Partially Redundant Fence Elimination for x86, ARM, and Power Processors

Robin Morisset Francesco Zappa Nardelli
ENS & Inria, France
robin.morisset@normalesup.org francesco.zappa_nardelli@inria.fr

Abstract

We show how partial redundancy elimination (PRE) can be instantiated to perform *provably correct* fence elimination for multi-threaded programs running on top of the x86, ARM and IBM Power relaxed memory models. We have implemented our algorithm in the backends of the `LLVM` compiler infrastructure. The optimisation does not induce an observable overhead at compile-time and can result in up-to 10% speedup on some benchmarks.

Keywords Compiler optimisations, Shared-memory concurrency, Weak-memory models

Categories and Subject Descriptors C.1.2 [*Multiple Data Stream Architectures (Multiprocessors)*]: Parallel Processors; D.3.4 [*Programming Languages*]: Optimization; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]

1. Motivation for backend fence elimination

Modern multicore and multiprocessor architectures exhibit *relaxed memory*, exposing behaviour that arises from hardware optimisations to the programmer. Each architecture defines its own *memory model*, stating precisely which writes can be observed by each read. Memory models can be as simple as x86-TSO, that governs the x86 multiprocessors, in which only thread-local write-buffers can be observed [20], or as complicated as those of ARM and IBM Power multiprocessors, in which writes propagates to different processors in different orders [18]. In addition, each architecture offers different primitives with different semantics to impose constraints on relaxed behaviours.

The C11 and C++11 standards attempted to provide a language design that enables *portable* concurrent programming. In C11/C++11 programs that exhibit racy memory accesses are undefined while well-synchronised ones must exhibit only sequentially consistent (e.g. interleaving) behaviours. It is well known that this design can be implemented efficiently as it enables a wide range of compiler and backend optimisation [24]. However, to enable the implementation of *portable* low-level concurrent algorithms, the C11 and C++11 standards also introduce an escape mechanism called *low-level atomics*: low-level atomic accesses do not race with each other and have their semantics specified by a *memory order attribute*. The memory order attributes come in several strengths. The `seq_cst` attribute requires the compiler to guarantee a sequentially consistent semantics for the access, the `release/acquire` and `release/consume` attributes enable a fast implementation of the message-passing idiom. Finally the `relaxed` attribute identifies accesses that ought to be mapped directly to hardware accesses.

While the design of `relaxed` accesses should not constrain compiler and hardware optimisations apart for ensuring coherence (even if the current formulation of the standard is unsatisfactory [22]), all other atomic accesses enforce their semantics by restricting allowed compiler and hardware optimisations [17]. To restrict hardware optimisations the compiler inserts *memory fences* and other synchronisation instructions in correspondence of atomic accesses; mappings for C11/C++11 memory accesses to major architectures have been proposed [1] and in several cases formally proved correct [5, 6]. We report the mappings relevant for our work in Figure 1. In [5] it is shown that these mappings are *locally optimal*: if the compiler translates arbitrary C11/C++11 code by *naively mapping* memory accesses to assembly instructions following these tables, weakening any of the clauses (e.g. replacing a `hwsync` by a weaker `lwsync`) yields an unsound scheme.

This result however does not preclude that for some programs an optimising compiler can improve the hardware fence placement given by the mappings. So the following code:

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

gets translated to the following ARMv7 pseudo-assembly:

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

We will give a precise account of the ARM memory model in Section 3.1, but it is easy to show that whenever there are two consecutive `dmb ish` instructions, the second acts as a no-op. As such it can safely be optimised away without affecting the semantics of the program in an arbitrary concurrent context. This simple peephole optimisation has already been implemented in the LLVM ARM backend [9]. However, more generally, the compiler backend can rely on *thread-local data-flow* informations to identify occasions to optimise the generated code while preserving the expected semantics. Consider for instance the snippet of C11 code below:

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

and the ARMv7 pseudo-assembly generated by the mappings for atomic accesses:

C++11 Operation	x86 Implementation	Power implementation	ARMv7 implementation
Non-atomic Load	`mov`	`ld`	`ldr`
Load Relaxed	`mov`	`ld`	`ldr`
Load Consume	`mov`	`ld`; `lwsync`	`ldr`; `dmb ish`
Load Acquire	`mov`	`ld`; `lwsync`	`ldr`; `dmb ish`
Load Seq_Cst	`mov`	`hwsync`; `ld`; `lwsync`	`ldr`; `dmb ish`
Store Relaxed	`mov`	`st`	`str`
Store Release	`mov`	`lwsync`; `st`	`dmb ish`; `str`
Store Seq_Cst	`xchg`	`hwsync`; `st`	`dmb ish`; `str`; `dmb ish`
Fence Rel/Acq		`lwsync`	`dmb ish`
Fence Seq_Cst	`mfence`	`hwsync`	`dmb ish`

Figure 1: C11/C++11 memory-access mappings used by LLVM 4.0 for x86, Power, ARMv7 architectures

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

In the above code neither of the two `dmb ish` barriers can be eliminated. Again, we will make this precise later, but intuitively, the former fence ensures that the load of `x` is not reordered with the store of `z`, and the latter fence ensures that the write to `z` is visible by all processors before the write to `y` is performed. However this fence placement is *not optimal*: when `a` is false two barriers in a row would be executed while one would suffice (to prevent the reordering of the load of `x` with the write of `y`), and the pseudo-code could be optimised as:

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

Observe that the optimised pseudo-assembly cannot be obtained by applying a source-to-source transformation to the original C11 program and then applying the reference mappings. This suggests that a class of fence optimisations can, and must, be implemented in an architecture-specific backend of a compiler, exploiting the precise semantics of the target architecture. However, apart from some peephole optimisations as the already cited one ([9]), at the time of writing, mainstream compilers are conservative about optimising atomic accesses or reordering atomic and non-atomic accesses.

Contributions In this paper we show how to instantiate Partial Redundancy Elimination (PRE) to implement an efficient and provably correct fence optimisation algorithm for the x86, ARM and IBM Power architectures, improving and generalising the algorithm proposed for x86 in [23]. Although these architectures implement widely different memory models, we identified a key property required for the correctness of our algorithm and we prove formally that our algorithm is correct with respect the memory model of the three architectures. We have implemented our optimisation algorithm in the x86, ARM, and Power backends of the LLVM Compiler Infrastructure [14]. We evaluate it on some benchmarks, including a signal processing application from the StreamIt suite, on which we observe a speedup up-to 10% on the Power architecture.

The paper is structured as follows:

- in Section 2 we illustrate our algorithm on a running example, and describe its implementation in the ARM LLVM backend;
- in Section 3 we recall the x86, ARM and Power memory models, as formalised in [4], and we prove the correctness of our algorithm;
- in Section 4 we discuss how to tune the algorithm to the IBM Power and x86 architectures, and in Section 5 we put it at work on several benchmarks;
- in Sections 6 and 7 we report on related works and discuss future perspectives.

2. A PRE-inspired fence elimination algorithm

The design of our fence elimination algorithm is guided by the representation of fences in the "Herding Cats" framework [4] for weak memory models. This framework, detailed in the next section, represents all the allowed behaviours of a program in terms of the sets of the memory events (atomic read or writes of shared memory locations) that each thread can perform, and additional relations between these events. Fence instructions do not generate events of their own, but are represented as a relation between events. This relation relates all pairs of memory accesses such that a fence instruction is executed in between by the sequential semantics of each thread. This implies that any program transformation that moves/inserts/removes fence instructions while preserving the fence relation between events is trivially correct. We can even go a bit further. The model is monotonic with respect to the fence relation: adding pairs of events to the fence relation can only reduce the allowed concurrent behaviours of a program. In turn any program transformation modifying the placement of fence instructions will be trivially correct provided that it does not remove pairs from the fence relation between events.

Consider now the snippet of C11/C++11 code below, where `x` and `y` are global, potentially shared, atomic variables and `i` is a local (not shared) variable:

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

and the ARMv7 pseudo-code[1] generated by desugaring the loop and applying the mappings for the atomic accesses:

[1] Our optimisation runs on the LLVM IR at the frontier between the middle end and the architecture specific backend. This IR extends the LLVM IR with intrinsics to represent architecture specific memory fences; the pseudo-code we rely-on carries exactly the same informations as the IR over which our implementation works.

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

A sequential execution of this code with `x = 2` executes the following instructions:

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

resulting in two events related by the fence relation (omitting the fence arrows from the entry point and to the return point):

If one fence is removed and the program transformed as in:

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

then a sequential execution of this code with `x = 2` executes the following instructions:

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

and both the events generated and the fence relation remain unchanged, and we are guaranteed that the two programs will have the same behaviours in any concurrent context. However, in every run, the latter will execute fewer, potentially expensive, `dmb ish` instructions.

2.1 Leveraging PRE

At a closer look, our problem is reminiscent of the *Partial Redundancy Elimination* (PRE) optimisation. Given a computation that happens at least twice in the source file (say `x+y`), PRE tries to insert, move or delete instances of the computation, while enforcing the property that there is still at least one such instance on each path between the definitions of its operands (say `x` and `y`) and its uses. Similarly, we want to insert, move or delete fences such that there is still at least one fence on each path between memory accesses that were before a fence and those that were after a fence in the original program.

How to implement PRE is a well-studied problem. Approaches can be classified into *conservative* (roughly speaking, the amount of computation the program does cannot be increased) and *speculative*. Speculative PRE (SPRE) is allowed to introduce some extra computation on rarely taken paths in the program to remove additional computations from the more common paths. We follow this latter approach because it can offer better results [12] in presence of accurate profiling information. Such *profile-guided optimisations* require compiling the program under consideration twice: first without optimisations, then a second time after having executed the program on a representative benchmark with profiling instrumentation on. Our algorithm can be adapted to avoid this complex process at a cost in performance, as discussed in Section 7.

We build on the elegant algorithm for SPRE presented in [19] (later generalised to conservative PRE in [25]), based on solving a *min-cut* problem. Given a directed weighted graph (V, E) with two special vertices *Source* and *Sink*, the *min-cut* problem consists in finding a set of edges $C \subseteq V$ such that there is no path from *Source* to *Sink* through $E \backslash C$ and C has the smallest possible weight.

Following [19], our algorithm first builds a graph from the SSA control-flow graph internally used by LLVM. In the built graph, nodes identify all program placements before and after each instructions and there is an edge from after instruction A to before instruction B with weight w if control went directly from A to B w times in the profiling information. Two special nodes are added, called *Source* and *Sink*. Edges with weight ∞ are added from the *Source* node and to the *Sink* node, following the strategy below.

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.

Once all fences have been deleted, a min-cut of the resulting graph is computed: the min-cut identifies all the places where fences must be inserted to guarantee that there is still a fence across every computation path between memory accesses originally separated by a fence. Each edge in the min-cut identifies two placements in the program, the former after a memory access and the latter before another memory access: the algorithm inserts a fence instruction just before the latter memory access.

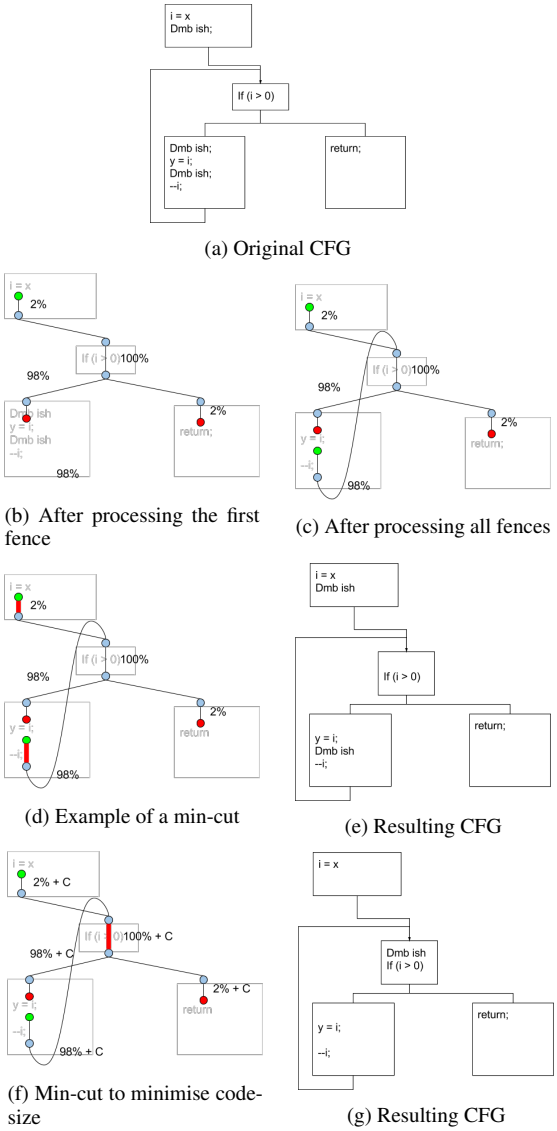
Algorithm 1 reports the details of our implementation in LLVM, the entry point is `TransformFunction`. Our optimisation runs after all the middle-end transformations; at this level, the IR is in SSA form but includes the architecture-specific intrinsics for fence instructions which have been inserted by the expansion of the C11/C++11 atomic memory accesses. It differs slightly from the description above inasmuch. Instead of creating two nodes for every instruction, it lazily constructs nodes (with the LLVM `GetNode` functions) at the beginning and end of basic blocks, and before and after memory accesses. Considering only these points is correct, because fences commute with instructions that do not affect memory. The `GetNode` functions refers to a hash table that maps locations in the code to nodes: if a node has already been generated for a location, a pointer to that same node is returned, it is not duplicated.

2.2 The algorithm on a running example

Let us illustrate our algorithm on the code snippet shown at the beginning of this section, assuming that some profiling data are available. Its control-flow graph (CFG) is shown in Figure 2a.

In the first phase, the algorithm considers each fence in order and builds an ad-hoc flow graph that for each fence connects all the memory accesses before the fence to all the memory accesses after the fence. The terms "before" and "after" are relative to the original control-flow of the function.

In the running example, after processing the first fence, we get the flow-graph in Figure 2b. The node in green is connected to the *Source* and the nodes in red are connected to the *Sink*. The percentages next to the edges are weights, set proportionally to the frequency of execution of these edges in the profiling data. After processing all the fences, we obtain the flow-graph in Figure 2c.



(a) Original CFG

(b) After processing the first fence

(c) After processing all fences

(d) Example of a min-cut

(e) Resulting CFG

(f) Min-cut to minimise code-size

(g) Resulting CFG

Figure 2: Running example of the fence elimination algorithm

At this point, all fences are removed from the program and the min-cut is computed. Figure 2d shows one possible min-cut on this graph (through bold red edges).

In the second phase fences are inserted on each edge that is part of the cut. The resulting optimised CFG is shown in Figure 2e.

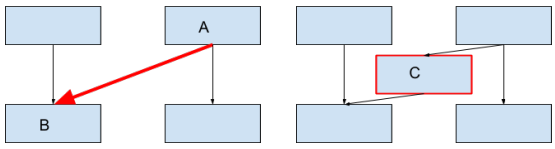
This algorithm generates a solution that is optimal in terms of the number of executions of fences at runtime; in some cases we might instead minimize the code size. As explained Section 7, we can modify this by modifying artificially the weights on the edges. In particular, adding a tiny constant C to each edge breaks ties among possible min-cuts according to the number of fences in the generated code. This approach is shown in Figure 2f, with the resulting CFG in Figure 2g; observe that here the fence in the first block has been removed.

2.3 Corner cases

Function boundaries on the control-flow graph Our program transformation is invoked on each function, and currently does not perform any whole program analysis. In the previous paragraphs

```
1  Function TransformFunction(fun)
2      for f a fence in fun do
3          nodeBeforeFence ←MakeGraphUpwards(f)
4          nodeAfterFence ←MakeGraphDownwards(f)
5          if nodeBeforeFence ≠ NULL && nodeAfterFence
             ≠ NULL then
6              MakeEdge (nodeBeforeFence,
                 nodeAfterFence)
7          DeleteFence (f)
8      cuts ←ComputeMinCut ()
9      foreach location ∈ cuts do
10         InsertFenceAt(location)
11 Function MakeGraphUpwards(root)
12     basicBlock ←GetBasicBlock(root)
13     for inst an instruction before root in basicBlock, going
         upwards do
14         if inst a memory access then
15             node ←GetNodeAfter(inst)
16             ConnectSource(node)
17             return node
18     node ←GetNodeAtBeginning(basicBlock)
19     if basicBlock is first block in function then
20         ConnectSource(node)
21         return node
22     for basicBlock2 a predecessor of basicBlock do
23         node2 ←GetNodeAtEnd(basicBlock2)
24         inst2 ←GetLastInst(basicBlock2)
25         node3 ←MakeGraphUpwards(inst2)
26         if node3 ≠ NULL then
27             MakeEdge(node3, node2)
28             MakeEdge(node2, node)
29 Function MakeGraphDownwards(root)
30     basicBlock ←GetBasicBlock(root)
31     for inst an instruction after root in basicBlock, going
         downwards do
32         if inst a memory access or a return instruction then
33             node ←GetNodeBefore(inst)
34             ConnectSink(node)
35             return node
36     node ←GetNodeAtEnd(basicBlock)
37     for basicBlock2 a successor of basicBlock do
38         node2 ←GetNodeAtBeginning(basicBlock2)
39         inst2 ←GetFirstInst(basicBlock2)
40         node3 ←MakeGraphDownwards(inst2)
41         if node3 ≠ NULL then
42             MakeEdge(node, node2)
43             MakeEdge(node2, node3)
```

Algorithm 1: Pseudocode of the fence elimination algorithm



(a) Critical edge in red (b) Breaking the critical edge

Figure 3: Breaking critical edges

we did not detail what to do when the control-flow graph hits the limits of the function being optimised (either the function entry point, or a return instruction). Since a function can be called in an arbitrary context, and a context can perform memory accesses before and after it, the function boundaries are treated as memory accesses (see lines 19 and 32), ensuring that fences at the beginning or end of the function are preserved. Similarly, a function call that may access the memory is treated as always modifying the memory.

Critical edges Algorithm 1 does not precisely detail how fences are inserted once the min-cut has been computed. If there is a cut required between the last instruction of a block A and the first instruction of a block B, there are three possibilities:

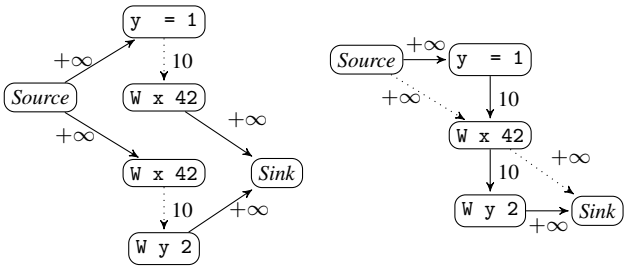
- block A is the only predecessor of block B: the fence can be inserted at the beginning of block B;
- block B is the only successor of block A: the fence can be inserted at the end of block A;
- if A has several successors, and B has several predecessors, then the edge between A and B is called a *critical edge* since there is no block in which a fence can be inserted ensuring that it is only on the path from A to B. The critical edge is broke by inserting an extra empty basic block between A and B.

This last case is illustrated in Figure 3a, inserting a fence at the location of the red edge is problematic, as inserting it either at A or at B puts it on a path without the red edge. This is solved by breaking the red edge in two, with a block in between, as in Figure 3b. The fence can now be safely inserted in block C.

Why two nodes for the each instruction Algorithm 1 systematically creates a node before and a node after each instruction. Perhaps surprisingly, a simpler solution where only one node is created for each instruction would not be be correct. A simple example illustrates this:

```
y = 1;
fence;
x = 42;
fence;
y = 2;
```

We report the flow graphs for this program, two nodes per memory access on the left, and one node per memory access on the right:



In the flow-graph on the left, the min-cut procedure finds a cut of finite weight (shown by the dotted lines). The edges in this cut are exactly where the fences are in the original program, so the optimisation correctly recognise that the program cannot be optimised.

If instead there is only one node for the store in the middle (flow-graph shown on the right), all cuts between *Source* and *Sink* have an infinite weight, and the min-cut procedure would pick a cut that contains edges involving *Source* and *Sink*. Such edges do not directly correspond to a location in the code, and cannot guide the successive fence-insertion-along-the-cut phase.

3. Correctness of the algorithm

Reasoning on shared-memory programs running on relaxed memory architectures is subtle and error prone. In this section we show that the algorithm is correct by formalising and proving our intuition that the algorithm never removes existing fences between memory accesses.

3.1 Background: the x86 and ARM/Power memory models

We build on the axiomatic formalisation of the x86 and ARM/Power memory models of [4]. In this framework the semantics of a program is defined in three steps. First the program is mapped to a set of memory events (e.g. atomic read or writes of shared memory locations) and several relations among these events. The intuition is that these events capture all possible memory accesses performed by the various threads running in an arbitrary concurrent context. The additional relations capture how several "syntactic" aspects of the source program lift to, and relate, events. These include the *program order* relation (po), that lifts the program order to the events, and the addr, data, and ctrl relations, that lift address, data and control dependencies between instructions to events.

Second, *candidate executions* are built out of the events by imposing restrictions on which values read events can return, each execution capturing a particular data-flow of the program. The restriction are expressed via two relations, called *reads-from (rf)* and *coherence-order (co)*. The former associates each read even to the write event it observes, the second imposes a per-location total order on memory writes, capturing the base property ensured by cache coherence. Several auxiliary relation are computed out of rf and co. The most important is the relation *from-read* (fr), defined as $\mathsf{rf}^{-1}; \mathsf{co}$, that relates a read to any store that is co-after the one it reads from. We also compute the projections of rf, fr and co to the events internal to a thread (the related events belong to the same thread, denoted by the suffix i) or external (the related events belong to different threads, denoted by the suffix e) to a thread; in turn we get the relations rfi, rfe, fri fre and coe.

Third, a *constraint specification* decides if a candidate executions are valid or not. HerdingCats is a framework that can be instantiated to different architectures. For this it is parametric in two relations *happens-before* (hb), defining the relative ordering of the events., and *propagation order* (prop), capturing the relative ordering of memory writes to different locations that somehow synchronise. We detail the instantiation of the framework to x86, Power, and ARM below. The framework then imposes only four axioms on candidate executions to determine their validity.

The first axiom, *SC per location*, captures cache coherence: for every memory location considered in isolation, the model behaves as if it was sequentially consistent. In formal terms:

$$\mathsf{acyclic}(\mathsf{poloc} \cup \mathsf{com})$$

where poloc is the program order restricted per location, and com is the union of the coherency order co, the reads-from relation rf, and the from-reads relation fr.

The second axiom, *no thin air*, prevents the so-called *out-of-thin-air reads*, where the read of a value is justified by a store that is itself justified by the original read. Such causality loops are

not observable on any modern hardware. In formal terms, the rule forbids cycles in *happens-before* (hb):

$$\mathsf{acyclic}(\mathsf{hb})$$

The third axiom, *observation*, guarantees that loads cannot read from stores that have been overwritten in-between:

$$\mathsf{irreflexive}(\mathsf{fre}; \mathsf{prop}; \mathsf{hb}^*)$$

In this rule fre is the restriction of fr to an inter-thread relation, hb^* is the transitive and reflexive closure of hb.

The fourth axiom, *propagation*, enforces the compatibility of the coherence order with this prop relation:

$$\mathsf{acyclic}(\mathsf{co} \cup \mathsf{prop})$$

This is needed because, to capture cumulativity of memory fences, prop relates both loads and stores on ARM and Power instantiations of the framework.

Instantiation for the x86 architecture The x86-TSO memory model [20] can be recovered by defining $\mathsf{hb} = \mathsf{po} \backslash \mathsf{WR} \cup \mathsf{mfence} \cup \mathsf{rfe}$ and $\mathsf{prop} = \mathsf{hb} \cup \mathsf{fr}$, with mfence being the restriction of po to pairs of accesses with a *mfence* instruction in between, and $\mathsf{po} \backslash \mathsf{WR}$ the program order relation from which every pair from a write to a read has been removed. Not including WR in hb enables observing the read-write reordering typical of the store-buffering implemented by x86-TSO.

Instantiation for the Power architecture The IBM Power memory model is significantly more complex. We report the formal definition of hb and prop; these are explained in detail in [4].

$$\begin{aligned}
\mathsf{fences} &\stackrel{\text{def}}{=} (\mathsf{lwsync} \backslash \mathsf{WR}) \cup \mathsf{hwsync} \\
\mathsf{propbase} &\stackrel{\text{def}}{=} \mathsf{rfe}?; \mathsf{fences}; \mathsf{hb}^* \\
\mathsf{prop} &\stackrel{\text{def}}{=} (\mathsf{propbase} \cap \mathsf{WW}) \\
&\qquad \cup (\mathsf{com}^*; \mathsf{propbase}^*; \mathsf{hwsync}; \mathsf{hb}^*) \\
\mathsf{dp} &\stackrel{\text{def}}{=} \mathsf{addr} \cup \mathsf{data} \\
\mathsf{rdw} &\stackrel{\text{def}}{=} \mathsf{poloc} \cap (\mathsf{fre}; \mathsf{rfe}) \\
\mathsf{detour} &\stackrel{\text{def}}{=} \mathsf{poloc} \cap (\mathsf{coe}; \mathsf{rfe}) \\
ii_0 &\stackrel{\text{def}}{=} \mathsf{dp} \cup \mathsf{rdw} \cup \mathsf{rfi} \\
ci_0 &\stackrel{\text{def}}{=} (\mathsf{ctrl} + \mathsf{cfence}) \cup \mathsf{detour} \\
ic_0 &\stackrel{\text{def}}{=} \emptyset \\
cc_0 &\stackrel{\text{def}}{=} \mathsf{dp} \cup \mathsf{poloc} \cup \mathsf{ctrl} \cup (\mathsf{addr}; \mathsf{po}) \\
ii &\stackrel{\text{def}}{=} ii_0 \cup ci \cup (ic; ci) \cup (ii; ii) \\
ci &\stackrel{\text{def}}{=} ci_0 \cup (ci; ii) \cup (cc; ci) \\
ic &\stackrel{\text{def}}{=} ic_0 \cup ii \cup cc \cup (ic; cc) \cup (ii; ic) \\
cc &\stackrel{\text{def}}{=} cc_0 \cup ci \cup (ci; ic) \cup (cc; cc) \\
\mathsf{ppo} &\stackrel{\text{def}}{=} (ii \cap \mathsf{RR}) \cup (ic \cap \mathsf{RW}) \\
\mathsf{hb} &\stackrel{\text{def}}{=} \mathsf{ppo} \cup \mathsf{fences} \cup \mathsf{rfe}
\end{aligned}$$

In these, lwsync and hwsync are restrictions of po to pairs of accesses with respectively a `lwsync` or `hwsync` (Power's synchronising fence instructions) in between. The relation $\mathsf{ctrl} + \mathsf{cfence}$ refers to control dependencies which have been reinforced by executing an `isync` instruction. Finally the relations ii, ic, ci cc are defined as the least fixed point of the equations above. The intuition behind these is that i stands for "initiation" of an access, while c stands for "commit" of an access.

We point out that, although the model is complicated, the construction of our algorithm ensures that the proof of correctness of our algorithm only needs to study how shuffling synchronising fence instructions influences the fences relation between events.

The Herd formalisation of Power has been proved to be at least as relaxed as, but not equivalent to, the Power model defined in [18]. In practice, to the best of our knowledge, extensive empirical testing of both allowed and not-allowed behaviours of the Herd formalisation of Power did not reveal any unsoundness against actual Power implementations.

Instantiation for the ARM architecture The instantiation of HerdingCats for ARM follows the IBM Power one, with three differences:

- the relation hwsync is replaced by the relation dmb ish, that is the restriction of po to pairs of accesses with respectively a `dmb ish` synchronising barrier in between; similarly the relation cfence is replaced by the relation isb, that is the restriction of po to pairs of accesses with respectively a `isb` instruction in between;
- the relation lwsync is removed;
- the relation cc_0 is redefined as $\mathsf{dp} \cup \mathsf{ctrl} \cup (\mathsf{addr}; \mathsf{po})$, losing the poloc term (the reason for this is detailed at length in [4]).

Again, we stress that the proof of correctness of our algorithm only needs to focus on dmb ish relation between events.

3.2 Proof of correctness

To show the correctness of our algorithm we first prove a key lemma that states that we may add a fence on a path that had none, but never remove all fences from a path that had at least one. This captures the main intuition that drove our design. The rest of the proof then proceeds by a monotonicity argument. The proof applies to ARM, Power, and x86 instantiations of the model.

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).*

Proof. Since the algorithm only affects fence placements, we can build X from X' such that it has the same events, po, dependencies, and rf and co relations. Let a, b be two memory accesses such that $(a, b) \in \mathsf{fences}(X)$. By the Herding Cats construction this implies that there is a path through the control-flow graph that goes first through a, then through a fence f, and finally through b. Since `MakeGraphUpwards` stops at the first memory access encountered (or may stop earlier if it hits the beginning of the function), this path goes through a node connected to the *Source* at some point between a and f. Symmetrically, it goes through a node connected to the *Sink* between f and b. Because `MakeGraphUpwards` and `MakeGraphDownwards` follow the control-flow path, these two nodes must be connected along the path. So the min-cut procedure will have to cut along the path to separate the *Source* from the *Sink*. The algorithm inserts a fence at this point, ensuring $(a, b) \in \mathsf{fences}(X')$. □

This result lifts to executions.

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'.*

Proof. The relation fences appears in a positive position in the definition of the hb and prop relations, and every other relation

that appears in their definitions (rfe, com, ppo) is invariant by our transformation (since they do not depend on fences, and by Lemma 1 fences are the only part of the program execution that is transformed). □

Given a valid execution of a program, its *final state* is obtained by taking the last write event in the co order for each shared memory location. We state that an optimisation that transform the program P into the program P' is *correct* if for every final state of the memory reached by a valid execution of P', there exists a valid execution of P that ends with the same final state.

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

Proof. We must show that for any valid execution of a program after `TransformFunction`, there is a valid execution of the source program that has the same final state. Corollary 1 provides an effective way to construct an execution E of the source program from an execution E' of the transformed program. Since co is unchanged by the construction and the events are identical, the final state of E and E' is the same. It remains to show that the execution is valid, that is, we must check that the four axioms of the Herding Cats model hold on E. We already know that $\mathsf{hb}(E) \subseteq \mathsf{hb}(E')$ and $\mathsf{prop}(E) \subseteq \mathsf{prop}(E')$. We also know that the axioms hold on E'. It is straightforward to chech that, whenever pairs of elements are added to prop or hb, the four axioms can only be made false, and never true. In turn, since they hold on E' they must also hold on E, ensuring that E is a valid execution of the source program. □

4. Implementation and extensions to other ISAs

We have implemented our optimisation as the first optimisation pass in the x86, ARM and IBM Power backends of LLVM 4.0. Min-cut is known to be equivalent to finding a maximum flow in a flow network (via the *max-flow min-cut theorem* of [10]), so our implementation uses the standard push-relabel algorithm of [11] to compute the min-cut of a graph.

LLVM internally provides the `BlockFrequency` class, that exports a relative metrics that represents the number of times a block executes. By default the block frequency info is derived from heuristics, e.g., loop back edges are usually taken, but if profile guided optimisation is enabled then it can rely on runtime profile information. Our implementation accesses the `BlockFrequency` class to compute the weight of the edges (all edges inside a basic block have the weight of the basic block) of the flow graph.

In Section 2 we have described the ARM implementation. The IBM Power and x86 passes differ slightly, as detailed below.

Extension to IBM Power As we have seen, the IBM Power memory model is very similar to the ARM one, but is gives to the programmer two different synchronising fence instructions. Heavyweight fences (instruction `hwsync`) are equivalent to the `dmb ish` instruction on ARM, but IBM Power also provides lightweight synchronising fences (instruction `lwsync`) that are faster but offer strictly less guarantees.

We can adapt our algorithm to IBM Power fence instructions by running it in two passes. In a first pass, the heavyweight fences are optimised, while ignoring the lightweight fences. In a second pass, the lightweight fences are optimised, considering every path with a heavyweight fence already cut (since heavyweight fences subsumes lightweight ones). This second pass can be implemented by adding the following code snippet after both line 17 and line 35 of Algorithm 1:

```
else if(inst is a heavyweight fence)
  return NULL;
```

This processing in two phases is sound. The first pass preserves the hwsync relation (by the same argument as before). Then the second pass may lose parts of the program execution that is lwsync relation, but it preserves the fences relation, defined as:

$$\mathsf{fences} \stackrel{\mathsf{def}}{=} (\mathsf{lwsync} \backslash \mathsf{WR}) \cup \mathsf{hwsync}$$

which is the only place where the relation lwsync is used. The proof then proceeds as before based on the monotonicity of the model with respect to both the hwsync and fences relations.

Extension to x86 In the x86 memory model fences prevent write-read reorderings and only matter between stores and loads. This is evident from the axioms: the relation mfence only appears in the definition of hb:

$$\mathsf{hb} = \mathsf{po} \backslash \mathsf{WR} \cup \mathsf{mfence} \cup \mathsf{fre}$$

where $\mathsf{po} \backslash \mathsf{WR}$ is the program order relation from which every pair from a write to a read has been removed. So we can relax our algorithm without affecting its correctness, and make it preserve $\mathsf{WR} \cap \mathsf{mfence}$ rather than mfence. This enables optimising the program on the left into the program on the right (which would have not been possible with the original algorithm):

```
x = 1                    x = 1
mfence                   x = 2
x = 2                    mfence
mfence                   tmp = x
tmp = x                  tmp = y
mfence
tmp = y
```

To implement this more aggressive optimisation, we alter our algorithm to connect the *Source* to the latest stores before each fence, and connect the earliest loads after them to the *Sink*. Algorithm 1 can be modified by replacing "memory access" by "store" on line 14, and "memory access" by "load" on line 32 of algorithm 1. The proof structure is unaltered: we can still show that for every execution of the optimised program, there is an execution of the original program with hb and prop that are no larger.

5. Experimental evaluation

We consider several concurrent algorithms, including Dekker and Bakery mutual exclusion algorithms, Treiber's stack, as well as a more realistic code-base, LibKPN [15]. The code of the first three benchmarks is taken from [23], with all shared memory accesses converted to sequentially consistent atomic accesses. The latter is a much larger C11 program (about 3.5k lines) that implements a state-of-the-art dynamic scheduler for green threads communicating through First-In First-Out queues. In the LibKPN code base, use of atomic qualifiers has been aggressively hand-optimised.

The table in Figure 4 reports on the number of fences deleted by the x86, ARM, and IBM Power, backend. This is not an ideal metric for our algorithm, because it does not capture cases where a fence is hoisted out of a loop: in this case there is one fence deleted and one inserted, but at run-time a fence per loop iteration might have been optimised away. However it is easily computable and gives some indications on the behaviour of the algorithm.

LLVM's x86 backend does not map sequentially consistent atomic writes to `mov; mfence` but relies on the locked `xchg` instruction, which is believed to be faster. We thus modified the x86 backend to implement an alternative mapping for sequentially consistent accesses: either `mov; mfence` is used for all stores, or `mfence; mov` is used for all loads. We report on both mappings.

The *compile time overhead* due to running our optimisation *is not measurable* and buried in statistical noise, even when compiling larger applications like LibKPN.

Benchmark	Configuration	Fences before optimisation	after optimisation	inserted	deleted
Bakery	ARM	18	16	0	2
Bakery	Power	24	19	10	15
Bakery	x86: all seq-cst, mfence after stores	4	3	0	1
Bakery	x86: all seq-cst, mfence before loads	10	2	1	9
Dekker	ARM	11	9	1	3
Dekker	Power	10	9	5	6
Dekker	x86: all seq-cst, mfence after stores	4	3	0	1
Dekker	x86: all seq-cst, mfence before loads	3	2	2	3
Treiber's stack	ARM	14	13	2	3
Treiber's stack	Power	14	12	4	6
Treiber's stack	x86: all seq-cst, mfence after stores	2	1	0	1
Treiber's stack	x86: all seq-cst, mfence before loads	4	2	2	4
LibKPN	ARM: default	110	110	4	4
LibKPN	ARM: all seq-cst	451	411	11	51
LibKPN	Power: default	92	90	2	4
LibKPN	Power: all seq-cst	448	394	54	108
LibKPN	x86: default	25	22	4	7
LibKPN	x86: all seq-cst, mfence after stores	189	144	5	50
LibKPN	x86: all seq-cst, mfence before loads	326	137	25	214

Figure 4: Fence instructions found in the assembly code before and after the optimisation pass

Observe that on x86 our algorithm mimics the results of the x86-tailored fence optimisation presented in [23].

For LibKPN we test both the hand-optimised version and a version where all atomic accesses are replaced by `seq_cst` accesses. On the hand-optimised code, our algorithm still finds opportunities to move and delete a few fences, e.g. reducing it from 25 to 22 on x86, or to move a few around according to the profiling informations available. As our optimisation cannot be expressed as a source to source program transformation it is possible that a few fences can be eliminated or moved, even if the qualifier annotations were "optimal". The test where all LibKPN atomic accesses are downgraded to `seq_cst` might suggest that, despite eliminating a fair number of redundant fences, the optimisaiton pass cannot generate code competitive with a hand-tuned implementation.

To investigate this point, and to evaluate the runtime speedup (or slowdown) induced by our of fence optimisation pass, we evaluate the execution time of several LibKPN benchmarks. These include both microbenchmarks to test some library constructions, and a larger signal processing application, called `radio`, taken from the *streamIt* benchmark suite[2] and ported to LibKPN. For testing we used a commodity x86 Intel Xeon machine with 12-cores (no hyper-threading), and a 48-cores IBM Power7 (IBM 8231-E2C). Unfortunately we do not have an ARM machine for testing.

Microbenchmarks timings do not benefit from running our optimisations. We conjecture that these stress-test tight loops involving synchronisation and barriers, and as such do not leave much space for improvement of the fence placement. Replacing all the atomic qualifiers with `seq_cst` atomics does not degrade the performance measurably either: this supports our conjecture.

The larger benchmark is much more informative. We report the average and standard deviation (in parentheses) of the performances observed (smaller is better), over 100 invocations of the benchmark:

	x86_64	IBM Power7
original code, not optimised:	252 (67)	1872 (852)
original code, optimised:	248 (55)	1766 (712)
all seq cst, not optimised:	343 (57)	3170 (1024)
all seq cst, optimised:	348 (46)	2701 (892)

[2] `http://groups.csail.mit.edu/cag/streamit/`

The large standard deviation observed is due to the dependance of the benchmark on the scheduling of threads. As a result on x86 performance change due to fence optimisation is hidden in statistical noise, matching the experiments in [23]. Similarly performance improvement due to passing profiling informations rather that relying on the default heuristics is also hidden in statistical noise (e.g. we observe 246 (53) for x86_64 on original code, optimised). On Power the results are more encouraging: speedup is observable and it is up to 10% on the code written by a non-expert programmer who systematically relies on the sequentially-consistent qualifier for atomic accesses.

6. Related work

Fence optimisation as a compiler pass When we started this work, the only fence elimination algorithm implemented in LLVM was described in [9]. This is an ARM-specific pass, that proposes both the straightforward elimination of adjacent `dmb ish` instructions in the same basic block with no memory access in-between (trivially performed by our algorithm too), and a more involved inter-block algorithm. This second algorithm was not integrated into LLVM, making it hard to understand and evaluate. It does not seem to have been proven correct either, and does not take into account profiling information. More generally compilers are extremely cautious when it comes at optimising memory barriers: the paper [17] reports on how a correct optimisation on C11 low-level atomics was undone by GCC developers because it was breaking the GCC4.7 internal invariant "never optimise an atomic access".

Vafeiadis and Zappa Nardelli [23] defined two algorithms (called FE1 and FE2) for removing redundant `mfence` instructions on x86 and proved them correct against the operational description of the x86-TSO memory model. The definition of the optimisations is specific to x86 and the correctness proof leverages complicated simulation-based techniques. Additionally, they proposed a PRE-like step that saturates with `mfence`s all branches of conditionals, in the hope that FE2 will later optimise these away later. Their algorithm does not take into account any profiling information. Our algorithm, once tailored for x86 as described in Section 4, subsumes both their optimisation passes, and additionally takes into account profiling informations while performing partial redun-

dancy elimination. Our correctness proof, building on an axiomatic model, turns out to be much simpler than theirs.

Fence synthesis based on whole program analysis Several research projects, including [13], [7], [3], [16], and [2], attempt to recover sequential consistency, or weaker safety specifications, by inserting fences in racy programs without atomics annotations. These projects attempt to compute optimal fence insertions but rely on *whole program analyses*: as such these are not directly implementable as optimisation passes in a compiler that performs separate compilation. Also they are too expensive to perform in a general-purpose compiler.

Despite the complementary goals, we remark that the algorithm in [3] is related to ours: to solve the global optimisation problem it uses an integer linear program solver, of which min-cut is a special case, and correctness follows along similar lines. Our algorithm can thus be seen as a variant of theirs, that trades-off optimality for the ability to apply in modular compilation, and to make use of atomics annotation by the programmer. It is interesting to remark that [3] gives an explicit cost to each fence and lets the solver decide on how to mix them, instead of the heuristic based on profiling informations we rely on. Obviously, this is only possible because they do an optimisation based on solving an ILP problem, instead of the simpler min-cut procedure we rely on.

Working inside a compiler, the work by Sura et al. [21], show how to approximate Sasha and Snir's delay sets and synthesising barriers so that a program has only SC behaviours: they performs much more sophisticated analyses than the ones we propose, but do not come with correctness proofs.

7. Perspectives

We have proposed, proved correct, and implemented in LLVM, a backend fence optimisation pass. Our optimisation pass cannot be implemented as a source-to-source transformation in the C11/C++11 memory model, but builds on the precise semantics of the barrier instructions in each targeted architecture. As shown in [8], the LLVM concurrency semantics itself is subtly different from the C11/C++11 memory model , motivating even further the study of ad-hoc backend optimisations. We have seen that in practice the pass can be effective in removing redundant barriers inserted by the conservative programmer that relies on strong qualifiers (e.g. `seq_cst`) for the atomic accesses. This is a promising result, as recent progress in understanding relaxed memory models paves the way for progress in this area, still largely unexploited by mainstream compilers.

It would be interesting to further investigate experimentally in which cases there is a benefit in replacing the `xchg` mapping for sequentially consistent stores used by LLVM with the `mov; mfence` mapping backed by an aggressive fence optimisation pass. More generally, large benchmarks to test performance of concurrent C11/C++11 code bases are still missing, and highly needed to perform a more in-depth performance evaluation.

Our algorithm can be improved in several ways, detailed below.

Interprocedural analysis As mentioned in Section 2.3, our algorithm is intra-procedural and treats any function call that may touch memory as a memory access. It is possible to slightly improve this by building function dictionaries. For this, starting from the leaves of the call-graph, we can annotate each function with:

- whether all paths from the first instruction hit a fence before a memory access or return instruction;
- whether all paths flowing backwards from a return instruction hit a fence before a memory access or the starting instruction.

Then, anytime a function call is encountered while looking for an access after (resp. before) a fence, and the first (resp. second) flag is set on that function, that path can be cut.

Leveraging coherency It is possible to implement a more aggressive optimisation if the algorithm keeps track of the C11/C++11 atomic access responsible for inserting each fence instruction. Consider a release store to a variable `x` on ARM: this is compiled to a `dmb ish` barrier followed by the store instruction. Since the semantics of release forces only to order previous accesses with this store and not with every later access, and accesses to the same location are already ordered locally (by the *SC-per-loc* rule of the Herding Cats framework), it is possible to ignore all accesses to `x` when looking for memory accesses to connect the *Source* node to.

The same idea can be applied to acquire or sequentially consistent loads. In particular, this would allow sinking the fence out of the loop when a loop contains only an acquire load, having the effect of transforming:

```
while (!x.load(acquire)) {};
```

into the more efficient:

```
while(!x.load(relaxed)) {};
fence(acquire);
```

Optimising for code size Following an idea from [19], it is possible to optimise for code size instead of runtime. Our algorithm can be easily adapted by using the constant 1 for the weight of all edges, instead of the frequency in the profiling information. The min-cut procedure will then minimise the number of fences in the generated code. A tradeoff between code-size and runtime can easily be obtained by using a hybrid metric (this idea comes from [19]).

A conservative variant of the algorithm As presented above, our algorithm is fundamentally speculative: it relies on the profiling information and can significantly worsen the performance of the code if this information is unreliable.

However any path that goes through a location dominated or post-dominated by a fence necessarily goes through that fence (by definition of dominators and post-dominators). Consequently, it is safe to put fences in all of these locations: there is no risk of introducing a fence on a path that had none. We conjecture that we can get a conservative variant of this algorithm by connecting the first node (resp. the last node) of any block in `MakeGraphUpwards` (resp. `MakeGraphDownwards`) that has at least one predecessor (resp. successor) that is not post-dominated (resp. dominated) by the original fence to the source (resp. the sink).

The paper [25] proposes a better solution to get a conservative algorithm, relying on dataflow analyses. Unfortunately this approach is hard to integrate with our currently implemented algorithm.

Artefact availability Our implementation of the optimisation, together with the code of the benchmarks, is available from `http://www.di.ens.fr/~zappa/projects/llvmopt`.

Acknowledgments Part of this work was done while the first author was at Google Mountain View. We are grateful to J.F. Bastien for useful discussions and code reviews, and to Nhat Minh Lê for help with the LibKPN benchmarks. The first author is supported by a Google PhD Fellowship.

References

[1] C/C++11 mappings to processors. `https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html`. Accessed: 2015-10-15.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Memorax, a precise and sound tool for automatic fence insertion under TSO. In *Tools and Algorithms for*

the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, pages 530–536, 2013.

[3] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence - A static analysis approach to automatic fence insertion. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 508–524, 2014.

[4] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, July 2014.

[5] Mark Batty, Kayvan Memarian, Scott Owens, Susmit Sarkar, and Peter Sewell. Clarifying and compiling c/c++ concurrency: From c++11 to power. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 509–520, New York, NY, USA, 2012. ACM.

[6] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 55–66, New York, NY, USA, 2011. ACM.

[7] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against tso. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 533–553, Berlin, Heidelberg, 2013. Springer-Verlag.

[8] Soham Chakraborty and Viktor Vafeiadis. Validating optimizations of concurrent c/c++ programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 216–226, New York, NY, USA, 2016. ACM.

[9] Reinoud Elhorst. Lowering C11 atomics for ARM in LLVM. Available from `http://llvm.org/devmtg/2014-04/PDFs/Talks/Reinoud-report.pdf`, 2014.

[10] Lester R Ford and Delbert R Fulkerson. Maximal flow through a network. *Canadian journal of Mathematics*, 8(3):399–404, 1956.

[11] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, October 1988.

[12] R Nigel Horspool and HC Ho. Partial redundancy elimination driven by a cost-benefit analysis. In *Computer Systems and Software Engineering, 1997., Proceedings of the Eighth Israeli Conference on*, pages 111–118, 1997.

[13] Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 111–120, Austin, TX, 2010. FMCAD Inc.

[14] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88, 2004.

[15] Nhat Minh Lê. *Kahn process networks as concurrent data structures: lock freedom, parallelism, relaxation in shared memory*. PhD thesis, École normale supérieure, France, December 2016.

[16] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 429–440, New York, NY, USA, 2012. ACM.

[17] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 187–196, 2013.

[18] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 175–186, New York, NY, USA, 2011. ACM.

[19] Bernhard Scholz, R. Nigel Horspool, and Jens Knoop. Optimizing for space and time usage with speculative partial redundancy elimination. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04), Washington, DC, USA, June 11-13, 2004*, pages 221–230, 2004.

[20] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-tso: A rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, July 2010.

[21] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. Compiler techniques for high performance sequentially consistent java programs. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 2–13, New York, NY, USA, 2005. ACM.

[22] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 209–220, 2015.

[23] Viktor Vafeiadis and Francesco Zappa Nardelli. Verifying fence elimination optimisations. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, pages 146–162, 2011.

[24] Jaroslav Ševčík. Safe optimisations for shared-memory concurrent programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 306–316, New York, NY, USA, 2011. ACM.

[25] Jingling Xue and Jens Knoop. A fresh look at PRE as a maximum flow problem. In *Compiler Construction, 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings*, pages 139–154, 2006.