Brief Announcement: Fence Insertion for Straight-line Programs is in P

Mohsen Lesani
University of California, Riverside
lesani@cs.ucr.edu

ABSTRACT

Relaxed memory models reorder instructions in the interest of performance. However, reordering of instructions can jeopardize correctness and memory fences should be used to preserve specific orders. Programs that carry explicit fences are over-specified as they are tied to specific architectures and memory models and are hence unportable. On the other hand, once the program specifies the high-level required orders, optimizing compilers can allocate optimum memory fences for multiple architectures. However, the fence insertion problem for general programs is NP-hard. In this paper, we consider fence insertion for straight-line programs. We present a polynomial-time greedy algorithm via reduction to the chain multi-cut problem.

T_1	T_2
$1: \quad write(l_1, 1)$	$1: \quad write(l_2, 1)$
$2: \quad x_1 = read(l_2)$	$2: \quad x_2 = read(l_1)$
$\{1 \rightarrow 2\}$	$\{1 \rightarrow 2\}$

(a) Input Program

T_1	T_2
$1: \quad write(l_1, 1)$	$1: \quad write(l_2, 1)$
$\quad fence(w, r)$	$\quad fence(w, r)$
$2: \quad x_1 = read(l_2)$	$2: \quad x_2 = read(l_1)$

(b) Program with Fences

Figure 1: Dekker

1 INTRODUCTION

Compilers and processors reorder instructions to gain performance. However, preservation of the order of specific instructions is crucial to correctness. Hardware architectures provide memory fence instructions that preserve the relative order of specific instructions that come before and after them in the program. Synchronization programs have been traditionally written with explicit fence instructions for specific architectures. Such a program is an over-specification as it hard-codes the enforcement of the required orders for a particular architecture. Further, it is an under-specification as the required orders that are implicitly provided by the architecture are not explicitly documented. Thus, fences are an implementation mechanism for a high-level abstraction.

Many authors have presented approaches to insert fences that enforce sequential consistency, including Lee et al. [10], Fang et al. [7], and Alglave et al. [1]. Others have tried to infer the required orders including Kuperstein et al. [9], Meshman et al. [11] and Dan et al. [6]. We proposed [2] to capture the required orders as a relation on the operations of each thread. A compiler can translate these declared orders to optimum fence instructions. This approach separates what from how. The programs can be verified using architecture-independent and algorithm-level reasoning. The compiler can automatically translate the program to multiple target architectures.

Optimum fence insertion for general programs is NP-hard. The compiler of our previous work used an exponential-time algorithm to insert optimum fences. In this paper, we consider the basic straight-line class of programs. We present a polynomial-time fence insertion algorithm for this class. This opens the problem of whether there are optimum or approximation algorithms for fence insertion to programs with basic control structures.

As an example, consider the specification of the Dekker synchronization algorithm in Figure 1. Figure 1(a) shows the input program. Each thread first writes to a location and then reads from the location that the other thread writes to. It is crucial to the correctness of the algorithm that the write operation happens before the read operation in each thread. This ordering requirement is captured by a relation on the operations of each thread in Figure 1(a). The first operation should be ordered before the second operation. Figure 1(b) shows the allocated fences for the program in Figure 1(a). We denote the fence instruction that orders write instructions before read instructions as $fence(w, r)$. We use a similar notation for any pairs of r or w.

As another example, consider Figure 2. The input program is depicted in Figure 2(a). It requires the order of two pairs of operations to be preserved: it requires the first operation to be ordered before the third operation and the second operation to be ordered before the fourth operation. On the order hand, it does not require the preservation of the order of other pairs of operations including the order of the first and second operations. Figure 2(b) shows a suboptimal fence insertion. This insertion is based on the heuristic of putting a fence right before the second operation of each pair of operations in the required order. Figure 2(c) shows the optimal fence insertion. A single fence after the second operation preserves the order of the two pairs of operations.

In this paper, we present an algorithm for optimal fence insertion for straight line programs and show that it runs in polynomial time.

PODC '17, Washington, DC, USA
© 2017 Copyright held by the owner/author(s). 978-1-4503-4992-5/17/07…$15.00
DOI: http://dx.doi.org/10.1145/3087801.3087849

```
1:   write(l_1, 1)
2:   write(l_2, 2)
3:   x_1 = read(l_3)
4:   x_1 = read(l_4)
{1 → 3, 2 → 4}
```

(a) Input Program

```
1:   write(l_1, 1)
2:   write(l_2, 2)
     fence(w, r)
3:   x_1 = read(l_3)
     fence(w, r)
4:   x_1 = read(l_4)
```

(b) Suboptimal Fence Insertion

```
1:   write(l_1, 1)
2:   write(l_2, 2)
     fence(w, r)
3:   x_1 = read(l_3)
4:   x_1 = read(l_4)
```

(c) Optimal Fence Insertion

Figure 2: Fence Insertion

2 FENCE INSERTION PROBLEM

Notation. For a set s, let $|s|$ denote the size and $\mathbb{P}(s)$ denote the power set of s. Let $\langle e_1, .., e_n \rangle$ denote the tuple of elements e_1 to e_n. For a tuple t, let $|t|$ denote the length of t and $t[i]$ denote the i-th element of t.

Concurrent Program. Let D be a finite data domain, L be a finite set of shared memory locations valued in D, and X be a finite set of local variables valued in D. Let O be the set of operations (1) $x = read(l)$, (2) $write(l,x)$ and (3) $write(l,d)$ where $x \in X$, $l \in L$ and $d \in D$. Intuitively, $x = read(l)$ denotes the read of the value of location l into the variable x and $write(l,d)$ and $write(l,x)$ denote the write of the value d and the value of the variable x into the location l respectively. Let τ denote a function from O to $\{r, w\}$ that maps operations to their types: $\tau(x = read(l)) = r$ $\tau(write(l,x)) = w$, and $\tau(write(l,d)) = w$ for every $x \in X$, $l \in L$, and $d \in D$. Similarly let ℓ denote the function from O to L that maps operations to the memory locations that they access.

A program π is a pair $\langle p, \rightarrow \rangle$ such that p is a tuple of operations over O and $\rightarrow$ is an irreflexive subset of the increasing total order of numbers $\{1, .., |p|\}$. Intuitively, p is the straight-line program of operations and the relation $\rightarrow$ is the set of pairs of operations that should preserve their order in p. A concurrent system is a tuple of programs.

Fence Insertion Problem. We now define the fence insertion problem. We present a definition and then revise it based on a decomposition.

Let F be the set of the following fence types: $fence(r,r)$, $fence(r,w)$, $fence(w,r)$ and $fence(w,w)$. Intuitively, putting the fence $fence(r,w)$ between a *read* and a *write* operation prevents them from reordering. The effect of the other three fence operations is similar.

Intuitively, a fence insertion f for a program $\langle p, \rightarrow \rangle$ specifies the set of fences whose addition to p ensures that the required order -¿ is preserved. A (multi-type) fence insertion f for a program $\pi = \langle p, \rightarrow \rangle$ is a function from $\{1, .., |p|\}$ to $\mathbb{P}(F)$ such that for every i and j that $i \rightarrow j$, there exists n, $i \leq n < j$ such that $fence(\tau(p[i]), \tau(p[j])) \in f(n)$. For each $i \in \{1, .., |p|\}$, $f(i)$ specifies the set of fences that should be added between the ith and $(i+1)$-th operation of p.

For example, the fence insertion for the first thread of the Dekker example depicted in Figure 1(b) is $\{1 \mapsto \{fence(w,r)\}, 2 \mapsto \emptyset\}$. In other words, the fence $fence(w,r)$ should be put after the first operation and no fence is needed after the second operation. As another example, the fence insertion depicted in Figure 2(b) is $\{1 \mapsto \emptyset, 2 \mapsto \{fence(w,r)\}, 3 \mapsto \{fence(w,r)\}, 4 \mapsto \emptyset\}$. In other words, the fence $fence(w,r)$ should be put after the second and third operations and no fence is needed after the other operations.

The size of a fence insertion f is the number of fences that it allocates. More precisely, the size of a fence insertion f is $\Sigma_{i \in dom(f)} |f(i)|$. Given a program π, the fence insertion problem is to find the minimum-size fence insertion of π. Let $OptFI(\pi)$ denote the minimum-size fence insertion of π.

Fence Instructions Independence. We divide the required orders to four disjoint categories based on the fence types needed to preserve the order. For a program $\langle p, \rightarrow \rangle$, let $\rightarrow_{|wr}$ be the subset of $\rightarrow$ where the first operation is a write and the second operation is a read in p. More precisely, the order $\rightarrow_{|wr}$ is defined as the following relation $\{i \rightarrow_{|wr} j \mid i \rightarrow j \wedge \tau(p[i]) = w \wedge \tau(p[j]) = r\}$. The orders $\rightarrow_{|rr}$, $\rightarrow_{|rw}$ and $\rightarrow_{|ww}$ are similarly defined. Only the fence type $fence(w,r)$ can be used to preserve an order in the subset $\rightarrow_{|wr}$ and it cannot be used to preserve an order in any of the other three subsets. The following lemma states that (multi-type) fence insertion problem can be decomposed to four independent fence insertion problems for a single fence type.

Lemma 2.1 (Independence of fence types). *For every program $\langle p, \rightarrow \rangle$, $OptFI(\langle p, \rightarrow \rangle) = \lambda n, OptFI(\langle p, \rightarrow_{|rr} \rangle)(n) \cup OptFI(\langle p, \rightarrow_{|rw} \rangle)(n) \cup OptFI(\langle p, \rightarrow_{|wr} \rangle)(n) \cup OptFI(\langle p, \rightarrow_{|ww} \rangle)(n)$*

The lemma is straightforward from the definition of the optimum fence insertion and the fact that each fence type can only preserve its corresponding order subset.

Note that in a memory model that implicitly provides a category of orders, no fence insertion is needed for that category. Equivalently, we can translate the allocated fences for that category to no operation. In addition, if a memory model provides the same fence instruction for two or more fence types, we merge the corresponding categories and consider the union of them as a category.

Based on the above lemma, we consider a single fence type $fence$ and the following definition of fence insertion in the rest of the paper.

Definition 2.2 (Fence Insertion). A fence insertion f for a program $\pi = \langle p, \rightarrow \rangle$ is a subset f of $\{1, .., |p|\}$ such that for every i and j that $i \rightarrow j$, there exists n, $i \leq n < j$ such that $n \in f$. Given a program π, the fence insertion problem *FenceIns* is to find the minimum-size fence insertion of π.

Reduction of Fence Insertion to Multi-cut. We reduce the fence insertion problem to the graph multi-cut problem. We first define the multi-cut problem.

Definition 2.3 (Multi-cut). Given a graph $G = \langle V, E \rangle$ with a set of of terminal node pairs $T \in \mathbb{P}(V \times V)$, a multi-cut is a set of edges

Algorithm 1 Chain Multi-cut Greedy Algorithm

```
function CHAINMULTICUT($G\langle V, E\rangle, T$) where
$V = \{v_1, \ldots, v_n\}, E = \{\{v_1, v_2\}, \ldots, \{v_{n-1}, v_n\}\}$,
$T \in \mathbb{P}(V \times V)$

    $T' \leftarrow \{(v_i, v_j) \mid (\langle v_i, v_j\rangle \in T \wedge i < j) \vee$
                    $(\langle v_j, v_i\rangle \in T \wedge j < i)\}$
    $T'' \leftarrow$ Sort pairs $\langle v_i, v_j\rangle \in T'$ by $j$

    $E' \leftarrow \emptyset$
    for each $\langle v_i, v_j\rangle \in T''$ do
        if $(\nexists \{v_{i'}, v_{j'}\} \in E' \colon i \leq i' \wedge j' \leq j)$ then
            $E' \leftarrow E' \cup \{\{v_{j-1}, v_j\}\}$

    return $E'$
```

$E' \subseteq E$ such that for every pair $\langle s,t\rangle \in T$, there is no path between s and t in $\langle V, E \setminus E'\rangle$. Given a graph G, the multi-cut problem *MultiCut* is to find the minimum-size multi-cut of G.

The fence insertion problem is reduced to the multi-cut problem.

LEMMA 2.4. *FenceIns* $\leq_p$ *MultiCut*

Given an instance $\langle p, \rightarrow\rangle$ of *FenceIns*, we construct an instance of *MultiCut* in polynomial time as follows. The graph is $G = \langle V,E\rangle$ where $V = \{1,..,|p|\}$ and $E = \{\{1,2\},...,\{|p|-1,|p|\}\}$ and the terminal node pairs $T = \rightarrow$. We define polynomial transformations from solutions of each instance to other. Given the solution f for the *FenceIns* instance, the solution for *MultiCut* instance is $E' = \{\{i,i+1\} \mid i \in f\}$. Given the solution $E' = \{\{i_1,i_1+1\},...,\{i_n,i_n+1\}\}$ for *MultiCut*, the solution for the *FenceIns* instance is $f = \{i_1,...,i_n\}$. It is straightforward to see the optimality of one solution from the the other.

3 CHAIN MULTI-CUT.

The multi-cut problem is known to be NP-Hard (even for trees) [3, 8]. But the graph constructed above is a chain, a restricted tree. In the following section, we describe a polynomial time algorithm for multi-cut of chains.

Algorithm 1 presents a greedy algorithm for the chain multi-cut problem. An instance of the problem is represented by the sequence of nodes $V = \{v_1,...,v_n\}$, the edges $E = \{\{v_1,v_2\}, ..., \{v_{n-1},v_n\}\}$ between the nodes and the set of terminal pair of nodes T. The algorithm first swaps the terminal pairs, if needed, to have the larger node of each pair as the second element. It then sorts the terminal pairs according to the *second* element. Then, the algorithm iterates over the sorted terminal pairs. It starts with an empty set of edges to cut E'. For each pair $\langle v_i,v_j\rangle$ of terminals, it adds the edge $\{v_{j-1},v_j\}$ to the cut set E' only if there is no edge in E' that already cuts the path between v_i and v_j.

The time complexity of the algorithm is dominated by the sort operation on the terminal pairs; thus, it is $O(n \cdot \log n)$. The optimality of the algorithm can be proved by the following pair of facts. First, the size of the minimum cut is at least the size of every disjoint set of paths between the terminals. Second, the pairs of terminals that lead to addition of an edge to the cut set during the iteration have disjoint paths.

Let us see why the second fact holds. Consider a pair of terminals t_1 with the path p_1 between them. Assume that iteration over t_1 leads to the addition of an edge to the cut set. The last edge of p_1 is added to the cut set. Let the pair of terminals t_2 with the path p_2 between them be the pair that leads to the addition of the next edge to the cut set. The algorithm adds an edge to the cut set only if there is no edge in the current cut set that is in the path between the current pair of terminals. The last edge of p_1 is in the cut set and yet the pair t_2 leads to the addition of a new edge. Hence, the last edge in p_1 is not in the path p_2. This means that either the last edge of p_2 is before the last edge of p_1 or the first edge of p_2 is after the last edge of p_1. As the pairs are iterated in the order of their last edge and the pair t_1 is iterated before t_2, the last edge of p_2 cannot be before the last edge of p_1 Hence, we have that the first edge of p_2 is after the last edge of p_1. Thus, the first edge of p_2 is after the last edge of p_1. This means that the path p_1 is disjoint from the path p_2. Similarly, by induction on the steps of iteration, it can be shown that the pairs that lead to addition of an edge to the cut set have disjoint paths.

4 GENERALIZATION

The graph multi-cut problem is known to be NP-hard (even for undirected trees) [3, 8]. However, in contrast to undirected trees, multi-cut for directed trees is in P [4]. The result is based on the total uni-modularity of the linear programming matrix for directed trees. For a survey of the complexity of multi-cut on different restricted graph types see [5]. Control flow graphs of programs are restricted graph types. If we consider only the if-then-else control structure, then the graph is a nested composition of diamonds and sequences. The question is whether our result can be extended to polynomial-time optimum or approximation algorithms for programs with basic control structures.

REFERENCES

[1] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2014. Don't sit on the fence: A static analysis approach to automatic fence insertion. In *CAV*. http://arxiv.org/abs/1312.1411

[2] John Bender, Mohsen Lesani, and Jens Palsberg. 2015. Declarative Fence Insertion. In *OOPSLA*.

[3] Julia Chuzhoy and Sanjeev Khanna. 2006. Hardness of Cut Problems in Directed Graphs. In *Proceedings of the Thirty-eighth Annual ACM Symposium on Theory of Computing (STOC '06)*. ACM, New York, NY, USA, 527–536.

[4] Marie-Christine Costa, Lucas LéTocart, and FréDéRic Roupin. 2003. A Greedy Algorithm for Multicut and Integral Multiflow in Rooted Trees. *Oper. Res. Lett.* 31, 1 (Jan. 2003), 21–27. https://doi.org/10.1016/S0167-6377(02)00184-0

[5] Marie-Christine Costa, Lucas Ltocart, and Frdric Roupin. 2005. Minimal multicut and maximal integer multiflow: A survey. *European Journal of Operational Research* 162, 1 (2005), 55 – 69.

[6] Andrei Marian Dan, Yuri Meshman, Martin Vechev, and Eran Yahav. 2013. *Predicate Abstraction for Relaxed Memory Models.* Springer Berlin Heidelberg, Berlin, Heidelberg, 84–104. https://doi.org/10.1007/978-3-642-38856-9_7

[7] Xing Fang, Jaejin Lee, and Samuel P. Midkiff. 2003. Automatic Fence Insertion for Shared Memory Multiprocessing. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS '03)*. ACM, New York, NY, USA, 285–294.

[8] N. Garg, V.V. Vazirani, and M. Yannakakis. 1997. Primal-dual approximation algorithms for integral flow and multicut in trees. *Algorithmica* 18, 1 (1997), 3–20. https://doi.org/10.1007/BF02523685

[9] Michael Kuperstein, Martin Vechev, and Eran Yahav. 2010. Automatic Inference of Memory Fences. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design (FMCAD '10)*. FMCAD Inc, Austin, TX, 111–120.

[10] Jaejin Lee and David A. Padua. 2000. Hiding Relaxed Memory Consistency with Compilers. In *PACT*.

[11] Yuri Meshman, Andrei Dan, Martin Vechev, and Eran Yahav. *Synthesis of memory fences via refinement propagation.* Technical Report.