Partially Redundant Fence Elimination for x86, ARM and Power Processors

Robin M., Francesco Z.

Presented by
Akshay Gopalakrishnan

June 15, 2025

Introduction

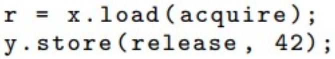
- Usage of memory Fences/Barriers are expensive.
- Compiling programs to hardware instructions want to avoid too many uses of fences.
- Fences can be redundant on compilation, thereby existing fence elimination optimizations.
- However, existing ones are either too constrained, or are not proven to be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

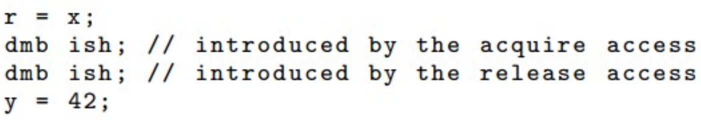
Motivating Example

Trivial



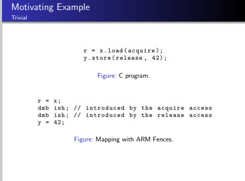
```
r = x.load(acquire);
y.store(release, 42);
```

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.



Motivating Example

Trivial

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

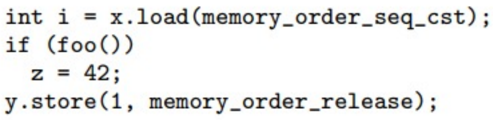
Figure: C program.

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

Figure: Mapping with ARM Fences.

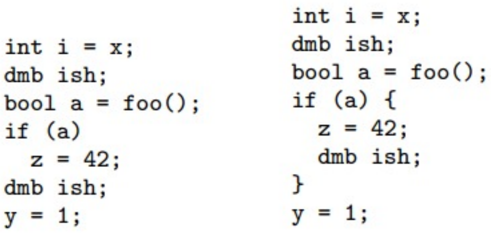
The example mapping here depicts the mapping of read/writes in C11 style to ARM equivalent memory access. In that sense, the semantics of a load-acquire requires adding fence after an ARM load. Whereas the store-release requires adding fence before an ARM store. The fence used in essence ensures the 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);
```

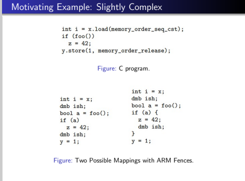
Figure: C program.



```
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.

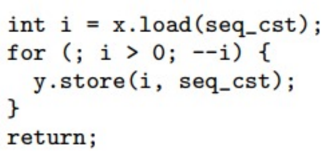
```
int i = x;
dmb ish;
bool a = foo();
if (a)
  z = 42;
dmb ish;
y = 1;
```

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

Figure: Two Possible Mappings with ARM Fences.

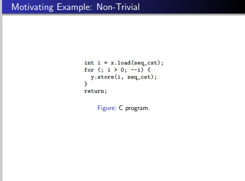
In this example, depending on whether the conditional is satisfied, we can have either two consecutive *dmb ish* (on true branch) or one *dmb ish*, like the mapping on the right. But what we get is actually the mapping on the left, if we apply the mapping from memory accesses to ARM instructions naively.

Motivating Example: Non-Trivial



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

Figure: C program.

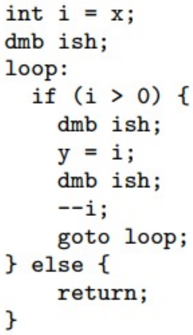


Motivating Example: Non-Trivial

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

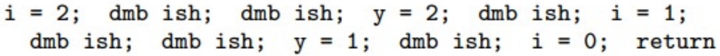
Figure: C program.

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*.



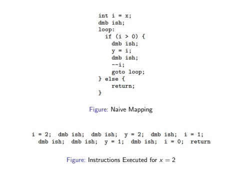
```
int i = x;
dmb ish;
loop:
  if (i > 0) {
    dmb ish;
    y = i;
    dmb ish;
    --i;
    goto loop;
} else {
    return;
}
```

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 $x = 2$



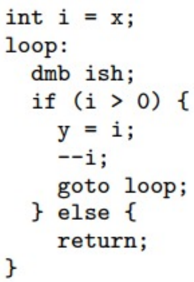
```
int i = x;
dmb ish;
loop:
  if (i > 0) {
    dmb ish;
    y = i;
    dmb ish;
    --i;
    goto loop;
} else {
    return;
}
```

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
```

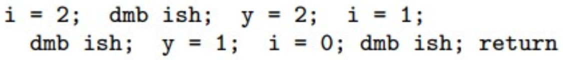
Figure: Instructions Executed for $x = 2$

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.



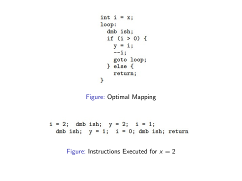
```
int i = x;
loop:
  dmb ish;
  if (i > 0) {
    y = i;
    --i;
    goto loop;
  } else {
    return;
}
```

Figure: Optimal Mapping



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

Figure: Instructions Executed for $x = 2$



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

Figure: Optimal Mapping

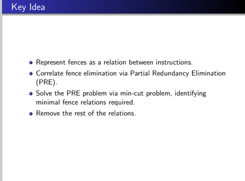
```
i = 2;  dmb ish;  y = 2;  i = 1;
  dmb ish;  y = 1;  i = 0; dmb ish; return
```

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.



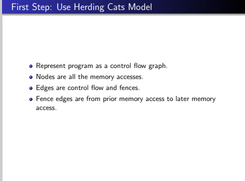
Key Idea

- 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.

The motivation is taken from Herding Cats paper, where fences are represented not as instructions, but as relations between memory events. Intuitively this makes sense, as a fence influences the order in which memory actions take place. Surprisingly, viewing these relations as graph edges where nodes are memory events is isomorphic to a PRE representation of a control flow graph. In this setting the information flow is instead the fences, which are edges.

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.



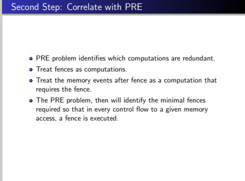
First Step: Use Herding Cats Model

- 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.

Program is converted to a graph/execution trace, where nodes are memory events and edges are simply control flow and fences. Fence edges, as intuitively understood, is from the memory event just before it to memory event just after it.

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.

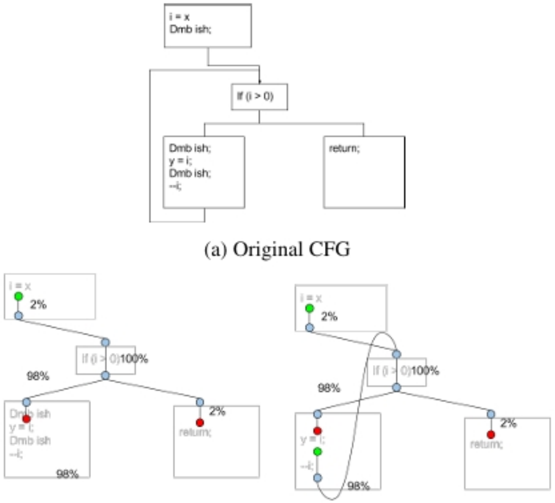


Second Step: Correlate with PRE

- 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.

The correlation with PRE is ingenious. Intuitively, the job of PRE is, given a computation and a target point, ensure every control flow path to the target point has that computation, while removing any repetition of computing it. In our case, this computation is a fence, and the target point is the memory event just after the fence. Thus, the PRE problem now is to eliminate redundant fences, while still ensuring at least one fence exists which reaches desired target memory event across every control flow path.

Example



(a) Original CFG

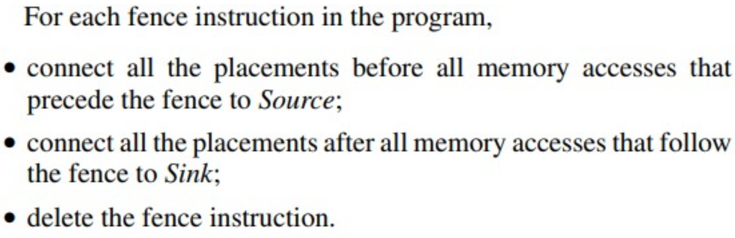
(b) After processing the first fence

(c) After processing all fences

Figure: Example Control Flow Graph with its PRE centric representation

Third Step: PRE Solve as Min Cut

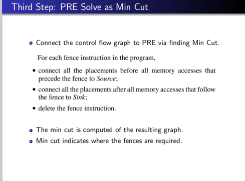
- Connect the control flow graph to PRE via finding Min Cut.



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.



Third Step: PRE Solve as Min Cut

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

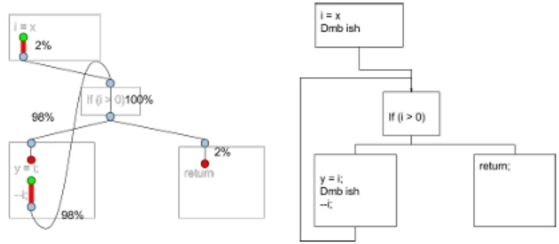
  For each fence instruction in the program,

  - 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.

Now comes the correlating with the control flow graph with PRE. There is an ingenious way of solving the PRE problem as a min cut problem, where the cut represents the locations/computations that must be preserved, to ensure flow to desired point. Thus, every desired point can be connected to a sink, and every source point, i.e. event just before fence can be connected to a source. Now all we have to do is find the min cut and thereby identifying locations where the fences must remain/inserted.

Example



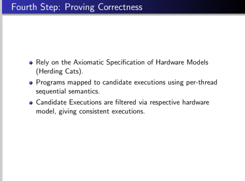
(d) Example of a min-cut

(e) Resulting CFG

Figure: Example Min Cut with the optimized Control Flow Graph.

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.

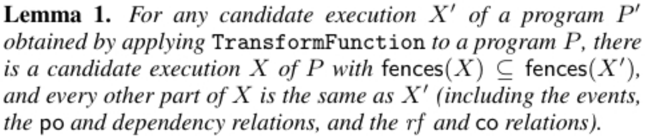


Fourth Step: Proving Correctness

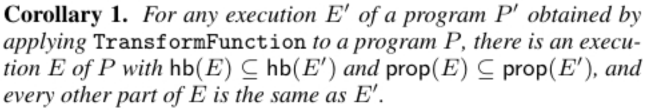
- Rely on the Axiomatic Specification of Hardware Models (Herding Cats).
- Programs mapped to candidate executions using per-thread sequential semantics.
- Candidate Executions are filtered via respective hardware model, giving consistent executions.

Per-thread sequential semantics dictates what read/write values are expected out of memory read/writes for a specific thread execution. The resultant graph for all such threads together will be a collection of events (graph nodes) with the syntactic order (as edges). Connecting each read value with appropriate write value results in a well-formed candidate execution; meaning to say that this execution seems to be possible via the concrete values. These executions are then filtered by the underlying memory model, which are rules over the shape of execution graphs reprsenting a program execution.

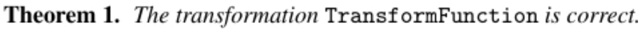
Consider *Transform Function* as our PRE algorithm applied on program P, transforming it to P'.



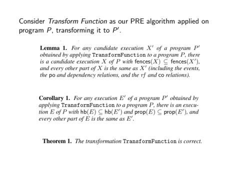
Lemma 1. *For any candidate execution X' of a program P' obtained by applying* `TransformFunction` *to a program P, there is a candidate execution X of P with* $\mathsf{fences}(X) \subseteq \mathsf{fences}(X')$, *and every other part of X is the same as X' (including the events, the* po *and dependency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



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



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* $\mathsf{fences}(X) \subseteq \mathsf{fences}(X')$, *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* $\mathsf{hb}(E) \subseteq \mathsf{hb}(E')$ *and* $\mathsf{prop}(E) \subseteq \mathsf{prop}(E')$, *and every other part of E is the same as E'.*

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

The first lemma is on candidate executions in general, simply stating that applying the optimization on them ensures there exists a candidate execution derived from the optimized program which has the same graph ordering among events with the exception that there are fewer fences, or fence edges. This is simply followed by a Corollary that uses the result on executions of the program, stating that the relations that are used to describe the consistent execution strictly increase, thereby making sure that executions from the transformed program are contained by that of the source. This is stated by the final theorem of correctness. As a note, this kind of proof leverages the monotonicity existing within the memory model considered (ARM, TSO, POWER).

Future Work/ Limitations

- Pending Interprocedural Analyses - Current algorithm only uses information within the same procedure to add/remove fences.
- Not Leverage C11 Specific Accesses - The algorithm is implemented on an IR that has already inserted fences for specific accesses of C11.
- Conservative weight estimates - Fence weights are determined via profiling information, which ultimately affects the calculation of Min Cut for PRE. This leads to a better solution, however, it is conservative.

Further reading from Paper

- Testing results.
- Corner cases of optimizations.

Thank you.