Polynomial-Time Fence Insertion for Structured Programs

Mohammad Taheri
Sharif University of Technology, Tehran, Iran
sm.taheri@sharif.edu

Arash Pourdamghani
Sharif University of Technology, Tehran, Iran
pourdamghani@ce.sharif.edu

Mohsen Lesani
University of California at Riverside, CA, USA
lesani@ucr.edu

Abstract

To enhance performance, common processors feature relaxed memory models that reorder instructions. However, the correctness of concurrent programs is often dependent on the preservation of the program order of certain instructions. Thus, the instruction set architectures offer memory fences. Using fences is a subtle task with performance and correctness implications: using too few can compromise correctness and using too many can hinder performance. Thus, fence insertion algorithms that given the required program orders can automatically find the optimum fencing can enhance the ease of programming, reliability, and performance of concurrent programs. In this paper, we consider the class of programs with structured branch and loop statements and present a greedy and polynomial-time optimum fence insertion algorithm. The algorithm incrementally reduces fence insertion for a control-flow graph to fence insertion for a set of paths. In addition, we show that the minimum fence insertion problem with multiple types of fence instructions is NP-hard even for straight-line programs.

2012 ACM Subject Classification Software and its engineering → Memory management; Software and its engineering → Synchronization; Software and its engineering → Concurrent programming structures

Keywords and phrases Fence Insertion, Synchronization, Concurrent Programming

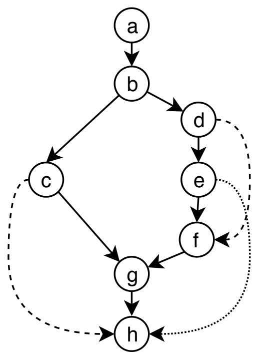
Digital Object Identifier 10.4230/LIPIcs.DISC.2019.34

Funding This project was supported by the NSF grant #1657204.

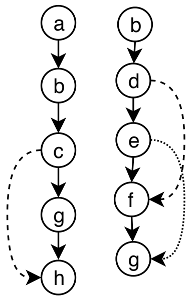
Acknowledgements We appreciate the reviewers of our DISC 2019 submission for constructive comments.

1 Introduction

To gain performance, processors reorder instructions. However, the correctness of concurrent programs is often crucially dependent on the preservation of the order of specific instructions. For example, a flag should be set before another flag is read. Thus, architectures provide memory fence instructions that preserve the relative order of specific instructions that come before and after them in the program. Experts have been traditionally programming synchronization algorithms with explicit fence instructions for specific architectures [7, 8, 12, 14]. The resulting program is an over-specification as it hard-codes the placement of the fences that enforce the required orders for a particular architecture. Further, it is an under-specification as the required orders that are implicitly provided by the architecture do not explicitly appear in the program. Fences are just an implementation mechanism for



(a) Control-flow graph (with solid arrows) and required orders (with dashed and dotted arrows).



(b) Decomposition.

Figure 1 Fence insertion for structured programs.

high-level order requirements. Concurrent algorithm designers require the order of certain instructions in their algorithms to be kept during the execution, and can readily declare these orders. Given the high-level order requirements [4, 11, 9], automatic synthesis tools that can decide the optimum fence insertion can enhance the ease of programming, reliability, maintainability, portability and performance of synchronization algorithms. This approach separates what from how. Programs can be verified using architecture-independent and algorithm-level reasoning and tools can automatically translate the program to multiple target architectures.

Lee et al. [21], Fang et al. [15], and Alglave et al. [5] presented methods to insert fences that enforce sequential consistency. Others have tried to infer the required orders including Kuperstein et al. [18], Meshman et al. [25] and Dan et al. [13]. Bender et. al. [9] proposed to capture the required orders as a relation on the statements of each thread and implemented a compiler to translate these declared orders to optimum fence insertion. Optimum fence insertion for general programs can be modeled as the minimum multi-cut problem that is NP-hard. The compiler presented in [9] used an exponential-time algorithm to insert optimum fences. Later Lesani [22] presented a polynomial-time fence insertion algorithm for the class of straight-line programs. This posed the problem of whether there are optimum or approximation algorithms for fence insertion to programs with basic control structures.

In this paper, we introduce a greedy and polynomial-time optimum fence insertion algorithm for the class of structured programs. This class includes programs that use structured branching statements such as if-then-else and switch-case and structured loop statements such as while and for. Further, through a reduction from the minimum set cover problem, we show that the minimum fence insertion problem with multiple types of fence instructions is NP-hard even for straight-line programs.

We observed that the control-flow graph of programs with structured branching has the structured form of nested diamonds. Figure 1.(a) shows an example. The diamond that branches at vertex b and merges at vertex g can represent an if-then-else statement where the left branch represents the then statement and the right branch represents the else statement. The example requires three orders: the order from c to h, from d to f, and from e to h should be enforced. The orders are shown as arrows. We use both dashed and dotted arrows to easily distinguish overlapping orders. The order between two vertices can be preserved in a path between them by placing a fence on an edge of the path. What is the minimum number of fences to preserve all the required orders?

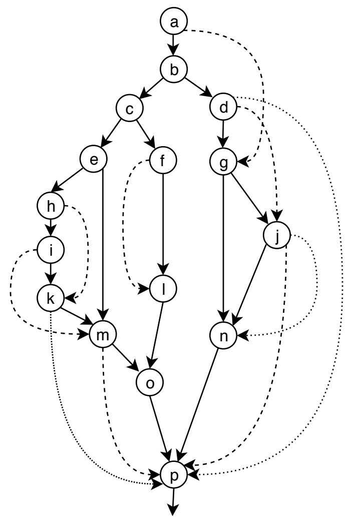


Figure 2 An example of AFG and its constraints. A constraint $\langle s, t\rangle$ is shown as a dashed arrow from the source s to the sink t. The diamonds $\langle e, m\rangle$ and $\langle g, n\rangle$ are at level 0. The diamond $\langle c, o\rangle$ is at level 1 and the diamond $\langle b, p\rangle$ is at level 2. The constraint $\langle h, k\rangle$ is an internal constraint for the diamond $\langle e, m\rangle$, the constraint $\langle d, j\rangle$ is a spanning constraint for the diamond $\langle g, n\rangle$, and The constraint $\langle d, p\rangle$ is a passing constraint for the diamond $\langle g, n\rangle$.

In this paper, we present an algorithm that reduces fence insertion for structured control-flow-graphs to fence insertion for a set of paths. It also presents a transformation that reduces fence insertion for looping structured programs to loop-free structured programs. For example, fence insertion for the graph shown in Figure 1.(a) is reduced to fence insertion for the two paths shown in Figure 1.(b). The high-level idea is that we can incrementally transform a diamond to a single branch by extracting branches. Fences can be independently inserted for the extracted branches. For example, the right branch of Figure 1.(a) is extracted in Figure 1.(b). The orders within a branch can be only preserved by fences inserted within that branch. Thus, the extracted right branch takes the order from d to f with it. Further, the extracted right branch can cover the spanning order from e to h with no extra fences. Thus, it takes in the spanning order from e to h too; it takes it as the shrunk order from e to g. Thus, fence insertion for the extracted right branch covers both constraints from d to f and from e to h. The left branch and the vertices above and below the diamond make the second path. The order from c to h overlaps with the left branch and stays within the second extracted path. The result is two paths and fencing for each can be done in polynomial time. We will elaborate on the algorithm in the following sections.

In the following sections, we first define the problem model (Section 2) and then present the greedy fence insertion algorithm for loop-free structured programs and state its optimality and complexity (Section 3). We then present a reduction from fence insertion for looping programs to fence insertion for loop-free programs (Section 4). Then, we prove the NP-hardness of fence insertion with multiple fence types (Section 5). Finally, we discuss the related works (Section 6) before we conclude (Section 7).

2 Problem Model

We now present the basic definitions and the problem instances that we use throughout the paper.

We consider the problem of *minimum fence insertion* for the set of *structured programs* that use branching statements such as if-then-else and switch-case and loop statement such as while and for. We represent this problem as the pair $\langle G, C\rangle$. The graph $G = \langle V, E\rangle$ is the control-flow graph (CFG) of the program. Each vertex $v \in V$ represents an executable instruction, and each edge $e \in E$ represents an execution transition. The control-flow graphs for loop-free structured programs are acyclic; thus, we call them *Acyclic Flow Graphs (AFG)*. The *constraints* C is the set of pairs of vertices of G that represent the required orders: a constraint is represented as a pair $\langle s, t\rangle$, where s and t are both vertices in V such that t is reachable from s. Figure 2 illustrates an instance of the fence insertion problem. The order between two vertices of a constraint can be preserved in a path between them by placing a fence on an edge of the path. To preserve the required order of a constraint between two vertices, a fence should be inserted in each path between them; then, we say that the constraint is *covered* by the inserted fences. Common hardware memory models do not allow reordering at the branch instructions. Therefore, we assume that branch instructions have implicit fences. Given a problem instance $\langle G, C\rangle$, the goal is to find the minimum number of fences that preserve all the constraints. We call the set of fences inserted on the edges of a graph, a *fencing* for that graph.

An AFG has a structured shape of nested diamonds. It has only one vertex with the input degree 0 that we call the start vertex and denote by v_0. It has only one vertex with the output degree 0 that we call the end vertex. Vertices with output degree more than one are called *branch* vertices and denoted by b. Vertices with input degree more than one are called *merge* vertices and denoted by m. The branch vertices start and the merge vertices end diamonds. Diamonds will help us reduce the problem on large graphs into a set of simple paths. A *simple path* is a path that has no branch vertex or merge vertex, except its head and tail. Diamonds are nested. A pair of $\langle b,m\rangle$ is a diamond of level 0 (called a *simple diamond*) iff (1) b is a branch vertex and m is a merge vertex and (2) all the paths starting from b reach m and all such paths are simple. In Figure 2, $\langle e, m\rangle$ and $\langle g, n\rangle$ are diamonds of level 0. A pair of $\langle b, m\rangle$ is a diamond of level k iff (1) b is a branch vertex and m is a merge vertex and (2) all the paths starting from b reach m, and if there is any branch vertex in between, it should be the starting vertex of a diamond of a level less than k whose merge vertex is not m. In Figure 2, $\langle c, o\rangle$ is a diamond of level 1 and $\langle b, p\rangle$ is a diamond of level 2.

Given a diamond, we categorize constraints into three types with respect to that diamond. A constraint is *internal* if both end points of the constraint are in the diamond. For example, in Figure 2, the constraint $\langle h, k\rangle$ is an internal constraint for the diamond $\langle e, m\rangle$. A constraint is *spanning* if only one of its endpoints is in the diamond. For example, in Figure 2, the constraints $\langle d, j\rangle$ and $\langle j, p\rangle$ are spanning constraints for the diamond $\langle g, n\rangle$. A constraint is *passing* if it passes over the diamond. More precisely, the branch vertex of the diamond is reachable from the source vertex of the constraint, the sink vertex of the constraint is reachable from the merge vertex of the diamond and every path from the source to the sink vertex of the constraint contains branch and merge vertices of the diamond. For example, in Figure 2, the constraint $\langle d, p\rangle$ is a passing constraint for the diamond $\langle g, n\rangle$. Since the branch vertex of a diamond is a jump instruction and the end vertex of a diamond represents the end label of the branch, there is no constraint between the two in real-world programs. Thus, we assume that constraints do not start at the branch vertex and finish at the merge vertex of a diamond.

3 Fence Insertion Algorithm for Loop-free Programs

Algorithm 1 Fence Insertion.

```
1: procedure FenceInsertion ($\langle AFG, C\rangle$)
2:     Eliminate the constraints in $C$ that are implicitly preserved in $AFG$.
3:     Find diamonds in $AFG$ and
4:         store them in a minimum priority queue $q$ based on their levels. (Algorithm 2)
5:     Decompose the diamonds in $q$ into a set of simple paths and their constraints and
6:         find the optimum fencing for each. (Algorithm 3 and Algorithm 4)
7:     Return the union of the fences placed on the simple paths.
```

In this section, we present an algorithm (Algorithm 1) that finds the optimal solution to the fence insertion problem for a given AFG in polynomial time. The algorithm has three steps. In the first step, it eliminates the constraints that are implicitly preserved, from the AFG. In the second step, it finds the diamonds of the AFG and puts them into a minimum priority queue based on their levels. The idea behind the algorithm for this step is to run a breadth-first search to label the merge and branch vertices, and then match them up accordingly. These labels are also used to assign the level of each diamond. The third step of the algorithm iterates through the diamonds in the priority queue and decomposes them into a set of separate simple paths. In this step, it also calculates the optimal fencing for each of the simple paths. Finally, it returns the union of these fencings. We will visit each of these steps in turn.

3.1 Constraint Elimination

Hardware memory models such as x86-TSO, SPARC TSO, MIPS and RISC-V can preserve control dependencies [27, 6, 3]. Therefore, spanning constraints that start before the branch vertex of a diamond and end inside a branch path of the diamond are implicitly preserved. Similarly, the passing constraints are implicitly preserved. For example, in the graph of Figure 2, the spanning constraints $\langle a, g\rangle$ and $\langle d, j\rangle$, and the passing constraint $\langle d, p\rangle$ are implicitly preserved. As Figure 5.(a) shows, the implicitly provided constraints are eliminated from the graph before the next steps. These constraints can be simply eliminated as follows: traverse the vertices from the start vertex, and at each branch vertex, eliminate the constraints which have been started but not finished.

3.2 Finding diamonds

In this section, we present an algorithm (Algorithm 2) that finds all the diamonds of the input graph and computes their levels. It returns a minimum priority queue of the diamonds based on their levels.

The algorithm starts from the start vertex v_0 of the graph and traverses the vertices by a breadth-first-search. It calculates a label for each vertex. It initializes the label of the start vertex v_0 to 0. In each iteration, a vertex v is visited. If v is not a branch vertex, the algorithm sets the label of v to the maximum of the labels of its parents. On the other hand, the labels advance on each branch. If v is a branch vertex, it sets the label of v to one plus the maximum of the labels of its parents. It also adds the branch vertex to a stack. If v is a merge vertex, its corresponding branch vertex b is popped from the stack. A new diamond is found with the branch vertex b and merge vertex v. The level of the diamond is the difference of the labels of the merge and branch vertices. When a diamond is found, it is added to a priority queue based on its level. Finally, the priority queue is returned.

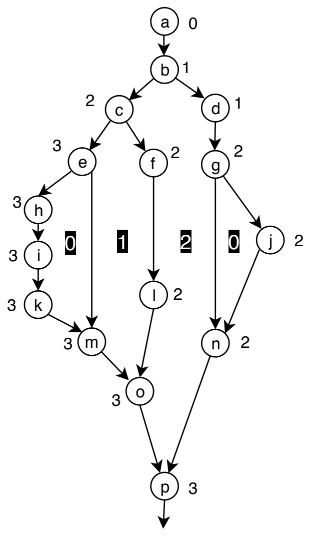
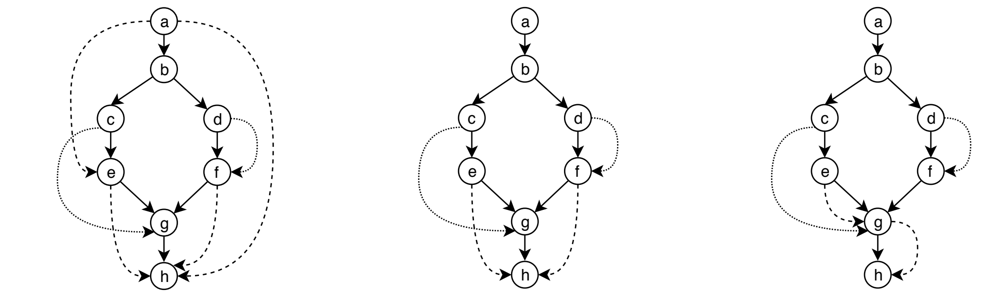


Figure 3 An example execution of Algorithm 2 that finds diamonds on the graph in Figure 2. The calculated label of each vertex is shown close to it. Four diamonds are found. The numbers with the dark background show the level of the enclosing diamonds. $level(\langle e, m\rangle) = 0$, $level(\langle g, n\rangle) = 0$, $level(\langle c, o\rangle) = 1$, $level(\langle b, p\rangle) = 2$.

As an example, Figure 3 shows the execution of Algorithm 2 on the graph in Figure 2. The calculated label of each vertex is shown close to it and the numbers with the dark background show the level of the enclosing diamonds. The label of the start vertex a is 0. The labels of the subsequent branch vertices b, c and e are 1, 2 and 3 respectively. The labels of the merge vertices m and o are both 3. Thus, the level of the diamond $\langle e, m\rangle$ is 0 and the level of the diamond $\langle c, o\rangle$ is 1. The algorithm finds four diamonds with the following levels: $level(\langle e, m\rangle) = 0$, $level(\langle g, n\rangle) = 0$, $level(\langle c, o\rangle) = 1$, and $level(\langle b, p\rangle) = 2$.

Algorithm 2 Finding Diamonds

```
1: procedure FindDiamonds ($AFG$)
2:                     ▷ $parents(v)$ and $children(v)$ return parents and children of $v$ in $AFG$
3:       ▷ Uses the FIFO queue $q$, the marked set $m$, the branch stack $s$ and the priority queue $p$
4:     $label(v_0) := 0$
5:     Add $v_0$ to $q$
6:     while $q$ is not empty do
7:         Pop $v$ from $q$
8:         Add $v$ from $m$
9:         $label(v) := \max_{p \in parents(v)} label(p)$
10:        if $v$ is a branch vertex then
11:            Push $v$ to $s$
12:            $label(v) := label(v) + 1$
13:        if $v$ is a merge vertex and all of its parents are in $m$ then
14:            Pop branch vertex $b$ from the stack
15:            Add diamond $\langle b, v\rangle$ with level $label(v) - label(b)$ to $p$
16:        else
17:            continue.
18:        Add $children(v)$ that are not in $m$ to $q$
19:    return $p$
```



(a) Example diamond 1.

(b) The diamond (a) after constraint elimination. The path *bceg* is absorbing. The path *bdfg* is emitting.

(c) The diamond (b) after the transformation of the spanning constraints. The constraints $\langle e, h\rangle$ and $\langle e, g\rangle$ are updated to $\langle f, h\rangle$ and $\langle g, h\rangle$.

Figure 4 Transformation of spanning constraints.

3.3 Decomposing Diamonds into Simple Paths

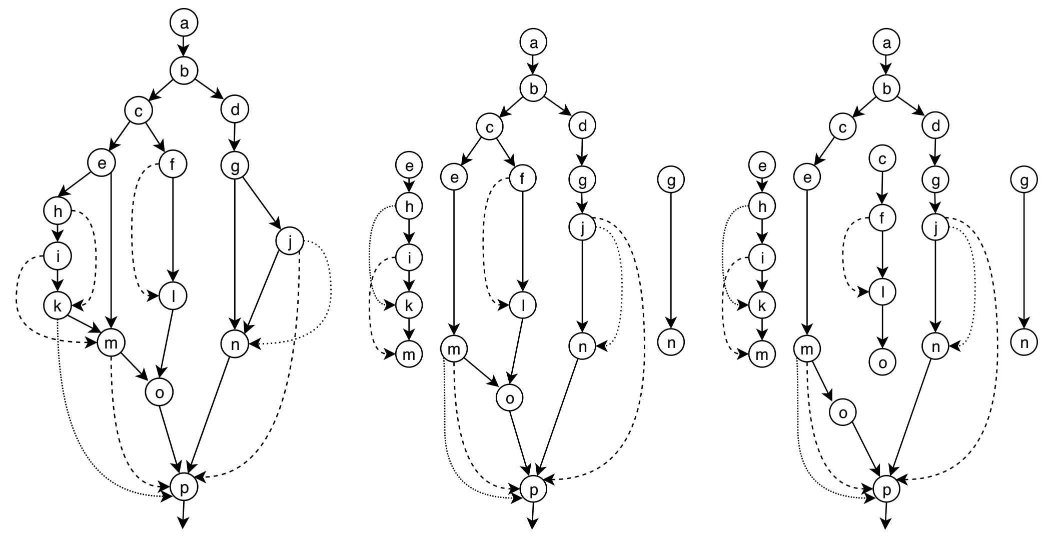
Algorithm 3 Decomposing Diamonds into a Set of Simple Paths.

```
1:  procedure FenceInsertion ($q$)    ▷ $q$ is the minimum priority queue ordering diamonds by level
2:      Initialize the set $F$ to $\emptyset$.
3:      while $q$ is not empty do
4:          Extract the innermost diamond $d$ from $q$.
5:          while there is more than one path in $d$ do
6:              Pick a path $p$ in $d$.
7:              Call Algorithm 4 on $p$ to find the fencing $f$ and the type $t$.
8:              Add $f$ to $F$
9:              if $t$ is absorbing then
10:                 Update the end point of the spanning constraints to the merge point of $d$.
11:             else                                                        ▷ $t$ is emitting
12:                 Update the start point of the spanning constraints to the merge point of $d$.
13:             Remove $p$ from $d$.
14:     return $F$
```

In this step, we present an algorithm (Algorithm 3) that decomposes each diamond into simple paths and finds the optimum fencing for them. The algorithm iterates the diamonds from the innermost to the outermost. For each diamond, it incrementally extracts simple paths until only a simple path remains in the diamond. Therefore, the degree of the nesting diamond decreases from one to zero. This makes the nesting diamond a simple diamond. As the algorithm iterates all the nested diamonds before the nesting one, diamonds are visited when they are already simple.

For each path of a diamond, the algorithm calls the fence insertion algorithm for simple paths (that we will see in Algorithm 4) to obtain an optimum fence placement for the internal constraints of the path. The rationale for the separation of paths is that the internal constraints of a path can be covered by only fences inside the path. Thus, the optimum fencing for the internal constraints can be locally determined. The algorithm then checks if the resulting fence placement can cover the spanning constraints of the path.

Accordingly there are two path types: absorbing and emitting. We use an example in Figure 4 to illustrate these types. Figure 4.(a) shows a simple diamond. Figure 4.(b) shows the resulting diamond after eliminating constraints that are implicitly preserved by the control dependencies. We illustrate the two path types on the diamond in Figure 4.(b).



(a) The graph of Figure 2 after constraint elimination.

(b) Decomposing diamonds of level 0.

(c) Decomposing diamonds of level 1.

Figure 5 Decomposition of Nested Diamonds into a Set of Simple Paths.

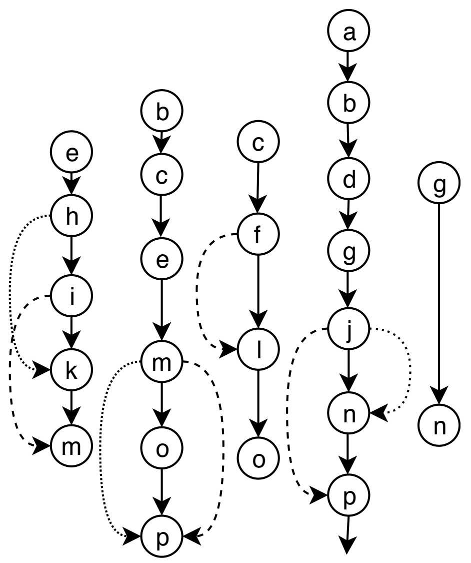


Figure 6 The Final Set of Simple Paths for Figure 5.(a)

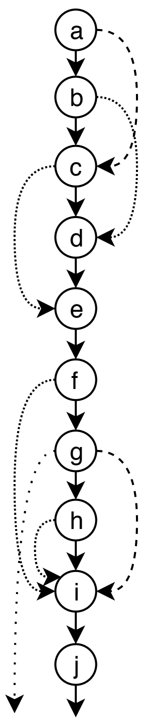


Figure 7 Fence insertion for a simple path. The algorithm visits the constraints in the order $\langle a, c\rangle$, $\langle b, d\rangle$, $\langle c, e\rangle$, $\langle f, i\rangle$, $\langle g, i\rangle$, and $\langle h, i\rangle$ and inserts the fences $\langle b, c\rangle$, $\langle d, e\rangle$, and $\langle h, i\rangle$. The inserted fences cover the spanning constraint starting from g.

- Absorbing: A path of a simple diamond is absorbing if the required fences for its internal constraints can cover its spanning constraints as well. For example, the paths *bceg* in Figure 4.(b) is absorbing because a fence at $\langle e, g \rangle$ handles both constraints.
- Emitting: A path of a simple diamond is emitting if it is not absorbing. In other words, a path is emitting if the required fences for its internal constraints cannot cover its spanning constraints. For example, the paths *bdfg* in Figure 4.(b) is emitting. The optimum fencing for the path *bdfg* in Figure 4.(b) is one fence on the edge $\langle d, f \rangle$ that does not cover the constraint $\langle f, h \rangle$.

To extract simple paths from a diamond, its spanning constraints should be transformed to be totally in or out of the path. The algorithm updates the spanning constraints of the paths according to their types. Absorbing paths absorb them inside and emitting paths emit them outside of the path. The rationale behind this transformation is that an absorbing path can cover the spanning constraint with no extra fence in the path. Thus, extra constraints are covered without increasing the number of fences. Therefore, the spanning constraint is pulled inside the path. On the other hand, in an emitting path, an extra fence is needed to cover the spanning constraint. This extra fence cannot cover any additional constraints inside the path but may cover other constraints outside the path. Therefore, the spanning constraint is pushed outside. Thus, the algorithm performs the following two transformations. (1) Transformation for absorbing paths: The spanning constraints stay in the path. The endpoints of the spanning constraints are updated to the merge point of the diamond. For example, in Figure 4.(b), the constraint $\langle e, h \rangle$ is updated to $\langle e, g \rangle$. (2) Transformation for emitting paths: The spanning constraints are pushed out of the path. The start points of the spanning constraints are updated to the merge point of the diamond. For example, in Figure 4.(b), the constraint $\langle f, h \rangle$ is updated to $\langle g, h \rangle$. We note that the transformation leaves the internal and passing constraints unchanged.

We illustrate the iteration over diamonds of different levels in Figure 5. Constraint elimination on the graph in Figure 2 results in Figure 5.(a). Figure 5.(b) shows the result of Algorithm 3 on Figure 5.(a) after processing the diamonds of level 0. The diamonds of level 0 are $\langle e, m \rangle$ and $\langle g, n \rangle$. For the diamond $\langle e, m \rangle$, the left simple path *ehikm* is extracted. The constraint $\langle k, p \rangle$ is a spanning constraint for this path. The optimum fencing for the internal constraints of this path is one fence on the edge $\langle i, k \rangle$ that does not cover the spanning constraint $\langle k, p \rangle$. So the path is emitting and the constraint $\langle k, p \rangle$ is shrunk to $\langle m, p \rangle$. Extracting the left path reduces the diamond to a simple path. The other diamond of level 0 is $\langle g, n \rangle$. The left edge $\langle g, n \rangle$ with no constraint can be simply extracted to reduce the diamond to a simple path.

Figure 5.(c) shows the result of Algorithm 3 on the graph of Figure 5.(b) after processing the diamonds of the next level, that has been level 1 in the original graph of Figure 5.(a). The only diamond of the next level is $\langle c, o \rangle$. There are no spanning constraints and simply extracting the right path *cflo* reduces the diamond to a simple path. Figure 5.(c) has only one diamond $\langle b, p \rangle$ left. After Algorithm 3 processes this diamond, the end result is the set of simple paths shown in Figure 6. The diamond $\langle b, p \rangle$ has no spanning constraints and it is simply split into two simple paths. The graph is decomposed into five separate simple paths.

3.4 Fence Insertion for Simple Paths

In this section, we present an algorithm (Algorithm 4) that finds the optimum fencing for simple paths. More precisely, given the internal constraints of a simple path and a bottom spanning constraint, the algorithm finds a minimal fence placement that covers the internal constraints and also decides whether the spanning constraint can be covered by no more fences. The algorithm can be trivially extended for more spanning constraints.

Algorithm 4 Fence Insertion and Deciding the Type for a Simple Path.

```
1: procedure FenceInsertionForSimplePath ($C$, $s$)
2:                          ▷ $C$ is the set of internal constraints and $s$ is the spanning constraint.
3:     Initialize $F$ to $\emptyset$.
4:     Sort $C$ to a list $L$ according to the end point in the top-to-bottom order.
5:     for (each constraint $c$ in $L$ in order) do
6:         Add to $F$ a fence $f$ on the last edge of $c$.
7:         Remove from $C$ the constraints that are covered by $f$.
8:     if ($f$ covers $s$) then
9:         Return $\langle F, \text{absorbing} \rangle$
10:    else
11:        Return $\langle F, \text{emitting} \rangle$
```

We illustrate the algorithm using the simple path shown in Figure 7 as an example. The example path has the set C of six internal constraints and a spanning constraint s at the bottom with the start vertex g and no end vertex. The algorithm first sorts the given set C of internal constraints to a list L according to their endpoints in the top-to-bottom order. In the example, the sorted order can be $\langle a, c\rangle$, $\langle b, d\rangle$, $\langle c, e\rangle$, $\langle f, i\rangle$, $\langle g, i\rangle$, and $\langle h, i\rangle$. It then iterates over constraints in L in order. For the current constraint c, the algorithm adds a fence at the bottom edge of c. It then removes any later constraint that is covered by the inserted fence. The rationale for putting the fence at the bottom edge is to cover the current constraint and also reach as far down as possible to cover the later constraints if possible. In the example, first the constraint $\langle a, c\rangle$ is visited and a fence is inserted at the edge $\langle b, c\rangle$. This fence covers the constraint $\langle b, d\rangle$ as well; so, it is removed from L. Next, the constraint $\langle c, e\rangle$ is visited that results in the insertion of the fence $\langle d, e\rangle$. Similarly, the next constraint $\langle f, i\rangle$ results in the fence $\langle h, i\rangle$. This fence covers the other two constraints as well. So the resulting set F of fences is $\langle b, c\rangle$, $\langle d, e\rangle$, and $\langle h, i\rangle$. The algorithm then checks whether the inserted fences cover the spanning constraint as well. If it does, the path is absorbing; otherwise, the path is emitting. In this example, the spanning constraint starting from g is covered by the inserted fence $\langle h, i\rangle$. So, the algorithm returns the set of fences F and that the path is absorbing.

3.5 Optimality and Complexity

In this section, we show that the optimality of the algorithm. We show that every optimal solution needs at least the number of fences that the algorithm inserts. In addition, we show the time and space complexity of the algorithm.

▶ **Theorem 1.** *Algorithm 4 provides optimum fence insertion for simple paths.*

Proof. The proof is by the following pair of facts. First, the size of the optimum solution is at least the size of every set of non-overlapping constraints. Second, the constraints that lead to addition of fences are non-overlapping. The algorithm visits the constraints by their endpoints, inserts fenced in the last edges of constraints, and removes all the covered constraints. Thus, if a fence is inserted for a constraint, its start point can be only at or after the end point of the last constraint that required a fence; thus, the two constraints do not overlap. ◀

▶ **Theorem 2.** *Algorithm 1 provides optimal fence insertion for AFGs.*

Proof. We prove this fact by induction on the level of diamonds. In the base case, suppose we have a diamond of level 0. The internal constraints of a branch can be covered by only fences inside the branch. Algorithm 1 uses Algorithm 4 to find the fencing for the branches of the diamond. By Theorem 1, each of these fencings are optimal for the internal constraints of that branch. Further, Algorithm 4 puts fences on the lowest possible edges. At the end, it checks whether the inserted fences can cover the spanning constraint as well and accordingly decides whether the paths are of absorbing or emitting type. Based on the type of the path, Algorithm 1 transforms the spanning constraints of the path: an absorbing path absorbs it inside and an emitting path emits it outside of the path. An absorbing path can cover the spanning constraint with no extra fence in the path. Thus, extra constraints are covered without increasing the number of fences. Therefore, the solution stays optimum after pulling the spanning constraint inside the path. On the other hand, in an emitting path, an extra fence is needed to cover the spanning constraint. If an extra fence is inserted inside the path, it cannot cover any additional constraints inside or outside the path. However, if it is put outside the path, it may cover other overlapping constraints. Therefore, pushing the spanning constraint outside can result in either the same or fewer number of fences. In the inductive case, consider a diamond of level k. Algorithm 1 reduces nested diamonds of lower levels to simple paths; thus, the diamond is reduced to a diamond of level 0. With the same argument as the base case, Algorithm 1 finds the optimum fencing. ◀

▶ **Theorem 3.** *Algorithm 1 is of $\mathcal{O}(|C|log|C| + |C||V| + |V|log|V|)$ time and $\mathcal{O}(|C| + |V|)$ space complexity.*

Proof. Algorithm 1 has three steps and its time complexity is the sum of their complexity. We consider each step in turn. We note that it takes $\mathcal{O}(|C|+|E|)$ space to represent the input.

In the first step, we eliminate the constraints that are implicitly covered by the branch vertices. The algorithm traverses the vertices from the start vertex, and for each branch vertex eliminates the constraints which have been started but not finished. Since each edge and each constraint is visited just once, the running time is $\mathcal{O}(|C| + |E|)$. Additionally, we do not need any extra memory for running this step. Therefore, its space complexity is $\mathcal{O}(|C| + |E|)$.

In the second step, the algorithm finds the diamonds (Algorithm 2) by traversing the graph vertices using a breath-first-search and pushes the diamonds into a priority queue. So, it takes $\mathcal{O}(|E|log|E| + |V|)$ time. The algorithm uses data structures that store vertices and thus, needs $\mathcal{O}(|V|)$ extra space for this step.

In the third step, the algorithm decomposes the diamonds (Algorithm 3) into simple paths and finds the optimum fencing for them. Algorithm 3 applies Algorithm 4 to each branch of each diamond. For a path p, let C_p be set of constraints on p. Algorithm 4 sorts C_p, which takes $\mathcal{O}(|C_p|log|C_p|)$ and then traverses C_p and inserts fences which takes $\mathcal{O}(|C_p|)$. Therefore, its time complexity is $\mathcal{O}(|C|log|C|)$. Also, it needs at most $\mathcal{O}(|E|)$ space to represent the fencing. Thus, fence insertion for the branches takes $\mathcal{O}(|C|log|C|)$ time and $\mathcal{O}(|E|)$ space for all the diamonds. In addition, Algorithm 3 updates spanning constraints for branches and extracts branches of each diamond. The graph has at most $\mathcal{O}(|E|)$ diamonds and in the worst case, a constraint may need to be updated when each diamond is visited. Therefore, updating the constraints takes $\mathcal{O}(|E||C|)$ time. It only needs $\mathcal{O}(|E|)$ additional space to represent the extracted paths.

We now sum the complexity of the steps. The time complexity of Algorithm 1 is $\mathcal{O}(|V| + |E|log|E| + |C|\log|C| + |C||E|)$. The space complexity of Algorithm 1 is $\mathcal{O}(|C| + |E| + |V|)$. To further simplify these orders, we show that $|E| \in \mathcal{O}(|V|)$. It is easy to see that the

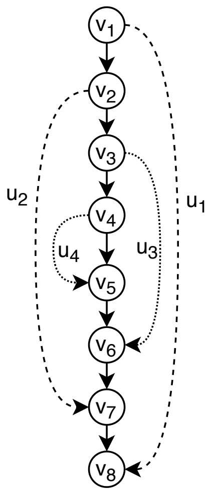
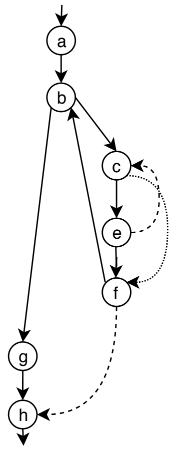
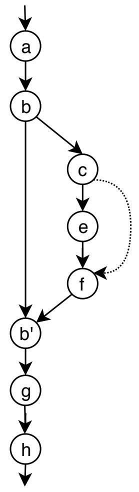


Figure 8 Reduction Example.



(a) A CFG with a loop.



(b) Transformation to an AFG.

Figure 9 Converting a Loop to a Diamond.

sum of the degree of all the merge vertices of an AFG is $\mathcal{O}(|V|)$. Similarly, the sum of the degrees of all the branch vertices of an AFG is $\mathcal{O}(|V|)$. Also, the sum of the degrees of all non-merge non-branch vertices is $\mathcal{O}(|V|)$. As a result, the sum of degrees of all the vertices is $\mathcal{O}(|V|)$ thus, $|E| \in \mathcal{O}(|V|)$. Therefore, the time complexity of Algorithm 1 is $\mathcal{O}(|V|log|V| + |C|log|C| + |C||V|)$ and its space complexity is $\mathcal{O}(|C| + |V|)$ ◀

4 Fence Insertion for Loops

In this section, we present a transformation for loops in a given CFG with loops to an AFG. Therefore, we can reduce fence insertion for any CFG to an AFG and use Algorithm 1 to find an optimal fence insertion.

We illustrate the transformation using an example. Figure 9.(a) shows a CFG with a loop. The vertex b is the branch instruction: it jumps either to the body of loop at the vertex c or out of the loop to vertex g. We call the edge $\langle b, c\rangle$ that jumps from the branch vertex to the loop body, the start edge. The body of the loop is a CFG in general. In this example, it is the simple path cef. We call the edge $\langle f, b\rangle$ that jumps from the end of the loop body back to the branch vertex, the return edge. We call the edge $\langle b, g\rangle$ that jumps from the branch vertex out of the loop, the exit edge.

We now transform the CFG in Figure 9.(a) to the AFG in Figure 9.(b). The graph has two internal constraints in the loop body: $\langle e, c\rangle$ and $\langle c, f\rangle$, and the constraint $\langle f, h\rangle$ from inside the loop body to outside of the loop. The constraint $\langle e, c\rangle$ is upwards: it requires the execution of e in one iteration of the loop to be executed before the execution of c in the next iteration of the loop. We notice that the branch instruction b is executed between the instructions of any iteration and the next. As mentioned in Subsection 3.1, hardware memory models can preserve control dependencies. Thus, the order of instruction between one iteration and the next is preserved. Therefore, the constraint $\langle e, c\rangle$ in Figure 9.(a) is implicitly enforced and is eliminated in Figure 9.(b). The constraint $\langle f, h\rangle$ will be eliminated with the same argument. Thus, we should preserve the constrains when the loop body is either executed once in an iteration or is not executed. To represent these two paths in a diamond, we add a vertex b' to represent a dummy instruction after the loop. The return edge $\langle f, b\rangle$ is updated to $\langle f, b'\rangle$. The exit edge $\langle b, g\rangle$ is updated to a dummy edge $\langle b, b'\rangle$ and the edge $\langle b', g\rangle$. Thus, the loop is transformed to the diamond $\langle b, b'\rangle$ and the constraint $\langle c, f\rangle$ remain unchanged.

No constraint ends at b'. Thus, Algorithm 4 puts no fence on the dummy edge $\langle b, b'\rangle$. In addition, the transformation did not change the constraints in the body or out of the body of the loop. Any fence on the new edge $\langle b', g\rangle$ corresponds to a fence on the old edge $\langle b, g\rangle$. Therefore, if a fence is needed in the resulting AFG, it is needed in the CFG as well and will cover the same set of constraints. Therefore, every optimal fence insertion for the AFG is an optimal fence insertion for the CFG.

5 Multi-type Fence Insertion Problem

Common architectures often offer different fence instructions that preserve the order of certain instruction pairs. In this section, we study the complexity of the insertion problem when there are different types of constraints and fences such that each fence type can cover a subset of the constraint types. (We note that these constraint types are defined based on the endpoint instructions for a target architecture, and are irrelevant to the three constraint types presented in Section 2.) We show that this problem is NP-hard even for straight-line programs through a reduction from the set cover problem.

An instance of the *Multi-type Fence Insertion* problem is defined as $\langle CT, FT, G, C\rangle$. CT is the set of constraint types. Each constraint has a type according to its endpoint instructions. FT is the set of fence types. Each fence type can cover a certain subset of the constraints types CT. G is the CFG. C is the set of constraints on G of different types from CT. The goal is to find the minimum number of fences, regardless of their types from FT, to cover C.

We provide a polynomial-time reduction from the minimum set cover problem to the multi-type fence insertion problem. The set cover has been one of the fundamental problems in computer science [17]. It has been shown that the minimum set cover problem is NP-hard [10] and it can be approximated with a $O(log\ n)$ factor [16].

▶ **Theorem 4.** *The multi-type fence insertion problem is NP-hard.*

Proof. We provide a reduction from an instance I of the minimum set cover problem to an instance of I' the multi-type fence insertion problem for straight-line programs. Consider an instance I of the minimum set cover problem $\langle U, S\rangle$ where $U = \{u_1, u_2, \ldots, u_n\}$ is the set of all elements and $S = \{S_1, S_2, S_3, \ldots, S_k\}$ are the subsets of U. The goal is to find the minimum number of the subsets in S that cover U.

The reduction defines the set of constraint types CT to be U. Each element of the set U corresponds to constraint type. The reduction also defines the set of fence types FT to be S. Each fence type S_i covers the set of constraint types that correspond to the elements in S_i. The reduction constructs a straight-line program with $2n$ instructions. Then, for each element $u_i \in U$, it creates a constraint $c_i = \langle v_i, v_{2n-i}\rangle$ with type u_i. Therefore, each constraint c_i will starts at the vertex v_i and ends at the vertex v_{2n-i} and can be covered by a fence S_i that includes the element u_i. Let us call the constructed instance I'.

As an example consider a minimum set cover instance with $U = \{u_1, u_2, u_3, u_4\}$ and $S = \{\{u_1, u_3\}, \{u_1, u_2, u_4\}, \{u_3, u_2\}\}$. As Figure 8 shows, it is reduced to a straight-line program with 8 instructions. The set of constraint types is U and the set of fence types is S. The constraints will be $\langle v_1, v_8\rangle$, $\langle v_2, v_7\rangle$, $\langle v_3, v_6\rangle$ and $\langle v_4, v_5\rangle$ of types u_1, u_2, u_3 and u_4 respectively.

First, we show that given a solution of the set cover instance I, a solution for the multi-type fence insertion problem I' can be constructed. Consider that the solution of I has chosen the subsets S_i to cover U. In the solution of I', we use the fences corresponding to the subsets S_i that I has chosen. The fences are all put in the middle edge of G. Since the subsets S_i cover all the elements u in U, the inserted fences S_i can cover the constraints that are of any type u in U. All the constructed constraints are of a type u in U; thus all of them are covered by the inserted fences. Next, we show that given a solution for the minimum multi-type fence insertion I', a solution for the minimum set cover I can be constructed. Any solution for the multi-type fence insertion can be transformed to a solution for it with the same number and type of fences by moving the fences to the middle edge. This is because the middle edge is in the middle of all the constraints. Thus, we consider an optimum solution for I' that has all the fences in the middle edge. A solution for the minimum set cover I can be constructed by simply choosing the sets S_i that correspond to the inserted fences in the solution of I'. Because otherwise, a smaller set cover for I can be transformed to a smaller fence insertion for I' that contradicts the assumption that the solution for I' is optimum. ◀

6 Related Work

In this paper, we introduced a polynomial-time algorithm to find the optimal fence insertion for structured programs and showed that fence insertion with multiple fence types is NP-hard. The previous methods that involve fence insertion can be grouped into the following categories.

Sequential Consistency: There have been attempts to insert fences to enforce sequential consistency. Lee et al. [21] used delay set analysis and dominators to reduce the number of fences that provide sequential consistency. To preserve sequential consistency during compilation, Fang et al. [15] applied several techniques to enhance fence insertion for memory models of specific architectures (SMPs on IBM Power 3 and Pentium 4). Linden et al. [23] presented a heuristic approach to output a correct but maybe suboptimal fence insertion to preserve sequential consistency on the x86-TSO memory model. Abdulla et al. [1, 2] applied reachability analysis for TSO and PSO memory models to optimize fence insertion for finite-state programs. Also, Alglave et al. [5] presented a practical and approximate static approach for fence insertion. They showed that certain cycles represent the violation of sequential consistency, statically detect the cycles and insert fences to remove them. The above related works try to preserve sequential consistency, are often empirical and do not focus on optimality guarantees. We observe that the correctness of concurrent programs is often dependent on only a few crucial orders rather than complete sequential consistency. Given these required orders, this paper presented a fence insertion algorithm that finds the optimum fencing to preserve them.

Inference: A few projects applied different techniques to automatically infer the required orders for the correctness of concurrent programs. Given a program, a correctness property, and a memory model, Kuperstein et al. [20] infer the required execution orders. They perform a whole-program state-space exploration that produces a logical formula, solve the formula to get a set of execution orders, and use those orders to insert fences. This approach infers sound but maybe suboptimal fence orders. Their follow-up works [19, 25, 13] extend the approach to degrees of infinite-state programs. Liu et al. [24] presented a dynamic inference approach that tests the input program to expose violations and adds orders to prevent the violations. The fence insertion algorithm presented in this paper takes the required orders as input and finds the optimum fencing to preserve them. The tools above can assist algorithm designers to declare the set of orders that are sufficient for correctness of the program.

Fence Elimination: To reduce the number of fences on a given relaxed memory model, there are practical techniques that eliminate redundant fences. Vafeiadis et al. [28] remove redundant fences that precede later fences or locked instructions. Morisset et al. [26] remove redundant fences for x86, ARM, and Power in the LLVM backend. Unlike the two above works that present correct techniques to reduce the number of fences, this paper proposes a fence-insertion algorithm with proof of optimality.

Hardness and Optimality: Lee et al. [21] showed that the decision version of the fence insertion problem with one type of fence on a general graph is NP-Complete. Bender et al. [9] implemented an exponential algorithm to compile declared orders to the optimum fencing. Lesani [22] focused on the limited class of straight-line programs and presented a polynomial-time algorithm. We observed that CFGs of structured programs have structured forms that can make the problem solvable in polynomial-time.

7 Conclusion

This paper considered the fence insertion problem for the class of structured programs and presented a greedy and polynomial-time optimum fence insertion algorithm. The algorithm reduces fence insertion for a control-flow graph (CFG) to fence insertion for a set of paths. It transforms looping CFGs to loop-free CFGs that are a set of nested diamonds. It then iterates the diamonds from the innermost to the outermost and incrementally extracts branches. Fence insertion for the extracted paths can be done independently and in polynomial time. This paper also proved that fence insertion with multiple fence types is NP-hard even for straight-line programs through a reduction from the set cover problem.

8 Future Work

This paper poses new avenues of investigation:

Multi-fence algorithms: If we assume that the number of different fence types is a constant k, the question is whether there is a polynomial-time parametrized algorithm for k. Otherwise, as the problem is NP-hard, the only other possible option is approximation algorithms.

Stochastic optimization: This paper presented an algorithm to minimize the number of fences. However, in common executions, some paths may be exercised more often than others. Given a probabilistic measure of how often each branch is executed, the question is to find the fencing that minimizes the probabilistic number of executed fences.

References

1 Parosh Aziz Abdulla, Mohamed Faouzi Atig, Magnus Lång, and Tuan Phong Ngo. Precise and sound automatic fence insertion procedure under PSO. In *International Conference on Networked Systems*, pages 32–47. Springer, 2015.

2 Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Tuan-Phong Ngo. The best of both worlds: Trading efficiency and optimality in fence insertion for TSO. In *European Symposium on Programming Languages and Systems*, pages 308–332. Springer, 2015.

3 Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996.

4 Jade Alglave and Patrick Cousot. Ogre and Pythia: an invariance proof method for weak consistency models. In *ACM SIGPLAN NOTICES*, volume 52 (1), pages 3–18. ACM, 2017.

5 Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence: A static analysis approach to automatic fence insertion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 39(2):6, 2017.

6 Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *International Conference on Computer Aided Verification*, pages 258–272. Springer, 2010.

7 Hagit Attiya, Danny Hendler, and Smadar Levy. An O (1)-barriers optimal RMRs mutual exclusion algorithm. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 220–229. ACM, 2013.

8 Hagit Attiya, Danny Hendler, and Philipp Woelfel. Trading fences with rmrs and separating memory models. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 173–182. ACM, 2015.

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

10 Korte Bernhard and J Vygen. Combinatorial optimization: Theory and algorithms. *Springer, Third Edition, 2005.*, 2008.

11 Karl Crary and Michael J Sullivan. A calculus for relaxed memory. In *ACM SIGPLAN Notices*, volume 50 (1), pages 623–636. ACM, 2015.

12 Luke Dalessandro, Michael F Spear, and Michael L Scott. NOrec: streamlining STM by abolishing ownership records. In *ACM Sigplan Notices*, volume 45 (5), pages 67–78. ACM, 2010.

13 Andrei Marian Dan, Yuri Meshman, Martin Vechev, and Eran Yahav. Predicate abstraction for relaxed memory models. In *International Static Analysis Symposium*, pages 84–104. Springer, 2013.

14 Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *International Symposium on Distributed Computing*, pages 194–208. Springer, 2006.

15 Xing Fang, Jaejin Lee, and Samuel P Midkiff. Automatic fence insertion for shared memory multiprocessing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 285–294. ACM, 2003.

16 Uriel Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652, 1998.

17 Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

18 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.

19 Michael Kuperstein, Martin Vechev, and Eran Yahav. Partial-coherence Abstractions for Relaxed Memory Models. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 187–198, New York, NY, USA, 2011. ACM. doi:10.1145/1993498.1993521.

20 Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. *ACM SIGACT News*, 43(2):108–123, 2012.

21 Jaejin Lee and David A Padua. Hiding relaxed memory consistency with compilers. In *Parallel Architectures and Compilation Techniques, 2000. Proceedings. International Conference on*, pages 111–122. IEEE, 2000.

22 Mohsen Lesani. Brief Announcement: Fence Insertion for Straight-line Programs is in P. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 97–99. ACM, 2017.

23 Alexander Linden and Pierre Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In *International SPIN Workshop on Model Checking of Software*, pages 144–160. Springer, 2011.

24 Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. *ACM SIGPLAN Notices*, 47(6):429–440, 2012.

25 Yuri Meshman, Andrei Dan, Martin Vechev, and Eran Yahav. Synthesis of memory fences via refinement propagation. In *International Static Analysis Symposium*, pages 237–252. Springer, 2014.

26 Robin Morisset and Francesco Zappa-Nardelli. Partially redundant fence elimination for x86, ARM, and Power processors. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 1–10. ACM, 2017.

27 Susmit Sarkar, Peter Sewell, Francesco Zappa-Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. *ACM SIGPLAN Notices*, 44(1):379–391, 2009.

28 Viktor Vafeiadis and Francesco Zappa-Nardelli. Verifying fence elimination optimisations. In *International Static Analysis Symposium*, pages 146–162. Springer, 2011.