LLVM Fence Synthesis

PARALLEL AND CONCURRENT PROGRAMMING

CS4560

Authors:

Group 3
Mihnea Bernevig (5496004)
Yigit Colakoglu (5495121)

April 13, 2025

Contents

1 Overview

In this project, we implemented three transformation passes for the LLVM compiler. Two of these passes are used to greedily insert Sequentially-Consistent (SC) fences for different target memory models, while the third pass eliminates redundant fences to optimize for overall code size.

Sequentially-Consistent fences enforce a happens-before relationship between the operations preceding the fence and those following it. This enforcement guarantees that all operations before the fence are completed prior to the execution of those after it, both at the hardware and compiler levels. We insert the fences during the IR stage so the LLVM backend can convert the IR fence instructions to the target architecture, or completely ignore them if necessary (For instance, SC fences are essentially ignored when compiling to x86).

2 Fence Insertion

We have developed two separate passes for fence insertion, one ensuring consistency under the Total Store Order (TSO) model and the other under the Partial Store Order (PSO) model.

Both of these passes work by traversing each function starting from their initial basic block, traversing all possible control flows of a program while keeping track of the last atomic memory operation that was performed so that it can insert a fence if a consecutive operation pair that is "too relaxed" appears in the path. The method used to traverse the basic blocks and insert fences for TSO and PSO memory order is shown in Algorithms 1 and 2, respectively.

Both of our algorithms insert two fences per operation pair, once after the initial operation, and once before the second operation. We have found that this approach works better in practice, especially when the LLVM IR contains ϕ nodes.

2.1 TSO vs. PSO Fence Synthesis

The main difference between the TSO and PSO memory synthesis passes is the consecutive atomic memory operation pairs that are "allowed". For instance, the PSO memory model allows `WW` memory operations to be re-ordered while TSO does not. So,the TSO pass inserts a fence before the second write if it encounters such a pair with relaxed memory order while the PSO pass does not care. A more detailed enumeration of the transformations we perform for each target memory order can be found in Table 1.

Op_1	$Order_1$	Op_2	$Order_2$	TSO	PSO
R	RLX	R	RLX	R-Fence-R	R-Fence-R
R	RLX	W	RLX	R-Fence-W	R-Fence-W
W	RLX	R	RLX	R-W	R-W
W	RLX	W	RLX	W-Fence-W	W-W
R	ACQ	*	*	R-Op_2	R-Op_2
*	*	W	REL	Op_1-W	Op_1-W
W	REL	R	ACQ	W-R	W-R
*	SEQ_CST	*	SEQ_CST	Op_1-Op_2	Op_1-Op_2
*	ACQ_REL	*	ACQ_REL	Op_1-Op_2	Op_1-Op_2

Table 1: Memory operation ordering under TSO and PSO

TL;DR: Fences are only introduced between two relaxed atomic memory operations as the C11 memory ordering for operations with other memory orders already behave the way we want them to.

Algorithm 1: TraverseBBGraph

Input: Basic block BB, ordering $order$, last memory operation $lastMemOp$

```
if BB is empty then
    return
for each instruction I in BB do
    if I is a LoadInst then
        loadOrder ← I.getOrdering();
        if lastMemOp is null then
            lastMemOp ← I;
            if order = Unordered ∧ loadOrder ∈ {Unordered, Monotonic} then
                insert fences with Sequentially-Consistent ordering before I;
            continue;
        if loadOrder ∉ {Unordered, Monotonic, Acquire} // Skip unsupported ordering then
            order ← Unordered;
            lastMemOp ← I;
            continue;
        if lastMemOp is a StoreInst then
            // No fence needed when previous operation is a store and current is a load (Write-Read)
        if lastMemOp is a LoadInst and order ∉ {Acquire, AcquireRelease, SequentiallyConsistent}
         then
            insert fences with Sequentially-Consistent ordering before I and after lastMemOp;
        order ← loadOrder;
        lastMemOp ← I;
    else if I is a StoreInst then
        storeOrder ← I.getOrdering();
        if lastMemOp is null then
            lastMemOp ← I;
            if order = Unordered ∧ storeOrder ∈ {Unordered, Monotonic} then
                insert fences with Sequentially-Consistent ordering before I;
            continue;
        if lastMemOp is a StoreInst and order ∉ {Release, AcquireRelease, SequentiallyConsistent}
         then
            insert fences with Sequentially-Consistent ordering before I and after lastMemOp;
        else if lastMemOp is a LoadInst and
         order ∉ {Acquire, AcquireRelease, SequentiallyConsistent} then
            insert fences with Sequentially-Consistent ordering before I and after lastMemOp;
        order ← storeOrder;
        lastMemOp ← I;
    else if I is a FenceInst then
        order ← I.getOrdering();
    else if I is a Terminator instruction then
        if I is a BranchInst then
            for each successor BB′ of I do
                if BB′ = BB then
                    continue;
                TraverseBBGraph(BB′, order, lastMemOp);
        else if I is a SwitchInst then
            for each successor BB′ of I do
                if BB′ = BB then
                    continue;
                TraverseBBGraph(BB′, order, lastMemOp);
        else
            return;
```

Algorithm 2: TraverseBBGraphPSO

Input: BasicBlock BB, AtomicOrdering $order$, Instruction pointer $lastMemOp$

```
1  if BB is empty then
2      return
3  for each instruction I in BB do
4      if I is a LoadInst then
5          loadOrder ← I.getOrdering();
6          if lastMemOp is null then
7              lastMemOp ← I;
8              if order = Unordered ∧ loadOrder ∈ {Unordered, Monotonic} then
9                  insert fences with Sequentially-Consistent ordering before I;
10             continue;
11         if loadOrder ∉ {Unordered, Monotonic, Acquire} then
12             order ← Unordered;
13             lastMemOp ← I;
14             continue;
15         if lastMemOp is a StoreInst then
16             continue;
17         if lastMemOp is a LoadInst and order ∉ {Acquire, AcquireRelease, SequentiallyConsistent}
            then
18             insert fences with Sequentially-Consistent ordering before I and after lastMemOp;
19         order ← loadOrder;
20         lastMemOp ← I;
21     else if I is a StoreInst then
22         storeOrder ← I.getOrdering();
23         if lastMemOp is null then
24             lastMemOp ← I;
25             if order = Unordered ∧ storeOrder ∈ {Unordered, Monotonic} then
26                 insert fences with Sequentially-Consistent ordering before I;
27             continue;
28         if lastMemOp is a StoreInst then
29             insert fences with Sequentially-Consistent ordering before I and after lastMemOp;
30         else if lastMemOp is a LoadInst and
            order ∉ {Acquire, AcquireRelease, SequentiallyConsistent} then
31             insert fences with Sequentially-Consistent ordering before I and after lastMemOp;
32         order ← storeOrder;
33         lastMemOp ← I;
34     else if I is a FenceInst then
35         order ← I.getOrdering();
36     else if I is a Terminator instruction then
37         if I is a BranchInst then
38             for each successor BB′ of I do
39                 if BB′ = BB then
40                     continue;
41                 TraverseBBGraphPSO(BB′, order, lastMemOp);
42         else if I is a SwitchInst then
43             for each successor BB′ of I do
44                 if BB′ = BB then
45                     continue;
46                 TraverseBBGraphPSO(BB′, order, lastMemOp);
47         else
48             return;
```

3 Fence Optimization

As we mentioned earlier, our initial methods insert fences greedily without attempting any optimizations to not insert any redundant fences. After this synthesis stage, we implemented a secondary optimization to remove redundant fences using redundant fence elimination techniques using the min-cut of the control flow graph [1].

In order to remove redundant fences, we build a special graph by finding and removing each fence instruction in the control flow, and inserting a node before and after where they used to be. Afterwards, for each inserted node before/after the fence, we traverse the basic block graph upwards and downwards until we encounter a memory operation or the start/end of the function flow graph, respectively. Once one of the following nodes are inserted, they are connected to a special source (if before the fence) or sink (if after the fence) node, essentially forming a network flow graph for the function. The resulting graph is directed, with weights of 1 for all edges that are not connected to the source/sink and ∞ for the edges that are connected to the source/sink. The pseudocode for the graph generation functions can be found in Algorithms 3-5.

Once we have the network flow graph G for a function, we need to calculate the min-cut of the resulting graph, the edges of which will correspond to the optimal placement of graphs without changing the function's behavior. To calculate the min-cut, we run the Ford-Fulkerson max-flow algorithm on the graph G. This results in a residual graph G' with the edges on the min-cut having been fully saturated. This means (given G was connected as a result of our graph generation procedure), G' is split into two subgraphs $G_s = (V_s, E_s), G_t = (V_t, E_t)$, with the edges of weight 0 between them corresponding to the min-cut. To find those edges, we traverse all nodes accessible from the source, marking all the vertices of the encounter. The edges whose out-vertex is marked and in-vertex is not are exactly the edges $\mathit{MinCut} = \{e = (a, b) | a \in V_s \wedge b \in V_t\}$. Finally, we insert back Sequantial Consistency fences on the locations that correspond to the min-cut.

Algorithm 3: makeGraphUpwards

Input: Instruction pointer $root$, Graph $graph$
Output: Node pointer

```
1  bb ← root.getParent();
2  foundRoot ← false;
3  for each instruction inst in bb in reverse order do
4      if inst = root then
5          foundRoot ← true;
6          continue;
7      if ¬foundRoot then
8          continue;
9      if inst is a memory access (i.e., a LoadInst or StoreInst) then
10         node ← getNode(inst, after);
11         graph.addNode(node);
12         graph.addEdge(graph.source, node);
13         return node;
14 node ← getNodeAtBeginning(bb);
15 if bb = bb.getParent().getEntryBlock() then
16     graph.addNode(node);
17     graph.addEdge(graph.source, node);
18     return node;
19 for each predecessor pred of bb do
20     node2 ← getNodeAtEnd(pred);
21     graph.addNode(node2);
22     lastInst ← getLastInst(pred);
23     node3 ← makeGraphUpwards(lastInst, graph);
24     if node3 ≠ null then
25         graph.addEdge(node, node2);
26         graph.addNode(node3);
27         graph.addEdge(node2, node3);
28 return node;
```

Algorithm 4: makeGraphDownwards

```
Input: Instruction pointer root, Graph graph
Output: Node pointer
bb ← root.getParent();
foundRoot ← false;
for each instruction inst in bb (in order) do
    if inst = root then
        foundRoot ← true;
        continue;
    if ¬foundRoot then
        continue;
    if inst is a memory access (i.e., a LoadInst, StoreInst, or ReturnInst) then
        node ← getNode(inst, before);
        graph.addNode(node);
        graph.addEdge(node, graph.sink);
        return node;
node ← getNodeAtEnd(bb);
for each successor succ of bb do
    node2 ← getNodeAtBeginning(succ);
    graph.addNode(node2);
    firstInst ← getFirstInst(succ);
    node3 ← makeGraphDownwards(firstInst, graph);
    if node3 ≠ null then
        graph.addEdge(node, node2);
        graph.addNode(node3);
        graph.addEdge(node2, node3);
return node;
```

Algorithm 5: TransformFunction

```
Input: Function pointer fun, Graph graph
for each Basic Block bb in fun do
    for each Instruction inst in bb do
        if inst is a FenceInst then
            nodeBeforeFence ← makeGraphUpwards(inst, graph);
            nodeAfterFence ← makeGraphDownwards(inst, graph);
            if nodeBeforeFence ≠ null and nodeAfterFence ≠ null then
                graph.addNode(nodeBeforeFence);
                graph.addNode(nodeAfterFence);
                graph.addEdge(nodeBeforeFence, nodeAfterFence);
for each Basic Block bb in fun do
    for each Instruction inst in bb do
        if inst is a FenceInst then
            remove inst from bb;
```

4 Testing

In order to test the validity of our fence synthesis techniques we decided write litmus tests used in the lectures in LLVM IR and use the LLVM test suite in order to check that they were inserted in where we expected them to. We implemented three different litmus tests that contained consecutive read-write, write-write and write-read operations, which covered the insertion cases for both TSO and PSO's fence synthesis methods. More specifically, we implemented the load-buffer, store-buffer and message passing litmus tests.

For each litmus test, we tested the operations with different memory orders to check that the memory orders of operations are considered accordingly during the synthesis as well. Moreover, we implemented an additional test to ensure that the control flow is accounted for when inserting fences.

References

[1] 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. [Online]. Available: https://doi.org/10.1145/3033019.3033021.