Partially Redundant Fence Elimination for x86, ARM and Power Processors

Robin M., Francesco Z.

Presented by
Akshay Gopalakrishnan

June 5, 2025

Introduction

- Usage of memory Fences/Barriers are expensive.
- Compiling programs to hardware instructions want to avoid too many uses of fences.
- Fences can be redundant on compilation, thereby existing fence elimination optimizations.
- However, existing ones are either too constrained, or are not proven to be correct.

Contributions

- This paper proposes a more aggressive Fence optimization using Partial Redundancy Elimination technique used in Compiler Optimizations.
- The source language is C11, target hardware are x86, ARM and Power.
- Key idea is to represent fences and program instructions as a control flow graph with weighted edges.
- This program graph is converted to a min cut problem, wherein the min-cut determines the location where fences are required.
- Using this information, other fences in the program can be removed.

Motivating Example

Trivial

```
r = x.load(acquire);
y.store(release, 42);
```

Figure: C program.

```
r = x;
dmb ish; // introduced by the acquire access
dmb ish; // introduced by the release access
y = 42;
```

Figure: Mapping with ARM Fences.

```
r = x.load(acquire);
y.store(release, 42);
```

Figure: C program.

```
r = x;
dmb ish; // introduced by the acquire access
dmb ish; // introduced by the release access
y = 42;
```

Figure: Mapping with ARM Fences.

The example mapping here depicts the mapping of read/writes in C11 style to ARM equivalent memory access. In that sense, the semantics of a load-acquire requires adding fence after an ARM load. Whereas the store-release requires adding fence before an ARM store. The fence used in essence ensures the semantics mentioned in `https://developer.arm.com/documentation/dui0489/c/arm-and-thumb-instructions/miscellaneous-instructions/dmb--dsb--and-isb` TLDR, *dmb ish; dmb ish* is equivalent to having just one *dmgb ish*.

Motivating Example: Slightly Complex

```
int i = x.load(memory_order_seq_cst);
if (foo())
  z = 42;
y.store(1, memory_order_release);
```

Figure: C program.

```
int i = x;
dmb ish;
bool a = foo();
if (a)
  z = 42;
dmb ish;
y = 1;
```

```
int i = x;
dmb ish;
bool a = foo();
if (a) {
  z = 42;
  dmb ish;
}
y = 1;
```

Figure: Two Possible Mappings with ARM Fences.

Motivating Example: Slightly Complex

```
int i = x.load(memory_order_seq_cst);
if (foo())
  z = 42;
y.store(1, memory_order_release);
```

Figure: C program.

```
int i = x;
dmb ish;
bool a = foo();
if (a)
  z = 42;
dmb ish;
y = 1;
```

```
int i = x;
dmb ish;
bool a = foo();
if (a) {
  z = 42;
  dmb ish;
}
y = 1;
```

Figure: Two Possible Mappings with ARM Fences.

In this example, depending on whether the conditional is satisfied, we can have either two consecutive *dmb ish* (on true branch) or one *dmb ish*, like the mapping on the right. But what we get is actually the mapping on the left, if we apply the mapping from memory accesses to ARM instructions naively.

Motivating Example: Non-Trivial

```
int i = x.load(seq_cst);
for (; i > 0; --i) {
  y.store(i, seq_cst);
}
return;
```

Figure: C program.

```
int i = x.load(seq_cst);
for (; i > 0; --i) {
  y.store(i, seq_cst);
}
return;
```

Figure: C program.

Here is an example with sequentially consistent memory accesses. As per the ARM mapping, a sequentially consistent load must be mapped to a memory load followed by *dmb ish*. Whereas for a sequentially consistent store, it is mapped to first a *dmb ish* followed by the memory store, and finally again a *dmb ish*.

```
int i = x;
dmb ish;
loop:
  if (i > 0) {
    dmb ish;
    y = i;
    dmb ish;
    --i;
    goto loop;
} else {
    return;
}
```

Figure: Naive Mapping

```
i = 2;  dmb ish;  dmb ish;  y = 2;  dmb ish;  i = 1;
  dmb ish;  dmb ish;  y = 1;  dmb ish;  i = 0;  return
```

Figure: Instructions Executed for $x = 2$

Partially Redundant Fence Elimination for x86, ARM and Power Processors

```
int i = x;
dmb ish;
loop:
  if (i > 0) {
    dmb ish;
    y = i;
    dmb ish;
    --i;
    goto loop;
} else {
    return;
}
```

Figure: Naive Mapping

```
i = 2;  dmb ish;  dmb ish;  y = 2;  dmb ish;  i = 1;
  dmb ish;  dmb ish;  y = 1;  dmb ish;  i = 0;  return
```

Figure: Instructions Executed for $x = 2$

Using the mapping rules, here is the ARM code. However, notice that for an example execution where $x = 2$, we see the unnecessary fence consecutive pairs.

```
int i = x;
loop:
  dmb ish;
  if (i > 0) {
    y = i;
    --i;
    goto loop;
  } else {
    return;
}
```

Figure: Optimal Mapping

```
i = 2;  dmb ish;  y = 2;  i = 1;
  dmb ish;  y = 1;  i = 0; dmb ish; return
```

Figure: Instructions Executed for $x = 2$

Partially Redundant Fence Elimination for x86, ARM and Power Processors

```
int i = x;
loop:
  dmb ish;
  if (i > 0) {
    y = i;
    --i;
    goto loop;
  } else {
    return;
}
```

Figure: Optimal Mapping

```
i = 2;  dmb ish;  y = 2;  i = 1;
  dmb ish;  y = 1;  i = 0; dmb ish; return
```

Figure: Instructions Executed for $x = 2$

What we want is a mapping like this. The hint is that first, we need the read to x be followed by a fence. This will happen in the first iteration of the loop body, before checking the loop condition. After that, a write to y is okay, as it is preceded by the fence already. Since i is thread-local, the fence placement w.r.t. i is irrelevant. Again, in the next iteration, before we check the loop condition, *dmb ish* is there, satisfying our requirement for mapping sequentially consistent load.

Key Idea

- Represent fences as a relation between instructions.
- Correlate fence elimination via Partial Redundancy Elimination (PRE).
- Solve the PRE problem via min-cut problem, identifying minimal fence relations required.
- Remove the rest of the relations.

First Step: Use Herding Cats Model

- Represent program as a control flow graph.
- Nodes are all the memory accesses.
- Edges are control flow and fences.
- Fence edges are from prior memory access to later memory access.

Second Step: Correlate with PRE

- PRE problem identifies which computations are redundant.
- Treat fences as computations.
- Treat the memory events after fence as a computation that requires the fence.
- The PRE problem, then will identify the minimal fences required so that in every control flow to a given memory access, a fence is executed.

Third Step: PRE Solve as Min Cut

- Connect the control flow graph to PRE via finding Min Cut.

For each fence instruction in the program,

- connect all the placements before all memory accesses that precede the fence to *Source*;
- connect all the placements after all memory accesses that follow the fence to *Sink*;
- delete the fence instruction.

- The min cut is computed of the resulting graph.
- Min cut indicates where the fences are required.

Fourth Step: Proving Correctness

- Rely on the Axiomatic Specification of Hardware Models (Herding Cats).
- Programs mapped to candidate executions using per-thread sequential semantics.
- Candidate Executions are filtered via respective hardware model, giving consistent executions.

Consider *Transform Function* as our PRE algorithm applied on program P, transforming it to P'.

Lemma 1. *For any candidate execution X' of a program P' obtained by applying* `TransformFunction` *to a program P, there is a candidate execution X of P with* fences$(X) \subseteq$ fences(X')*, and every other part of X is the same as X' (including the events, the* po *and dependency relations, and the rf and* co *relations).*

Corollary 1. *For any execution E' of a program P' obtained by applying* `TransformFunction` *to a program P, there is an execution E of P with* hb$(E) \subseteq$ hb(E') *and* prop$(E) \subseteq$ prop(E')*, and every other part of E is the same as E'.*

Theorem 1. *The transformation* `TransformFunction` *is correct.*