence placement

To properly evaluate whether the ARMv8, instruction dependency analysis (DOB analysis) causes Fency to insert fewer fences overall when targeting ARMv8 we will compare Fency v2 *with* instruction dependency analysis to Fency v2 *without* instruction dependency analysis.

4.2.1 Method

Both the SC-ARMv8 robustness condition and the x86-ARMv8 robustness condition use the dob relation. However, since the only difference between the two is the wr relation, we will only analyze SC-ARMv8 robustness with and without DOB analysis.

Enabling DOB analysis should result in fewer non-robust pairs, which may result in fewer fences. Not all programs benefit equally from this analysis: programs with many dependencies between non-robust pairs will benefit the most.

4.2.2 Results

Table 4.6 contains the results of running Fency v2 with and without DOB analysis.

- The first column contains the program name.
- The second column, "Dep", shows the number of instruction dependencies found in the entire program.
- The third column, "NRP", shows how many non-robust pairs were found without (WO) and with (W) DOB analysis enabled. The difference Δ is calculated as (W - WO). Fewer is better.
- The fourth column, "F", contains the number of fences that Fency inserted without (WO) and with (W) DOB analysis enabled. Fewer is better.

Table 4.6: Results of running Fency with and without DOB analysis. "NRP": number of non-robust pairs found. "F": number of fences inserted. "W": with DOB-analysis enabled. "WO": without DOB-analysis.

Program	Dep.	NRP (WO/W/Δ)	F (WO/W/Δ)
barrier	2	4/4/0	2/2/0
cilk-sc	60	83/76/-7	9/9/0
cilk-tso	60	47/45/-2	8/8/0
cldequeue-ra	16	4/4/0	3/3/0
cldequeue-ra-noloop	14	1/1/0	1/1/0
cldequeue-sc	16	18/17/-1	7/7/0
cldequeue-sc-noloop	14	10/9/-1	4/4/0
cldequeue-tso	16	13/13/0	6/6/0
dekker-sc	11	46/43/-3	7/7/0
dekker-tso	11	14/11/-3	4/3/-1
iriw	0	5/5/0	4/4/0
lamport-ra	8	0/0/0	0/0/0
lamport-sc	6	3/2/-1	1/1/0
lamport-tso	6	0/0/0	0/0/0
lb	2	2/2/0	2/2/0
lb-2	2	2/2/0	2/2/0
mp	2	2/2/0	2/2/0
mutex	2	0/0/0	0/0/0
nbw	9	32/32/0	9/9/0
peterson-ra	8	20/14/-6	8/8/0
peterson-ra-b	19	6/6/0	4/4/0
peterson-ra-d	10	22/22/0	10/10/0
peterson-sc	8	38/32/-6	10/10/0
peterson-tso	5	10/7/-3	6/4/-2
rcu	30	62/62/0	11/11/0
rcu-offline	101	70/55/-15	10/8/-2
sb	0	0/0/0	0/0/0
seqlock	8	4/4/0	3/3/0
spinlock	20	16/12/-4	6/6/0
spinlock4	41	32/24/-8	12/12/0
ticketlock	13	8/8/0	2/2/0
ticketlock4	25	16/16/0	4/4/0
usb	0	10/10/0	4/4/0

4.2.3 Discussion

The majority of the programs we tested were unaffected by the DOB analysis. One possible explanation is the lack of instruction dependencies in each program.

Only a few programs, cilk-sc, cilk-tso, rcu, and rcu-offline, see a reduction in both the number of non-robust pairs and the number of required fences. Incidentally, these programs also contain the highest number of instruction dependencies.

Consider this assembly snippet from rcu-offline:

```
1 ldr     w15, [x10, :lo12:m]
2 dmb ishld
3 cmp     w15, #0
4 csel    x15, x12, x11, eq    ; implicitly depends on cmp
5 ldar    w15, [x15]
```

The second load instruction (`ldar`) has an (implicit) address dependency on the first load instruction, `ldr`. The registers through which the address dependency is carried are marked red. The `csel` instruction implicitly depends on the `cmp` instruction, which in turn depends on the `ldr`.

This dependency eliminates the need for the `dmb ishld` fence. With DOB analysis enabled, Fency will indeed correctly omit the fence.

4.3 Running Fency v2 on C/C++11 programs

Ideally, we would like to run Fency on a large, concurrent program such as Chromium, Firefox or Redis. However, Fency does not support programs that consist of multiple compilation units.

It might still be interesting to see whether Fency's newly gained function call support enables it to run on larger programs, even though these programs are restricted to a single compilation unit. To evaluate the function call support, we will run Fency v2 on the CDS Checker benchmarks [8]. These benchmarks are written in C/C++11. They implement several data structures and algorithms, such as a concurrent hashtable, hashmap, and Dekker's algorithm.

4.3.1 Method

We run Fency's SC-x86 robustness analysis on the CDS Checker benchmarks [8]. It is, unfortunately, not possible for Fency to analyze multiple LLVM modules, so we are restricted to programs that consist of a single compilation module. We can run Fency on the following benchmarks:

- barrier (174 LoC): A simple implementation of a spinning barrier.
- cliffc-hashtable (1800 LoC): A simplified version of Cliff Click's concurrent hashmap implementation [17].
- concurrent-hashmap (800 LoC): Another concurrent hashmap implementation.
- dekker-fences (180 LoC): Dekker's well-known mutual exclusion algorithm. This program is similar to Fency's dekker-sc.
- linuxrwlocks (130 LoC): An example implementation of the Linux RW locks.
- mcs-lock (350 LoC): An MCS lock implementation.

Table 4.7: Results of running Fency on a subset of the CDS Checker benchmarks

Program	RWs	Nodes	Edges	Fences	NRP	Time (ms)
barrier	18	66	2254	1	2	58
cliffc-hashtable	460	-	-	-	-	8977
concurrent-hashmap	124	2744	5895548	2	60	76740
dekker-fences	22	132	5942	2	4	52
linuxrwlocks	28	222	38790	0	0	88
mcs-lock	62	938	420578	8	27	10873
mpmc-queue	30	158	18142	0	0	319
seqlock	14	38	1300	0	0	13

- mpmc-queue (320 LoC): An implementation of a multi-producer, multi-consumer queue.
- seqlock (170 LoC): A sequence lock implementation.

4.3.2 Results

The results of running Fency on the above programs can be viewed in Table 4.7. The table includes the following:

- RWs: The number of reads and writes in the program. This gives an approximation of the program size.
- Nodes & Edges: the number of nodes and edges in the MPG.
- NRP: The number of non-robust pairs Fency found.
- Time: The time it took in ms for Fency to analyze the complete program and insert the appropriate fences.

The number of MPG nodes required for the cliffc-hashtable program exceeded the configured limit (10,000 nodes).

Fency concludes linuxrwlocks, mpmc-queue, and seqlock to be SC-x86 robust. The analysis takes between 14 milliseconds up to 76 seconds.

4.3.3 Discussion

The time it takes for Fency to analyze a program is strongly related to the size of the program. For example, it only takes a couple of milliseconds to verify the seqlock program with 14 read and write operations. It takes Fency over 70 seconds to analyze the concurrent-hashmap with 3068 MPG nodes. This can be explained by the time complexity of the robustness checking algorithm, which is $O(n^6)$ for a program consisting of n accesses[5]. The theoretical maximum number of nodes in the MPG is n^2, and the maximum number of edges is n^4. Therefore, the number of nodes for cliffc-hashtable can well exceed 200,000 nodes and 44,000,000,000 edges. The robustness checking algorithm becomes intractable as program size increases.

Chapter 5

Related work

Much work has been done on weak memory models, robustness checking and fence insertion. Comparable tools focus on SC-K robustness, while Fency expands this existing work by implementing M-K robustness for SC, x86, ARMv8 and ARMv7. This section discusses other, comparable tools and highlights the differences between them and Fency.

We will also dicuss other relevant work on (weak) memory models and fence insertion.

5.1 Robustness checking and fence insertion tools

Fency is not the first robustness checking tool. Numerous other tools came before it. We will discuss the most relevant tools in this section.

Generally, we can classify the robustness checking tools by:

- Exploration technique - how they explore possible executions of a program. Some tools, including Fency, overapproximate the possible executions traces to capture all potential executions at once, while other tools may use model checkers to explore the state space. Model checkers, such as SPIN [18], explore all possible execution traces one-by-one by simulating the concurrent program.
- Type of model - how they determine whether an execution is valid. Fency uses an axiomatic model to verify whether an execution is M-K robust. Another common approach is to check an execution using an operational model. Operational models are abstractions of actual machines [19]. For example, an operational model might implement a store buffer to simulate x86-TSO.

5.1.1 Trencher

Trencher [16] is a program that checks whether a program is SC-x86 robust. Its input format is a text file that describes a program as a state machine. It uses the SPIN model checker [18] to enumerate all possible executions, and attempts to find executions that cannot happen under SC, which they call "attacks". It uses an *operational model* to simulate store buffers. When it determines that an execution trace is possible in TSO but not in SC, it tries to find the execution cycle that causes this difference in behavior. It tries to find an optimal set of fences by encoding the fences as a 0/1-integer linear programming (ILP) problem. Given a cost function C, it tries to compute a fence set F that minimizes the following equation:

$$\sum_{f \in F} C(f)$$

For example, with a cost function $C(f) = 1$, it will minimize the size of the fence set. Finally, Trencher outputs state transitions that need to be strengthened by fences.

5.1.2 Offence

Offence[10] is a robustness checker and fence insertion tool that supports SC-x86, SC-Power and SC-ARMv7 robustness. It takes small assembly snippets annotated with memory locations as its input. Offence then checks these snippets for robustness against weak memory models, which they call "stability". Unlike Trencher, which uses a model checker, Offence overapproximates execution traces. Also unlike Trencher, Offence uses an axiomatic model to check whether an execution trace is allowed in SC. Offence tries to maximize the number of pairs that are ordered by a single fence.

5.1.3 Musketeer

Musketeer[11] is a robustness checker and fence insertion tool. It supports SC-x86, SC-Power and SC-ARMv7. Given a C program compiled with goto-cc[1], it is able to compute the fences required to make it robust.

Like Fency and Offence, it overapproximates execution traces and uses an axiomatic model. While constructing the CFG, which they call the *abstract event graph*, they inline all function calls (this prevents Musketeer from supporting recursion). Unlike Fency and Offence, Musketeer uses integer linear programming to insert fences. Instead of simply using $C(f) = 1$ as their cost function, they assign costs to fences based on whether they are full fences, lightweight fences, control fences or dependencies.

After computing the optimal fence set (w.r.t. the overapproximated execution traces), it inserts these fences into the C source code using inline assembly statements. Interestingly, Musketeer is also supports inserting *fake dependencies* instead of fences.

Consider the following ARM snippet:

```
1 ldr w15, [x]
2 str w16, [y]
```

To order this pair, we can either insert a store fence between the memory accesses, or we can create a fake dependency.

```
1 ldr w15, [x]
2 eors w15, w15, w15
3 str w16, [y, w15]
```

Musketeer inserts an XOR instruction that always evaluates to 0. It then uses the result of this XOR instruction as an offset for the store instruction. This results in the processor thinking that the store instruction depends on the previous load instruction.

Musketeer has been successfully run on larger programs, such as memcached.

5.2 Other weak memory model tools

Robustness checkers are not the only tools that attempt to analyze weak memory behavior. In this section, we will highlight several tools that take a slightly different approach, analyze other aspects of weak memory models, or help support weak memory research.

5.2.1 VSync

The VSync tool [20] has a slightly different goal compared to Fency, Trencher and Musketeer. Instead of inserting fences, VSync tries to maximally *relax* existing fences in a program. For example, if VSync tries to determine when a full fence can be *relaxed* to just a load fence.

VSync broadly implements the following algorithm:

[1] `goto-cc` is part of the CProver framework, which is a suite of formal verification tools for C. See `https://www.cprover.org/goto-cc/`

1. Start with a program where every fence is a full fence
2. Relax one of the fences in the program: replace this fence with a lighter fence.
3. Check whether the program is still correct using a (proprietary) model checker
4. If it is correct: go back to step 2 and try to relax another fence.
5. If the program is not correct anymore: restore the original fence.
6. If we maximally relaxed all fences, we are done.
7. If there are still fences that we can try to relax, go back to step 2.

This algorithm ensures that VSync always finds *maximally relaxed* fences.

VSync allows users to plug in their own memory model, such as x86-TSO [6], ARMv8 [2], ARMv7 [3], Power [4], or the Linux Kernel Memory Model [21].

The authors note that checking whether a program is correct given a particular barrier combination can cost a significant amount of time, because the model checker needs to explore all possible execution paths. To speed up finding the correct barrier combination, they speculatively accept certain barrier combinations after a configurable timeout. Only once a maximally relaxed barrier combination is found do they run the model checker without a timeout.

Their model checker uses *stateless model checking* (SMC), a state-of-the-art technique for verifying weak memory models. While SMC is able to enumerate all possible executions of a program, it also requires that programs have a finite state space. Therefore, SMC is not able to handle programs that continuously poll a shared variable in a loop. However, VSync implements a technique, *Await Model Checking* (AMC), which, under certain assumptions, can efficiently detect and return early from these types of executions.

It takes VSync only 0.12 seconds to find the most optimal barrier combination for a ticketlock implementation, approximately 11 minutes for the Linux qspinlock implementation and almost 200 hours for the mutex used in musl libc.

5.2.2 herd, mole and diy

The `herd` tool [12] simulates axiomatic models. These axiomatic models have to be defined using a the "cat" language, which is domain-specific language for defining these axiomatic models. Currently, `herd` supports ARMv8, C11, LKMM and x86. The axiomatic models for these memory models are available online[2].

The `mole` tool [12] detects weak memory patterns in software published as Debian packages. It, however, does not enforce robustness.

The `diy` tool [14] generates litmus tests that reproduce an SC violation. It highlights discrepancies between SC and a weak memory model.

5.3 Fence insertion

There is interesting research available that aims to optimize fence insertion. Morisset and Zappa Nardelli [22] researched how to optimize fence placement for existing fences. Not only does their algorithm capture trivial optimizations such as eliminating two fences without memory accesses in between, it is also able to optimize fences across multiple basic blocks. They implemented their optimization as an LLVM compiler pass and saw performance increase by up to 10%. Their algorithm can potentially be used by Fency to optimize its fence placement.

[2]The sources for the axiomatic models are available at `https://github.com/herd/herdtools7/tree/master/catalogue`

Chapter 6

Conclusion

Fency is a fence insertion tool that enforces M-K robustness on LLVM-based languages. The initial prototype was implemented as a function pass, which limited its usability and made it impossible to implement the fence insertion algorithm as described in [5]. Additionally, it had an ad-hoc alias analysis implementation that did not use LLVM's built-in alias analysis algorithms. Finally, it inserted too many fences in cases where ARMv8 dependencies already ordered instructions.

To guide our research effort, we defined the following research question:

How does dependency-ordered before (dob) analysis and alias analysis affect Fency's ability to insert fences?

We then fixed Fency's shortcomings by reimplementing Fency as a module pass, integrating it with LLVM's alias analysis infrastructure, and adding a new dependency tracking analysis. After completing these improvements, we attempted to answer the research question by comparing Fency v1 with our new version, Fency v2.

We found that the alias analysis had a negligible effect on the number of fences that Fency inserts.

Our reimplementation does fix a soundness issue that was present in Fency v1. Fency v1 is unsound because it ignores memory accesses when LLVM does not annotate them with an affected global memory location. However, this memory access might still access a shared variable despite LLVM being unable to determine the exact memory location that it accesses. Fency v2 fixes this issue by conservatively assuming that a memory access can access *any* memory location when more specific information is unavailable.

More importantly, because it is built on top of LLVM's alias analysis component, any improvement to LLVM's alias analysis implementation will positively affect Fency. Fency will be able to take advantage of a more accurate alias analysis implementation immediately, should it become available.

The DOB analysis did not impact many programs. It largely depended on the number of dependencies in a program.

Restructuring Fency as a module pass paved the way for numerous other improvements. As a result, Fency v2 now supports function calls. We ran Fency on several C/C++ programs to demonstrate this new capability. We noted that while Fency can theoretically analyze large programs, the robustness checking algorithm will quickly become intractable.

6.1 Future work

Future work could focus on four key areas:

- Reducing the MPG. The size of the MPG severely limits Fency's ability to analyze larger programs. Future research may focus on eliminating redundant nodes from the MPG or represent the MPG differently without sacrificing Fency's soundness.
- More efficient fence placement. Fency's fence insertion algorithm, while correct, is very inefficient. One potential improvement is implementing an approach such as in [16]. Here, Boujjani et al. implement a fence insertion algorithm using linear programming. This may help Fency eliminate some fences or optimize the placement of the fences.
- Analyzability of the generated MPG. Currently, Fency can output the MPG as a JSON-encoded text file. This makes it time-consuming to analyze its output. Data visualization techniques could help create an intuitive visualization of the MPG, which would help to debug programs tremendously.
- Implementing support for analyzing larger programs consisting of multiple compilation units. Currently, Fency only supports analyzing a single LLVM module at a time. Extending this support to multiple compilation units should be possible but may require Fency to be rewritten as a standalone binary.

Bibliography

[1] A. Bouajjani, R. Meyer, and E. Möhlmann, "Deciding robustness against total store ordering," in *Automata, Languages and Programming*, L. Aceto, M. Henzinger, and J. Sgall, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 428–440, ISBN: 978-3-642-22012-8.

[2] *Arm architecture reference manual for a-profile architecture*, Arm Limited, 2022.

[3] *Arm architecture reference manual, armv7-a and armv7-r edition*, Arm Limited, 2022.

[4] *Power isa version 2.06 revision b*, IBM, 2010.

[5] S. Chakraborty, "Robustness between weak memory models," in *2021 Formal Methods in Computer Aided Design (FMCAD)*, IEEE, 2021, pp. 173–182.

[6] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "X86-tso: A rigorous and usable programmer's model for x86 multiprocessors," *Commun. ACM*, vol. 53, no. 7, pp. 89–97, 2010, ISSN: 0001-0782. DOI: 10.1145/1785414.1785443.

[7] "Llvm modules." (2022), [Online]. Available: https://llvm.org/doxygen/group__LLVMCCoreModule.html (visited on 11/10/2022).

[8] B. Norris and B. Demsky, "Cdschecker: Checking concurrent data structures written with c/c++ atomics," *SIGPLAN Not.*, vol. 48, no. 10, pp. 131–150, Oct. 2013, ISSN: 0362-1340. DOI: 10.1145/2544173.2509514.

[9] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in weak memory models," in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 258–272, ISBN: 978-3-642-14295-6.

[10] J. Alglave and L. Maranget, "Stability in weak memory models," in *Computer Aided Verification*, G. Gopalakrishnan and S. Qadeer, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 50–66, ISBN: 978-3-642-22110-1.

[11] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl, "Don't sit on the fence: A static analysis approach to automatic fence insertion," in *Computer Aided Verification*, A. Biere and R. Bloem, Eds., Cham: Springer International Publishing, 2014, pp. 508–524, ISBN: 978-3-319-08867-9.

[12] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: Modelling, simulation, testing, and data mining for weak memory," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 2, 2014, ISSN: 0164-0925. DOI: 10.1145/2627752.

[13] J. Alglave, L. Maranget, P. E. McKenney, A. Parri, and A. Stern, "Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel," *SIGPLAN Not.*, vol. 53, no. 2, pp. 405–418, 2018, ISSN: 0362-1340. DOI: 10.1145/3296957.3177156.

[14] J. Alglave, W. Deacon, R. Grisenthwaite, A. Hacquard, and L. Maranget, "Armed cats: Formal concurrency modelling at arm," *ACM Trans. Program. Lang. Syst.*, vol. 43, no. 2, 2021, ISSN: 0164-0925. DOI: 10.1145/3458926.

[15] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: X86-tso," in *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, ser. TPHOLs '09, Munich, Germany: Springer-Verlag, 2009, pp. 391–407, ISBN: 9783642033582. DOI: 10.1007/978-3-642-03359-9_27.

[16] A. Bouajjani, E. Derevenetc, and R. Meyer, "Checking and enforcing robustness against tso," in *Programming Languages and Systems*, M. Felleisen and P. Gardner, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 533–553, ISBN: 978-3-642-37036-6.

[17] "Cliff click's concurrent hashmap." (2022), [Online]. Available: https://github.com/boundary/high-scale-lib/blob/3654434eda00b68d37d22dcd70e4f65db9432d06/src/main/java/org/cliffc/high_scale_lib/NonBlockingHashMap.java (visited on 10/10/2022).

[18] G. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, 1st. Addison-Wesley Professional, 2011, ISBN: 0321773713.

[19] J. Alglave, "A shared memory poetics," Ph.D. dissertation, l'Université Paris 7 - Denis Diderot, 2010.

[20] J. Oberhauser, R. L. d. L. Chehab, D. Behrens, M. Fu, A. Paolillo, L. Oberhauser, K. Bhat, Y. Wen, H. Chen, J. Kim, and V. Vafeiadis, "Vsync: Push-button verification and optimization for synchronization primitives on weak memory models," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021, Virtual, USA: Association for Computing Machinery, 2021, pp. 530–545, ISBN: 9781450383172. DOI: 10.1145/3445814.3446748.

[21] "Linux kernel memory model." (2018), [Online]. Available: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0124r6.html (visited on 10/10/2022).

[22] R. Morisset and F. Zappa Nardelli, "Partially redundant fence elimination for x86, arm, and power processors," in *Proceedings of the 26th International Conference on Compiler Construction*, ser. CC 2017, Austin, TX, USA: Association for Computing Machinery, 2017, pp. 1–10, ISBN: 9781450352338. DOI: 10.1145/3033019.3033021.

Acronyms

CFG Control Flow Graph. 39

DOB Dependency Ordered-Before. 6, 34, 36

GHB Global happens-before. 12

ILP Integer Linear Programming. 38

MPG Memory Pair Graph. 16, 31, 32, 37, 42

NRP Non-Robust Pair. vi, 32, 33, 34, 37

SC Sequential Consistency. 3

TSO Total Store Order. 3

Appendix A

Definitions of the Robustness conditions in Fency

Helper functions

```
1  fn isWriteReadPair((i, j): (MemoryAccess, MemoryAccess)) -> bool;
2      RMWs = CFG.findAll(RMW)
3
4      return isWrite(i) && isRead(j) && && !isSC(i) && !isLL(j)
5          && !(i alwaysAccessesSameMemoryAs j)
6          && ((i, j) neverPassesThroughAnyOf RMWs)
```

`isWriteReadPair` checks whether a pair of memory accesses access a different location without any intermediate RMW.

SC-x86

```
1  fn x86::isSCOrdered((i, j): (MemoryAccess, MemoryAccess)) -> bool:
2      Fences = CFG.findAll(X86::MFENCE)
3
4      return isRead(i) || isWrite(j)
5          || (i alwaysAccessesSameMemoryAs j)
6          || ((i, j) alwaysPassesThroughAnyOf Fences)
```

SC-ARMv8

```
1  fn ARMv8::isSCOrdered((i, j): (MemoryAccess, MemoryAccess)) -> bool:
2      if i alwaysAccessesSameMemoryAs j:
3          return true
4
5      B = CFG.findAll(ARMv8::DMB_ISH)
6          | CFG.findAll(AcquireReadsFrom(i))
7
8      if (i, j) alwaysPassesThroughAnyOf B:
9          return true
10
11     if isStoreRelease(i) && isAcquireLoad(j))
12         || isAcquireLoad(i) || isAcquirePC(i)
13         || isStoreRelease(j):
14         return true
15
16     LoadFences = CFG.findAll(ARMv8::DMB_ISHLD)
```

```
    if isRead(i) && isRead(j)
        && ((i, j) alwaysPassesThroughAnyOf (B | LoadFences)):
        return true

    SameLocationStoreReleases = CFG.findAll(SameLocationStoreRelease(J))
    RWFences = B | LoadFences | SameLocationStoreReleases
    if isRead(i) && isWrite(j)
        && ((i, j) alwaysPassesThroughAnyOf RWFences):
        return true

    StoreFences = CFG.findAll(ARMv8::DMB_ISHST)
    WWFences = B | StoreFences | SameLocationStoreReleases
    if isWrite(i) && isWrite(j)
        && ((i, j) alwaysPassesThroughAnyOf WWFences):
        return true

    return false
```

x86-ARMv8

```
fn ARMv8::isX86Ordered((i, j): (MemoryAccess, MemoryAccess)) -> bool:
    return isWriteReadPair(i, j) || ARMv8::isSCOrdered((i, j))
```

SC-ARMv7

```
fn ARMv7::isSCOrdered((i, j): (MemoryAccess, MemoryAccess)) -> bool:
    Fences = CFG.findAll(ARMv7::DMB)

    return (i alwaysAccessesSameMemoryAs j)
        || ((i, j) alwaysPassesThroughAnyOf Fences)
```

x86-ARMv7

```
fn ARMv7::isX86Ordered((i, j): (MemoryAccess, MemoryAccess)) -> bool:
    return ARMv7::isSCOrdered((i, j)) || isWriteReadPair(Pair)
```

ARMv8-ARMv7

```
fn ARMv7::isARMv8Ordered((i, j): (MemoryAccess, MemoryAccess)) -> bool:
    return ARMv7::isSCOrdered((i, j)) || isWrite(i)
```

Appendix B

Comparing Fency v1 and Fency v2: full results

The full results for the comparison between Fency v1 and Fency v2. The “MPG Nodes” and “MPG Edges” columns contain the number of nodes and edges in the memory pair graph as “v1/v2/difference”. The “NRP”, “F”, and “R” columns contain the number of Non-Robust Pairs, Fences, and whether Fency considers the program M-K robust.

Program	MPG Nodes	MPG Edges	NRP	F	R	Time (ms)
barrier	8/8/0	28/28/0	4/4/0	2/2/0	N/N	0/0/0
cilk-sc	96/104/8	3109/4169/1060	0/0/0	0/0/0	Y/Y	6/68/62
cilk-tso	96/104/8	3109/4169/1060	0/0/0	0/0/0	Y/Y	6/65/59
cldequeue-ra	35/35/0	598/598/0	0/0/0	0/0/0	Y/Y	1/6/5
cldequeue-ra-noloop	16/16/0	122/122/0	0/0/0	0/0/0	Y/Y	0/2/2
cldequeue-sc	35/35/0	598/598/0	3/3/0	2/2/0	N/N	1/5/4
cldequeue-sc-noloop	16/16/0	122/122/0	2/2/0	1/1/0	N/N	0/2/2
cldequeue-tso	35/35/0	598/598/0	1/1/0	1/1/0	N/N	1/5/4
dekker-sc	102/102/0	3762/3762/0	18/18/0	6/6/0	N/N	5/31/26
dekker-tso	102/102/0	3762/3762/0	0/0/0	0/0/0	Y/Y	6/31/25
iriw	9/9/0	27/27/0	0/0/0	0/0/0	Y/Y	0/1/1
lamport-ra	9/13/4	72/156/84	0/0/0	0/0/0	Y/Y	0/1/1
lamport-sc	9/13/4	72/156/84	0/0/0	0/0/0	Y/Y	0/1/1
lamport-tso	9/13/4	72/156/84	0/0/0	0/0/0	Y/Y	0/1/1
lb	2/2/0	2/2/0	0/0/0	0/0/0	Y/Y	0/0/0
lb-2	0/2/2	0/2/2	0/0/0	0/0/0	Y/Y	0/0/0
mp	2/2/0	2/2/0	0/0/0	0/0/0	Y/Y	0/0/0
mutex	0/0/0	0/0/0	0/0/0	0/0/0	Y/Y	0/2/2
nbw	50/50/0	884/884/0	0/0/0	0/0/0	Y/Y	2/10/8
peterson-ra	48/48/0	518/518/0	0/0/0	0/0/0	Y/Y	2/8/6
peterson-ra-b	245/245/0	15468/15468/0	9/5/-4	6/4/-2	N/N	25/197/172
peterson-ra-d	48/48/0	518/518/0	2/2/0	2/2/0	N/N	2/10/8
peterson-sc	48/48/0	518/518/0	10/10/0	2/2/0	N/N	2/8/6
peterson-tso	26/26/0	224/224/0	0/0/0	0/0/0	Y/Y	0/3/3
rcu	178/207/29	11767/15907/4140	45/45/0	4/4/0	N/N	39/903/864
rcu-offline	329/355/26	36937/44833/7896	8/8/0	3/3/0	N/N	230/2783/2553
sb	0/0/0	0/0/0	0/0/0	0/0/0	Y/Y	0/0/0
seqlock	12/15/3	74/136/62	0/0/0	0/0/0	Y/Y	0/2/2
spinlock	36/48/12	724/1552/828	0/0/0	0/0/0	Y/Y	2/15/13
spinlock4	72/96/24	2936/6272/3336	0/0/0	0/0/0	Y/Y	4/56/52
ticketlock	50/60/10	882/1652/770	0/0/0	0/0/0	Y/Y	2/20/18
ticketlock4	100/120/20	3564/6664/3100	0/0/0	0/0/0	Y/Y	6/73/67
usb	6/6/0	10/10/0	2/2/0	2/2/0	N/N	0/0/0

Figure B.1: A comparison between Fency v1 and Fency v2 for SC-x86

Program	MPG Nodes	MPG Edges	NRP	F	R	Time (ms)
barrier	8/8/0	28/28/0	4/4/0	2/2/0	N/N	0/0/0
cilk-sc	232/244/12	19375/42425/23050	73/76/3	8/9/1	N/N	28/201/173
cilk-tso	232/244/12	19375/42425/23050	45/45/0	7/8/1	N/N	28/198/170
cldequeue-ra	50/50/0	1243/1927/684	4/4/0	3/3/0	N/N	2/7/5
cldequeue-ra-noloop	30/30/0	458/774/316	1/1/0	1/1/0	N/N	1/3/2
cldequeue-sc	50/50/0	1243/1927/684	18/17/-1	7/7/0	N/N	2/7/5
cldequeue-sc-noloop	30/30/0	458/774/316	10/9/-1	4/4/0	N/N	1/3/2
cldequeue-tso	50/50/0	1243/1927/684	13/13/0	6/6/0	N/N	2/7/5
dekker-sc	102/102/0	3762/3762/0	41/43/2	7/7/0	N/N	6/39/33
dekker-tso	102/102/0	3762/3762/0	11/11/0	3/3/0	N/N	6/37/31
iriw	9/9/0	27/27/0	5/5/0	4/4/0	N/N	0/1/1
lamport-ra	9/13/4	72/156/84	0/0/0	0/0/0	Y/Y	0/2/2
lamport-sc	9/13/4	72/156/84	0/2/2	0/1/1	Y/N	0/2/2
lamport-tso	9/13/4	72/156/84	0/0/0	0/0/0	Y/Y	0/2/2
lb	2/2/0	2/2/0	2/2/0	2/2/0	N/N	0/0/0
lb-2	0/2/2	0/2/2	0/2/2	0/2/2	Y/N	0/0/0
mp	2/2/0	2/2/0	2/2/0	2/2/0	N/N	0/0/0
mutex	2/4/2	2/12/10	0/0/0	0/0/0	Y/Y	0/1/1
nbw	86/86/0	3022/6118/3096	30/32/2	9/9/0	N/N	9/20/11
peterson-ra	48/48/0	518/518/0	14/14/0	8/8/0	N/N	2/11/9
peterson-ra-b	128/128/0	4060/9652/5592	8/6/-2	6/4/-2	N/N	8/46/38
peterson-ra-d	68/68/0	1236/2864/1628	18/22/4	10/10/0	N/N	3/16/13
peterson-sc	48/48/0	518/518/0	32/32/0	10/10/0	N/N	2/11/9
peterson-tso	26/26/0	224/224/0	7/7/0	4/4/0	N/N	0/3/3
rcu	58/79/21	679/2251/1572	44/62/18	10/11/1	N/N	2/67/65
rcu-offline	193/235/42	6290/30011/23721	43/55/12	14/8/-6	N/N	428/770/342
sb	0/0/0	0/0/0	0/0/0	0/0/0	Y/Y	0/0/0
seqlock	25/30/5	408/808/400	4/4/0	2/3/1	N/N	1/3/2
spinlock	74/90/16	3618/7538/3920	8/12/4	4/6/2	N/N	4/28/24
spinlock4	148/180/32	14572/30316/15744	16/24/8	8/12/4	N/N	13/101/88
ticketlock	72/84/12	1704/5644/3940	4/8/4	2/2/0	N/N	4/25/21
ticketlock4	144/168/24	6864/22712/15848	8/16/8	4/4/0	N/N	11/89/78
usb	18/18/0	82/290/208	10/10/0	4/4/0	N/N	0/1/1

Figure B.2: A comparison between Fency v1 and Fency v2 for SC-ARMv8

Program	MPG Nodes	MPG Edges	NRP	F	R	Time (ms)
barrier	8/8/0	28/28/0	0/0/0	0/0/0	Y/Y	0/0/0
cilk-sc	232/244/12	19375/42425/23050	68/73/5	8/9/1	N/N	28/203/175
cilk-tso	232/244/12	19375/42425/23050	44/44/0	7/8/1	N/N	28/198/170
cldequeue-ra	50/50/0	1243/1927/684	4/4/0	3/3/0	N/N	2/7/5
cldequeue-ra-noloop	30/30/0	458/774/316	1/1/0	1/1/0	N/N	1/3/2
cldequeue-sc	50/50/0	1243/1927/684	16/15/-1	8/8/0	N/N	2/7/5
cldequeue-sc-noloop	30/30/0	458/774/316	9/8/-1	5/5/0	N/N	1/3/2
cldequeue-tso	50/50/0	1243/1927/684	12/12/0	6/6/0	N/N	2/7/5
dekker-sc	102/102/0	3762/3762/0	23/25/2	3/3/0	N/N	5/37/32
dekker-tso	102/102/0	3762/3762/0	11/11/0	3/3/0	N/N	6/37/31
iriw	9/9/0	27/27/0	5/5/0	4/4/0	N/N	0/1/1
lamport-ra	9/13/4	72/156/84	0/0/0	0/0/0	Y/Y	0/2/2
lamport-sc	9/13/4	72/156/84	0/2/2	0/1/1	Y/N	0/2/2
lamport-tso	9/13/4	72/156/84	0/0/0	0/0/0	Y/Y	0/2/2
lb	2/2/0	2/2/0	2/2/0	2/2/0	N/N	0/0/0
lb-2	0/2/2	0/2/2	0/2/2	0/2/2	Y/N	0/0/0
mp	2/2/0	2/2/0	2/2/0	2/2/0	N/N	0/0/0
mutex	2/4/2	2/12/10	0/0/0	0/0/0	Y/Y	0/1/1
nbw	86/86/0	3022/6118/3096	27/29/2	9/9/0	N/N	9/20/11
peterson-ra	48/48/0	518/518/0	14/14/0	8/8/0	N/N	2/11/9
peterson-ra-b	128/128/0	4060/9652/5592	6/6/0	4/4/0	N/N	8/47/39
peterson-ra-d	68/68/0	1236/2864/1628	12/22/10	10/10/0	N/N	3/16/13
peterson-sc	48/48/0	518/518/0	22/22/0	10/10/0	N/N	2/10/8
peterson-tso	26/26/0	224/224/0	7/7/0	4/4/0	N/N	1/3/2
rcu	58/79/21	679/2251/1572	35/53/18	8/9/1	N/N	2/68/66
rcu-offline	193/235/42	6290/30011/23721	36/49/13	14/9/-5	N/N	420/777/357
sb	0/0/0	0/0/0	0/0/0	0/0/0	Y/Y	0/0/0
seqlock	25/30/5	408/808/400	4/4/0	2/3/1	N/N	1/3/2
spinlock	74/90/16	3618/7538/3920	8/12/4	4/6/2	N/N	4/29/25
spinlock4	148/180/32	14572/30316/15744	16/24/8	8/12/4	N/N	13/99/86
ticketlock	72/84/12	1704/5644/3940	4/8/4	2/2/0	N/N	4/25/21
ticketlock4	144/168/24	6864/22712/15848	8/16/8	4/4/0	N/N	12/90/78
usb	18/18/0	82/290/208	8/10/2	4/4/0	N/N	0/1/1

Figure B.3: A comparison between Fency v1 and Fency v2 for x86-ARMv8

Program	MPG Nodes	MPG Edges	NRP	F	R	Time (ms)
barrier	8/8/0	28/28/0	4/4/0	2/2/0	N/N	0/0/0
cilk-sc	244/244/0	20229/42425/22196	27/29/2	7/7/0	N/N	3506/170/-3336
cilk-tso	244/244/0	20229/42425/22196	19/21/2	6/6/0	N/N	3443/172/-3271
cldequeue-ra	65/65/0	2212/3505/1293	5/5/0	3/3/0	N/N	4/8/4
cldequeue-ra-noloop	40/40/0	858/1428/570	3/3/0	2/2/0	N/N	2/4/2
cldequeue-sc	65/65/0	2212/3505/1293	20/20/0	7/7/0	N/N	4/8/4
cldequeue-sc-noloop	40/40/0	858/1428/570	11/11/0	5/5/0	N/N	2/4/2
cldequeue-tso	65/65/0	2212/3505/1293	15/15/0	6/6/0	N/N	4/8/4
dekker-sc	102/102/0	3762/3762/0	46/46/0	8/8/0	N/N	6/34/28
dekker-tso	102/102/0	3762/3762/0	14/14/0	4/4/0	N/N	5/33/28
iriw	9/9/0	27/27/0	5/5/0	4/4/0	N/N	0/1/1
lamport-ra	13/13/0	108/156/48	0/0/0	0/0/0	Y/Y	0/1/1
lamport-sc	13/13/0	108/156/48	0/3/3	1/1/0	Y/N	0/2/2
lamport-tso	13/13/0	108/156/48	0/0/0	0/0/0	Y/Y	0/1/1
lb	2/2/0	2/2/0	2/2/0	2/2/0	N/N	0/0/0
lb-2	0/2/2	0/2/2	0/2/2	2/2/0	Y/N	0/0/0
mp	2/2/0	2/2/0	2/2/0	2/2/0	N/N	0/0/0
mutex	2/4/2	2/12/10	0/0/0	0/0/0	Y/Y	0/1/1
nbw	86/86/0	3022/6118/3096	17/17/0	8/8/0	N/N	8/15/7
peterson-ra	48/48/0	518/518/0	20/20/0	6/6/0	N/N	2/10/8
peterson-ra-b	162/162/0	6592/17710/11118	4/2/-2	2/1/-1	N/N	13/56/43
peterson-ra-d	84/84/0	2054/4868/2814	14/14/0	8/8/0	N/N	4/18/14
peterson-sc	48/48/0	518/518/0	38/38/0	8/8/0	N/N	2/10/8
peterson-tso	26/26/0	224/224/0	10/10/0	6/6/0	N/N	0/3/3
rcu	72/79/7	845/2251/1406	31/47/16	10/10/0	N/N	10/54/44
rcu-offline	217/235/18	7093/30011/22918	15/29/14	11/10/-1	N/N	1017/697/-320
sb	0/0/0	0/0/0	0/0/0	0/0/0	Y/Y	0/0/0
seqlock	30/30/0	493/808/315	2/2/0	2/2/0	N/N	2/3/1
spinlock	90/90/0	4458/7538/3080	0/0/0	0/0/0	Y/Y	198/25/-173
spinlock4	180/180/0	17932/30316/12384	0/0/0	0/0/0	Y/Y	5536/88/-5448
ticketlock	84/84/0	1992/5644/3652	0/0/0	0/0/0	Y/Y	36/22/-14
ticketlock4	168/168/0	8016/22712/14696	0/0/0	0/0/0	Y/Y	846/79/-767
usb	18/18/0	82/290/208	10/10/0	4/4/0	N/N	0/1/1

Figure B.4: A comparison between Fency v1 and Fency v2 for SC-ARMv7

Program	MPG Nodes	MPG Edges	NRP	F	R	Time (ms)
barrier	8/8/0	28/28/0	0/0/0	0/0/0	Y/Y	0/1/1
cilk-sc	244/244/0	20229/42425/22196	24/26/2	7/7/0	N/N	3446/171/-3275
cilk-tso	244/244/0	20229/42425/22196	18/20/2	6/6/0	N/N	3443/170/-3273
cldequeue-ra	65/65/0	2212/3505/1293	5/5/0	3/3/0	N/N	4/9/5
cldequeue-ra-noloop	40/40/0	858/1428/570	3/3/0	2/2/0	N/N	2/5/3
cldequeue-sc	65/65/0	2212/3505/1293	18/18/0	7/7/0	N/N	4/8/4
cldequeue-sc-noloop	40/40/0	858/1428/570	10/10/0	5/5/0	N/N	2/4/2
cldequeue-tso	65/65/0	2212/3505/1293	14/14/0	6/6/0	N/N	4/8/4
dekker-sc	102/102/0	3762/3762/0	28/28/0	4/3/-1	N/N	5/34/29
dekker-tso	102/102/0	3762/3762/0	14/14/0	4/4/0	N/N	5/34/29
iriw	9/9/0	27/27/0	5/5/0	4/4/0	N/N	0/1/1
lamport-ra	13/13/0	108/156/48	0/0/0	0/0/0	Y/Y	0/2/2
lamport-sc	13/13/0	108/156/48	0/3/3	1/1/0	Y/N	0/2/2
lamport-tso	13/13/0	108/156/48	0/0/0	0/0/0	Y/Y	0/2/2
lb	2/2/0	2/2/0	2/2/0	2/2/0	N/N	0/0/0
lb-2	0/2/2	0/2/2	0/2/2	2/2/0	Y/N	0/0/0
mp	2/2/0	2/2/0	2/2/0	2/2/0	N/N	0/0/0
mutex	2/4/2	2/12/10	0/0/0	0/0/0	Y/Y	0/1/1
nbw	86/86/0	3022/6118/3096	17/17/0	8/8/0	N/N	9/15/6
peterson-ra	48/48/0	518/518/0	20/20/0	6/6/0	N/N	2/10/8
peterson-ra-b	162/162/0	6592/17710/11118	2/2/0	2/1/-1	N/N	13/55/42
peterson-ra-d	84/84/0	2054/4868/2814	14/14/0	8/8/0	N/N	4/18/14
peterson-sc	48/48/0	518/518/0	28/28/0	8/8/0	N/N	2/10/8
peterson-tso	26/26/0	224/224/0	10/10/0	6/6/0	N/N	0/3/3
rcu	72/79/7	845/2251/1406	25/41/16	10/10/0	N/N	10/53/43
rcu-offline	217/235/18	7093/30011/22918	13/26/13	9/8/-1	N/N	1009/702/-307
sb	0/0/0	0/0/0	0/0/0	0/0/0	Y/Y	0/0/0
seqlock	30/30/0	493/808/315	2/2/0	2/2/0	N/N	2/3/1
spinlock	90/90/0	4458/7538/3080	0/0/0	0/0/0	Y/Y	199/25/-174
spinlock4	180/180/0	17932/30316/12384	0/0/0	0/0/0	Y/Y	5513/87/-5426
ticketlock	84/84/0	1992/5644/3652	0/0/0	0/0/0	Y/Y	37/22/-15
ticketlock4	168/168/0	8016/22712/14696	0/0/0	0/0/0	Y/Y	867/80/-787
usb	18/18/0	82/290/208	8/10/2	4/4/0	N/N	0/1/1

Figure B.5: A comparison between Fency v1 and Fency v2 for x86-ARMv7

Program	MPG Nodes	MPG Edges	NRP	F	R	Time (ms)
barrier	8/8/0	28/28/0	0/0/0	0/0/0	Y/Y	0/0/0
cilk-sc	244/244/0	20229/42425/22196	20/22/2	7/7/0	N/N	3546/169/-3377
cilk-tso	244/244/0	20229/42425/22196	14/16/2	6/6/0	N/N	3472/171/-3301
cldequeue-ra	65/65/0	2212/3505/1293	4/4/0	3/3/0	N/N	4/8/4
cldequeue-ra-noloop	40/40/0	858/1428/570	2/2/0	2/2/0	N/N	2/4/2
cldequeue-sc	65/65/0	2212/3505/1293	15/15/0	7/7/0	N/N	4/8/4
cldequeue-sc-noloop	40/40/0	858/1428/570	8/8/0	5/5/0	N/N	2/4/2
cldequeue-tso	65/65/0	2212/3505/1293	12/12/0	6/6/0	N/N	4/8/4
dekker-sc	102/102/0	3762/3762/0	28/28/0	4/3/-1	N/N	6/34/28
dekker-tso	102/102/0	3762/3762/0	14/14/0	4/4/0	N/N	5/33/28
iriw	9/9/0	27/27/0	5/5/0	4/4/0	N/N	0/1/1
lamport-ra	13/13/0	108/156/48	0/0/0	0/0/0	Y/Y	0/1/1
lamport-sc	13/13/0	108/156/48	0/3/3	1/1/0	Y/N	0/2/2
lamport-tso	13/13/0	108/156/48	0/0/0	0/0/0	Y/Y	0/1/1
lb	2/2/0	2/2/0	2/2/0	2/2/0	N/N	0/0/0
lb-2	0/2/2	0/2/2	0/2/2	2/2/0	Y/N	0/0/0
mp	2/2/0	2/2/0	1/1/0	1/1/0	N/N	0/0/0
mutex	2/4/2	2/12/10	0/0/0	0/0/0	Y/Y	0/1/1
nbw	86/86/0	3022/6118/3096	15/15/0	7/7/0	N/N	9/16/7
peterson-ra	48/48/0	518/518/0	18/18/0	6/6/0	N/N	2/10/8
peterson-ra-b	162/162/0	6592/17710/11118	0/0/0	0/0/0	Y/Y	13/66/53
peterson-ra-d	84/84/0	2054/4868/2814	10/10/0	6/6/0	N/N	4/19/15
peterson-sc	48/48/0	518/518/0	18/18/0	6/6/0	N/N	2/9/7
peterson-tso	26/26/0	224/224/0	8/8/0	4/4/0	N/N	0/3/3
rcu	72/79/7	845/2251/1406	4/20/16	4/4/0	N/N	10/53/43
rcu-offline	217/235/18	7093/30011/22918	9/23/14	6/6/0	N/N	1009/698/-311
sb	0/0/0	0/0/0	0/0/0	0/0/0	Y/Y	0/0/0
seqlock	30/30/0	493/808/315	2/2/0	2/2/0	N/N	2/3/1
spinlock	90/90/0	4458/7538/3080	0/0/0	0/0/0	Y/Y	200/25/-175
spinlock4	180/180/0	17932/30316/12384	0/0/0	0/0/0	Y/Y	5378/87/-5291
ticketlock	84/84/0	1992/5644/3652	0/0/0	0/0/0	Y/Y	37/22/-15
ticketlock4	168/168/0	8016/22712/14696	0/0/0	0/0/0	Y/Y	866/79/-787
usb	18/18/0	82/290/208	2/2/0	2/2/0	N/N	0/1/1

Figure B.6: A comparison between Fency v1 and Fency v2 for ARMv8-ARMv7