Declarative Fence Insertion

John Bender
University of California,
Los Angeles (UCLA)
johnbender@cs.ucla.edu

Mohsen Lesani
Massachusetts Institute
of Technology (MIT)
lesani@csail.mit.edu

Jens Palsberg
University of California,
Los Angeles (UCLA)
palsberg@ucla.edu

Abstract

Previous work has shown how to insert fences that enforce sequential consistency. However, for many concurrent algorithms, sequential consistency is unnecessarily strong and can lead to high execution overhead. The reason is that, often, correctness relies on the execution order of a few specific pairs of instructions. Algorithm designers can declare those execution orders and thereby enable memory-model-independent reasoning about correctness and also ease implementation of algorithms on multiple platforms. The literature has examples of such reasoning, while tool support for enforcing the orders has been lacking until now. In this paper we present a declarative approach to specify and enforce execution orders. Our fence insertion algorithm first identifies the execution orders that a given memory model enforces automatically, and then inserts fences that enforce the rest. Our benchmarks include three off-the-shelf transactional memory algorithms written in C/C++ for which we specify suitable execution orders. For those benchmarks, our experiments with the x86 and ARMv7 memory models show that our tool inserts fences that are competitive with those inserted by the original authors. Our tool is the first to insert fences into transactional memory algorithms and it solves the long-standing problem of how to easily port such algorithms to a novel memory model.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers; D.4.1 [*Process Management*]: Concurrency; F.3.2 [*Semantics of Programming Languages*]: Program Analysis

Keywords Compilers, Concurrency, Weak Memory Models, Static Analysis

1. Introduction

For concurrent programs, programmers often face a gap between their assumptions about execution and the memory model of a specific architecture. For example, some programmers assume that execution is sequentially consistent, while most architectures provide weaker properties. Specifically, many of the performance optimizations performed by modern processors involve relaxing the perceived execution order of instructions. For example, both x86 and ARMv7 permit stores to move past loads of different addresses due to out of order execution or store buffering. A processor with such optimizations has a *weak memory model*.

The gap between a programmer's assumptions and a weak memory model creates the challenge to insert fences that enforce the assumptions on the weak memory model. Fortunately, researchers have shown how to insert fences automatically. For the case of sequential consistency, the goal is to do better than to insert a fence between every pair of consecutive instructions. A common idea is to recognize that so-called critical cycles [**?**] represent minimal violations of sequential consistency. A tool can identify critical cycles and insert fences to break them, as demonstrated by, for example, Alglave et al. [**?**]. Their approach first builds an integer-linear program that encodes the program, critical cycles, architecture, and potential fences, and then solves the ILP to determine where to insert fences.

While some concurrent programs are designed to be correct under the assumption of sequential consistency, others rely on weaker assumptions. For algorithms based on weaker assumptions, sequential consistency slows down execution more than is necessary because it requires many fences.

What is the nature of those weaker assumptions? We have found that, often, each assumption can be phrased as a required thread-local *execution order* of a pair of instructions. For example, if i_1 and i_2 are instructions, then the assumption (i_1, i_2) says that i_1 must happen before i_2. A programmer can enforce that assumption by, for example, inserting a fence right after i_1 or right before i_2. However, such a fence is a blunt instrument and an implementation detail, while the assumption (i_1, i_2) is memory-model independent. We have found that the number of required execution orders is usually rather small, often just a few per method. In contrast, sequential consistency is the special case where, in each thread, every instruction must happen before the next.

We believe execution orders should be an integral part of a concurrent algorithm and we envision that future languages may support them as a novel form of "control structure." For now, we note that algorithm designers can *declare* such ex-

ecution orders. The idea is that the execution orders specify all that is needed from the implementation without consideration for the memory model. Then all reasoning about correctness can take place at the algorithm level, while the implementation on a particular platform is an entirely separate issue. Our thesis is that concurrent algorithms with declared execution orders are easier to design and prove correct than concurrent algorithms with fences. Declared execution orders raise the level of abstraction for specifications of concurrent algorithms: they enable algorithm designers to focus on *what*, that is, which assumptions are needed for correctness, and leave the *how*, that is, how to enforce those assumptions, to a tool.

Lesani and Palsberg [**?**] proved the correctness of the transactional memory algorithm TL2 based on a few execution orders, and Lesani [**?**] proved the correctness of Dekker's algorithm based on a few execution orders. Those examples demonstrate the viability of the approach based on declared execution orders. However, tool support for enforcing the orders has been lacking until now.

The problem. How do we specify and enforce the assumptions made by concurrent algorithms? The specification should be memory-model independent, while the enforcement should be parameterized by a memory model.

Our solution. In this paper, we present a declarative approach to specify and enforce the assumptions made by concurrent algorithms. Our specification language is memory-model independent and supports declaration of execution orders that must be enforced. Thus, our approach enables concurrent-algorithm designers to specify their algorithms and assumptions independently of memory models. Our fence insertion algorithm first identifies the execution orders that are enforced automatically by a given memory model, and then inserts fences that enforce the rest. Thus, our algorithm bridges the gap between the specified assumptions and a given weak memory model.

Our tool. We have implemented our approach in a tool called Parry that as input takes a concurrent algorithm written in C/C++, declared execution orders, and a memory model, and as output produces C/C++ with fences.

Our experiments. Our benchmarks are seven open-source implementations of concurrent algorithms. Three of the benchmarks are transactional memory algorithms for which we first removed all fences and then added specifications of execution orders. Our experiments on x86 and ARMv7 show that Parry inserts fences that are competitive with those inserted by the original authors. For example, for the TL2 transactional memory algorithm [**?**] on x86, we show that Parry inserts one fewer fence than the experts who implemented the algorithm. Our tool is the first to insert fences into transactional memory algorithms and it solves the long-standing problem of how to easily port such algorithms to a novel memory model. For example, for the Byteeager implementation of the RSTM transactional memory

```
1  bool p0() {
2    flag0 = 1;
3    // fence
4    if(flag1 == 1){
5      return false;
6    }
7    // critical
8    flag0 = 0;
9    return true;
10 }
```

```
1  bool p1() {
2    flag1 = 1;
3    // fence
4    if(flag0 == 1){
5      return false;
6    }
7    // critical
8    flag1 = 0;
9    return true;
10 }
```

$$O_{\mathsf{Dekker}} = \{\, \mathsf{st}(\mathsf{flag0}) \rightarrow \mathsf{ld}(\mathsf{flag1}),\ \mathsf{st}(\mathsf{flag1}) \rightarrow \mathsf{ld}(\mathsf{flag0}) \,\}$$

Figure 1. Dekker's Mutual Exclusion Algorithm

algorithm, the implementers had in mind particular memory models that excluded ARMv7. Parry discovered the need for a fence that would have been missed entirely if one tried simply to modify the fences used for other architectures.

Modularity. Our approach both assumes and provides modularity. The correctness theorems for concurrent algorithms typically assume *noninterference*, that is, other threads won't access the algorithm's shared state, which therefore is encapsulated. In return, those correctness theorems guarantee that the algorithm will work correctly in any noninterfering context. Thus, our tool inserts fences once-and-for-all that will work in any noninterfering context.

Rest of the paper. In Section 2 we give examples, in Section 3 we present our algorithm, in Section 4 we introduce our implementation, in Section 5 we show experimental results, and in Section 6 we discuss related work.

2. Examples

We begin with two examples to illustrate the ideas behind declarative execution orders. First we will consider Dekker's mutual exclusion algorithm as an introduction. Then we will examine part of the TL2 transactional memory algorithm to highlight the subtlety of enforcing execution orders. We conclude with an outline of how we enforce execution orders.

2.1 Dekker's Mutex

Figure **??** shows an implementation of Dekker's mutual exclusion algorithm. Each procedure represents a process that will attempt to enter a critical section. Note that `flag0` and `flag1` are shared memory addresses.

The idea is that each process will signal its intent to enter the critical section by setting its own flag. Then it will check whether the other process has done the same. If not, it executes the critical section and signals to the caller that it was able to enter the critical section. If the other process has set its flag then it will exit early and signal to the caller that it was not able to enter the critical section.

If either of the stores on line 2 happens after the corresponding loads on line 4, the algorithm cannot guarantee mutual exclusion. This can occur on most modern processor

architectures. In particular, x86 may buffer stores delaying their global visibility, thereby allowing the stores to "move" past the loads. Consider the following execution: `p0` stores 1 to `flag0` which languishes in the store buffer, `p0` loads a value of 0 for `flag1`, `p0` enters the critical section, `p1` stores 1 to `flag1`, `p1` reads 0 from `flag0` because the store buffer of `p0` has not been flushed, `p1` enters the critical section.

Importantly, it has been shown that if both stores on line 2 happen before the corresponding loads on line 4 the algorithm will permit either one process to enter the critical section or neither process [**?**]. To ensure the correct execution of the algorithm we define a set of execution orders between the stores on line 2 and corresponding loads on line 4 that must be enforced, see the definition of O_{Dekker} in Figure **??**.

To enforce execution orders architectures provide memory fence instructions. These instructions guarantee that some subset of the supported memory operations will take place before and after the fence during execution. For x86 an appropriate memory fence is the `mfence` instruction. For Dekker's algorithm we can use an `mfence` instruction between the stores and loads to ensure that the stores happen before the fence and that the loads happen after the fence. That is we can use the fence to enforce the order.

The problem of enforcing execution orders becomes more complicated when considering portability between architectures. For Dekker's algorithm, a sequentially consistent architecture requires no fence, but on ARMv7 one of three fences `dsb`, `dmb` and `dmb st` can be used to enforce the order. Even on x86 we have a choice: a `lock xchg` instruction can be used here in place of the store and the order will be enforced.

All of these choices have an impact on the performance of the algorithm during execution. We call these choices *fence selection*. Additionally we consider the placement or insertion of fences. For Dekker's algorithm the placement is straightforward but this is not always the case.

2.2 TL2 Commit

We illustrate the complexity of *fence insertion* by examining the commit procedure of the TL2 transactional memory algorithm which we will reference throughout the paper.

Intuitively, a transactional memory [**? ?**] is a concurrent object that encapsulates and manages accesses to an array of memory locations. The TM interface has four highly concurrent methods, namely init, read, write, and commit, that a typical user program calls a large number of times.

When TL2 is managing a transaction, stores made inside the transaction do not go to main memory. Instead TL2 records the stores in a "write-set". When the transaction ends, the algorithm attempts to acquire locks for each address, commit the write-set to main memory, and release the acquired locks.

For TL2's commit to function properly the "real" stores made to each memory address from the write-set must be seen to take place before the release of the corresponding

```
1658 ...
1659 # ifndef TL2_EAGER
1660 #   ifdef TL2_OPTIM_HASHLOG
1661 for (wr = logs; wr != end; wr++)
1662 #   endif
1663 {
1664   // write the deferred stores
1665   WriteBackForward(wr);
1666 }
1667 # endif
1668
1669 // make stores visible before unlock
1670 MEMBARSTST();
1671
1672 // release locks and increment version
1673 DropLocks(Self, wv);
1674
1675 // ensure loads are from global writes
1676 MEMBARSTLD();
1677
1678 return 1;
1679 ...
```

$$O_{\mathsf{TxCommit}} \quad = \quad \{\mathsf{st}(\mathsf{addr}) \rightarrow \mathsf{st}(\mathsf{lock}),\ \ \mathsf{st}(\mathsf{lock}) \rightarrow \mathsf{ld}(x)\}$$

Figure 2. STAMP TL2 TxCommit Procedure

locks. Otherwise, an external observer may see an address in the write-set as unlocked before the actual store from the write-set makes it to main memory.

Similarly, the release of the lock for each address must be seen to take place before any load after the commit is finished. This ensures that loads performed in the same thread as the transaction will see the same values in memory from the write-set and locks as any external observer.

Figure **??** shows the source code for these execution orders from the TL2 commit procedure, `TxCommit`, as it appears in the TL2 implementation included with the STAMP benchmark suite [**?**]. The `WriteBackForward` procedure contains the store instruction that moves values from the write-set to main memory and the `DropLocks` procedure contains the store instruction that releases the locks.

Where addr represents the addresses in the write-set, lock represents the corresponding lock address, and x represents any memory address, we write these orders as a set, see Figure **??**. To enforce these orders, the TL2 *authors* have placed memory fence macros on lines 1670 and 1676. An *implementer* who is porting TL2 to different architectures can define each macro to be an architecture appropriate memory fence to enforce the correct behavior. The drawback of such a fence-centric approach is that for a programmer who wishes to understand the TM algorithm and perhaps to port it to a different architecture, a fence in itself says little about why the programmer chose it and placed it at a particular program point. Fences are best viewed as an implementation mechanism for a higher level of abstraction.

Fence Insertion. Inserting fences requires knowledge of the control flow paths between the ordered instructions, as well as the other instructions on those paths.

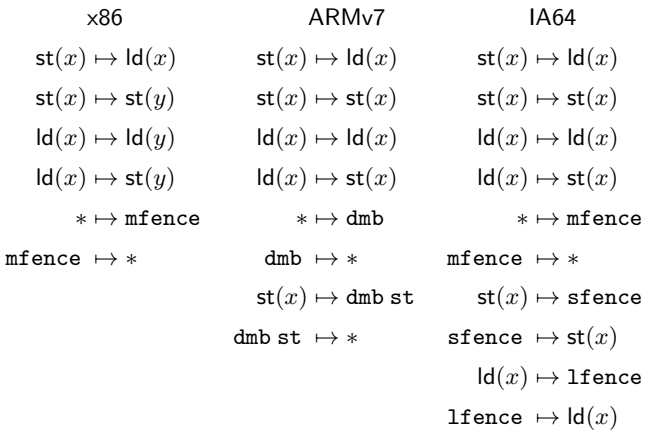

Figure 3. Architecture Definitions

Without knowing the control flow information it is possible to miss paths and allow executions in which the instructions may be seen to pass each other. On the other hand, a naive approach to fence placement that avoids missing paths by inserting a fence directly after the first instruction in the order can be expensive. For example, if the first fence macro is placed directly after the call to `WriteBackForward` it can result in an expensive loop over the fence when the `TL2_OPTIM_HASHLOG` flag is set at compile-time.

Without knowing the other instructions in the control flow paths, one might place a new fence where another already exists, or where properties of the memory models makes fences unnecessary in the presence of other instructions.

For example, consider the first order in `TxCommit`. The x86 memory model already enforces many orders. Looking at the orders enforced by the x86 architecture definition in Figure **??**, we can see $\mathsf{st}(x) \mapsto \mathsf{st}(y)$ suggests that two stores to any address will not move past each other. We use this to prove that the instructions in the first order of `TxCommit`, $\mathsf{st}(\mathsf{addr}) \rightarrow \mathsf{st}(\mathsf{lock})$, will not move past each other:

$$p, \mathsf{x86} \vdash \mathsf{st}(\mathsf{addr}) \rightarrow \mathsf{st}(\mathsf{lock})$$

That is, if x86 prevents stores from moving past each other and the order we want to enforce involves two stores, then we can conclude that the order is enforced. This means we can safely define the first macro in the example as a no-op on x86. Note that our architecture rules for x86 do not include non-temporal hinted store instructions like `movnti` and `movntdq`.

In contrast, this order is not enforced by ARMv7. Since stores can be seen to move past other stores when the addresses are different on ARMv7 we do not include this rule in the architecture definition. Instead the rule relating stores requires that the addresses be the same.

The second order in `TxCommit`, $\mathsf{st}(\mathsf{lock}) \rightarrow \mathsf{ld}(x)$, represents a more complex example of how intervening instructions can affect order enforcement. When `TxCommit` is compiled with Clang, the compiler generates a store and a load to the same temporary address after the lock release in `DropLocks` but before the end of the procedure as illustrated in both graphs in Figure **??**. We can use these instructions and the properties of x86 to prove the *transitive order* in Figure **??** where `tmp` represents the temporary memory location. We conclude that the store to lock can never be seen to move past the final load in `TxCommit` and also any subsequent load. We define the notation, rules and memory model properties more completely in the next section.

Selecting Fences. Selecting the correct fence requires knowledge of how the compiler will treat the source code and knowledge of the fences available for each architecture. We saw an example of this in Dekker's algorithm.

For `TxCommit` on ARMv7 the second macro can be defined correctly using many different fence configurations according to the ARM documentation [**?**], e.g. `dsb`, `dmb`, or a qualified `dmb st`. Both the litmus test documentation [**?**] and the assembler reference [**?**] are complicated texts in accordance with the complexity of the ARMv7 memory model. Determining the best fence is a nontrivial task.

2.3 Our Approach

We decompose fence insertion into two sub-problems. First we eliminate any orders that are provably enforced by existing instructions and the properties of the target architecture. For the remaining orders we modify the program with new instructions that enforce the remaining orders.

We address both sub-problems by considering control flow graphs with a restricted set of instructions. Each node is labeled with the instruction, the address that it operates on, and the line number that it was generated from in the source code. Every path between two instructions in the control flow graph represents a possible execution involving those instructions. We construct these graphs using the intermediate representation and control flow graph that LLVM generates for a given procedure.

At the top of Figure **??** we have a control flow sub-graph for the second order in `TxCommit`, constructed from LLVM's output when compiling the procedure. It contains all of the paths and a subset of the instructions (elided with dashed arrows) that appear between the store, $\mathsf{st}(\mathsf{lock})$, to release the locks in `DropLocks` and the end of the `TxCommit` procedure. Note that the store to release the locks appears in a cycle that comes from the body of the `DropLocks` procedure though it does not appear in the code in Figure **??**. Also the name of the variable containing the lock address has been renamed to lock for clarity.

Order Elimination. We eliminate an order by proving that the order is enforced for every path between the ordered instructions. Proofs for orders correspond with paths in the graph through a second set of *architecture edges*. If two instructions exist in a control flow path and the architecture

$$\dfrac{p, \mathsf{x86} \vdash \mathsf{st}(\mathsf{lock}) \rightarrow \mathsf{st}(\mathsf{tmp}) \qquad \dfrac{p, \mathsf{x86} \vdash \mathsf{st}(\mathsf{tmp}) \rightarrow \mathsf{ld}(\mathsf{tmp}) \quad p, \mathsf{x86} \vdash \mathsf{ld}(\mathsf{tmp}) \rightarrow \mathsf{ld}(x)}{p, \mathsf{x86} \vdash \mathsf{st}(\mathsf{tmp}) \rightarrow \mathsf{ld}(x)}}{p, \mathsf{x86} \vdash \mathsf{st}(\mathsf{lock}) \rightarrow \mathsf{ld}(x)}$$

Figure 4. Derivation of $\mathsf{st}(\mathsf{lock}) \rightarrow \mathsf{ld}(x)$ in TxCommit

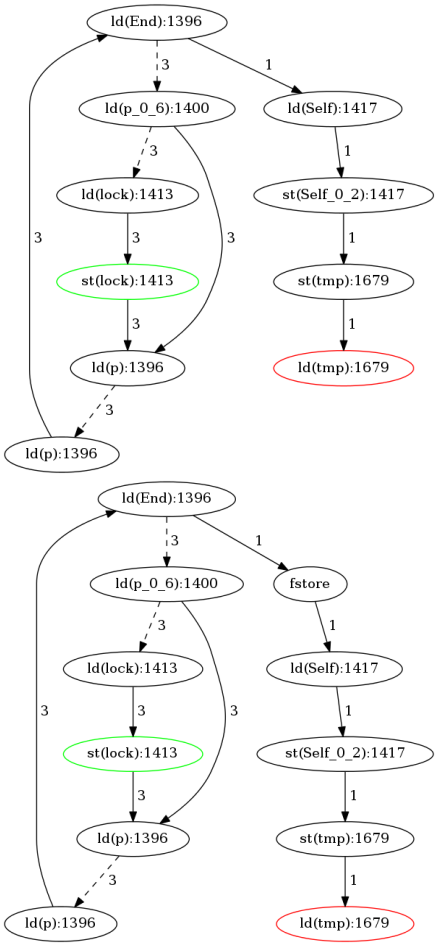

Figure 5. A graph before and after fence insertion

guarantees that they will never move past each other, we add an architecture edge.

Returning to our example, recall that on x86 we can eliminate the second order in `TxCommit` because the compiler generates a load and store to a temporary variable. At the top of Figure **??**, we add architecture edges (not pictured) between two nodes pairs. We add the first edge between the nodes labeled with `store(lock):1413` and `store(tmp):1679` because stores cannot move past stores. We add the second edge between the nodes labeled with `store(tmp):1679` and `ld(tmp):1679` because stores can not move past loads from the same address. These edges correspond with a transitive order derivation for every path between the store to lock and the load to tmp. As a result we can eliminate the order.

In contrast, on ARMv7, we cannot add the first architecture edge between the store to lock and the store to tmp because stores are permitted to move past other stores for different addresses. In that case, since we can't eliminate the order with the original graph, we must alter the graph so that we can prove the order is enforced.

Fence Insertion. We model the problem of finding these graph alterations as minimum multi-commodity cut [**?**] (hereafter multi-cut). Intuitively, multi-cut finds a minimum set of edges such that, when they are removed, no paths exist between the sources and sinks for all commodities. If we define the sources and sinks for commodities using the paired instructions in orders and then "split" the edges in the resulting cut with a fence, we will have transitive orders for all paths and all orders.

The altered control flow graph at the bottom in Figure **??** shows the results of fence insertion on ARMv7 for `TxCommit`. The algorithm selects a single edge from the original graph to split with a fence, represented here as fstore. This alteration ensures that all paths from the store to the load are provably enforced.

Note also that the cut should happen outside the loop-cycle in the control flow graph. This prevents an unnecessary performance penalty when placing the fence. This is handled directly by the minimum cut algorithm. Since the cut is determined based on the sum of the capacities of the edges in the cut, we can use larger capacities to discourage the selection of edges that occur in loops. This ensures a fence will only be placed in a loop when all paths for an order are in a loop. Modeling fence insertion as multi-cut accounts for the full generality of control flow graphs including odd control flow configurations and order overlap.

Fence Selection. Finally, we select appropriate fences for each placement by defining a partial order over fences based on their capabilities. For example on ARMv7 a `dmb` fence can enforce strictly more orders than `dmb st` since the latter only waits for stores. In the case of Figure **??** either fence will work but we prefer the "weaker" `dmb st` represented here as the abstract fence type fstore. We do this under the assumption that it is less costly during execution which is supported by the ARMv7 documentation [**?**].

2.4 Orders, Not Fences

The memory fence is a blunt instrument that relates possibly hundreds of instructions across many execution paths and blurs its original purpose. Instead we supply a scalpel in the form of declarative orders, which are more specific about the desired behavior of the program. Declarative orders at the source level enable authors to reason more easily about their algorithm, while a compiler can do the work to insert fences that enforce the orders.

3. The Fence Insertion Algorithm

Our algorithm for fence insertion takes three inputs: a control-flow graph G, architecture rules A, and declared execution orders O. The output is a transformed graph $\mathsf{Insert}(G, A, O)$ with fences that enforce those execution orders. In Section **??**.1, we define a basic version of Insert and prove it correct. The basic version relies on the simplifying assumption that the only fence available is fany. We will assume that fany enforces that all instructions prior to the fence happen before all instructions after the fence. In Section **??**.2, we describe briefly how to generalize Insert to work with multiple fences.

3.1 The Core Algorithm

We proceed as follows. First we define the basic concepts of graphs, architecture rules, and declared execution orders, along with some helper notation. Then we define a correctness criterion of the form $G, A \models O$, which says that the combination of G and A enforces all the declared execution orders O. This brings us to definition of our algorithm Insert whose goal is to produce an output graph G that satisfies $G, A \models O$. Finally, we give an example and then prove the correctness of Insert.

Graphs. A control-flow graph $G = (V, E, \ell)$ consists of a set V of nodes, a set $E \subseteq (V \times V)$ of directed edges, and a labeling function ℓ. Intuitively, a node is a program point, the label of a node is the instruction at that program point, and an edge is potential control flow between two program points. We use i, j to range over V. The function ℓ maps each element of V to a *label*, which is an element of

$$\{\ \mathsf{st}(a),\ \mathsf{ld}(a),\ \mathsf{fany}\ \}$$

where a is an address, st abbreviates *store*, ld abbreviates *load*, and fany is a fence. We use l to range over labels. Notice that our control-flow graphs focus entirely on loads, stores, and fences. This is is in contrast to the conventional notion of a control-flow graph that represents every instruction in a program. One can abstract such a conventional graph into one of our graphs by, intuitively, omitting the nodes of no interest to our approach.

For a graph G with i_1, i_2 among its nodes, we use $\mathsf{paths}(G, i_1, i_2)$ to denote the set of *paths* in G from i_1 to i_2. A path from i_1 to i_2 is itself a graph in which 1) each node has one outgoing edge, except i_2 which has no outgoing edges, and 2) all nodes on the path are reachable from i_1 by following zero, one, or more edges. Our notion of path is usually known as a *simple path* because it allows no loops. Still, we will use the terminology "path" for simplicity. We will use p to range over paths.

Architecture rules. A set of architecture rules specifies a memory model. Intuitively, the fewer the rules, the weaker the memory model. The idea is that even if a control-flow graph has an edge from i_1 to i_2, the execution of i_1 and i_2 may happen in either order or overlap, unless specific architecture rules enforce an order of execution. A set A of architecture rules consists of rules of the form $L \mapsto R$, where L, R are rule components that range over

$$\{\ *,\ \mathsf{st}(x),\ \mathsf{ld}(x),\ \mathsf{fany}\ \}$$

and where x is a variable that ranges over addresses. Intuitively, $*$ is a wildcard. A rule $L \mapsto R$ expresses that if we have a graph (V, E, ℓ) with two nodes i_1 and i_2 such that i_1 can reach i_2, and such that we can *instantiate* $L \mapsto R$ to $(\ell(i_1), \ell(i_2))$, then we can conclude that i_1 must happen before i_2. We will define the notion of instantiation below.

For example, the rules $(* \mapsto \mathsf{fany}),\ (\mathsf{fany} \mapsto *)$ express, intuitively, that fany is a fence. Specifically, the first rule says that all instructions that can reach the fence will happen before the fence, while the second rule says all instructions that can be reached from the fence will happen after the fence. The combined effect of those two rules is that all instructions prior to the fence happen before all instructions after the fence. In this section we define Insert in a way that relies on that A contains those two rules.

As another example, the rule $\mathsf{st}(x) \mapsto \mathsf{st}(y)$ expresses, intuitively, that all store instructions must happen in the order in which they are reached in the control-flow graph.

As a third example, the rule $\mathsf{st}(x) \mapsto \mathsf{ld}(x)$ expresses, intuitively, that if a store instruction to a particular address can reach a load instruction from that same address in the control-flow graph, then the store instruction must happen before the load instruction.

We will now define a notion of *instantiating* an architecture rule to a pair of labels. Specifically, if $(L \mapsto R)$ is an architecture rule and (l_1, l_2) is a pair of labels, then we write $(L \mapsto R) \rhd (l_1, l_2)$ to denote that $(L \mapsto R)$ instantiates to (l_1, l_2).

The definition of instatiation will ensure that for rules such as $(\mathsf{st}(x) \mapsto \mathsf{ld}(x))$, the two occurrences of x must be replaced with the *same* address. Our technical device to make that happen is that of a *substitution*. We use σ to range over substitutions that map variables of the form x to addresses. For our use, each substitution has either a domain of either zero, one, or two elements, depending on whether a rule mentions zero, one or two variables. The definition of $(L \mapsto R) \rhd (l_1, l_2)$ uses the relation $\blacktriangleright$ to distribute the use of a substitution to each of L and R. Now we are ready to present the detailed definition of instantiation.

We say that a rule $(L \mapsto R)$ instantiates to a pair of labels (l_1, l_2) if we can derive $(L \mapsto R) \triangleright (l_1, l_2)$ with the following rules:

$$\frac{(L, \sigma) \blacktriangleright l_1 \qquad (R, \sigma) \blacktriangleright l_2}{(L \mapsto R) \triangleright (l_1, l_2)}$$

$$\begin{array}{rcl} (*, \sigma) & \blacktriangleright & l \\ (\mathsf{st}(x), \sigma) & \blacktriangleright & \mathsf{st}(\sigma(x)) \\ (\mathsf{ld}(x), \sigma) & \blacktriangleright & \mathsf{ld}(\sigma(x)) \\ (\mathsf{fany}, \sigma) & \blacktriangleright & \mathsf{fany} \end{array}$$

The first rule says that we can instantiate $(L \mapsto R)$ to (l_1, l_2) if we can find a substitution σ such that L guided by σ instantiates to l_1 (written $((L, \sigma) \blacktriangleright l_1)$), and R guided by σ instantiates to l_2 (written $((R, \sigma) \blacktriangleright l_2)$). The other four rules define the cases where a rule component, guided by a substitution, instantiates to a label. Specifically, $*$ instantiates to any label, $\mathsf{st}(x)$ instantiates to $\mathsf{st}(\sigma(x))$, $\mathsf{ld}(x)$ instantiates to $\mathsf{ld}(\sigma(x))$, and fany instantiates to fany.

Declared execution orders. For a graph $G = (V, E, \ell)$, the declared execution orders is a set $O \subseteq (V \times V)$.

Correctness criterion. We will now define a correctness criterion $G, A \models O$. Intuitively, $G, A \models O$ says that the combination of G and A enforces all the declared execution orders O. The goal of our approach is to produce an output graph G that satifies $G, A \models O$.

We define the correctness criterion in two steps. First we define a judgment $p, A \vdash i_1 \to i_2$. For a path $p = (V, E, \ell)$, architecture rules A, and nodes i_1, i_2 on p, we define that $p, A \vdash i_1 \to i_2$ holds if it can be derived by these rules:

$$p, A \vdash i_1 \to i_2 \quad \begin{array}{l} (\text{where } i_1 \text{ can reach } i_2 \text{ in } p \wedge \\ \quad (L \mapsto R) \in A \wedge \\ \quad (L \mapsto R) \triangleright (\ell(i_1), \ell(i_2)) \ ) \end{array}$$

$$\frac{p, A \vdash i_1 \to j \qquad p, A \vdash j \to i_2}{p, A \vdash i_1 \to i_2}$$

The first rule instantiates an architecture rule in A, and the second rule is transitivity.

Now we are ready to define the overall correctness criterion. For a graph $G = (V, E, \ell)$, architecture rules A, and declared execution orders $O \subseteq (V \times V)$, define:

$$\begin{array}{l} G, A \models O \iff \\ \forall (i_1, i_2) \in O : \ \forall p \in \mathsf{paths}(G, i_1, i_2) : \ p, A \vdash i_1 \to i_2 \end{array}$$

Notice that the definition considers *all* paths between i_1 and i_2. This ensures that the declared execution order will be enforced, irrespective of the control flow.

Algorithm overview. Our algorithm Insert composes three functions Elim, Cut, and Refine. Intuitively, Insert proceeds in three steps:

1. Elim determines a subset of the declared execution orders that are enforced by the architecture.
2. Cut determines *where* to insert fences that will enforce the rest of the declared execution orders.
3. Refine inserts the fences.

We now describe Elim, Cut, Refine, and Insert. After those descriptions, we will give an example.

The Elim **function.** Our approach uses a function Elim that determines a subset of the declared execution orders for which we need *no* fences. We rely on the fact that Elim satisfies the following property:

$$\mathsf{Elim}(G, A, O) \subseteq O \text{ and } G, A \models \mathsf{Elim}(G, A, O). \qquad (1)$$

Programmers can implement $\mathsf{Elim}(G, A, O)$ in many ways, including the trivial approach that always returns the empty set. Our implementation, as a default, uses a straightforward exponential-time algorithm that for each $(i_1, i_2) \in O$ enumerates all $p \in \mathsf{paths}(G, i_1, i_2)$, and for each such p uses brute-force to determine whether $p, A \vdash i_1 \to i_2$. The result is that $\mathsf{Elim}(G, A, O)$ returns a maximal subset of O. The maximal size helps us insert few fences.

In addition we have implemented a linear time approximation algorithm which works by finding enough nodes i with the property, $\{(i_1, i), (i, i_2)\}, A \vdash i_1 \to i_2$ such that, when every i is removed $\mathsf{paths}(G, i_1, i_2) = \emptyset$.

In either case, we need no modifications to G to enforce the orders in $\mathsf{Elim}(G, A, O)$ so now let us focus on where to insert fences to enforce the orders in $O \setminus \mathsf{Elim}(G, A, O)$.

The Cut **function.** Our approach uses a function Cut that determines *where* to insert fences. We rely on that Cut satisfies the following property:

$$\mathsf{Cut}(G, O) \text{ is a } \textit{multi-cut} \text{ for } G, O. \qquad (2)$$

The multi-cut specifies where to insert fences. Let us recall the standard notion of a multi-cut [**?**]: given a graph $G = (V, E, \ell)$ and a set $O \subseteq (V \times V)$, a multi-cut for G, O is a set K, where $K \subseteq E$, such that $\forall i_1, i_2 \in O$: $\mathsf{paths}((V, E \setminus K, \ell), i_1, i_2) = \emptyset$. Programmers can implement $\mathsf{Cut}(G, O)$ in many ways, such as the trivial approach that always returns E, an approximation algorithm [**?**], and an integer linear program [**?**]. We experimented with those and chose an ILP with a polynomial number of constraints in the size of the graph [**?**]. We use SAGE [**?**] and the default solver GLPK [**?**] to solve the ILP, which returns a multi-cut of minimal size, which in turn helps us insert few fences. Given G, O and an integer n, the problem to decide whether there exists a multi-cut for G, O with at most n elements is NP-complete for $|O| > 2$ [**?**]. Now let us consider *how* to use a multi-cut to insert fences.

The Refine **function.** Our approach uses a function Refine that inserts fences. We will give the definition of Refine in detail and later we will prove that the definition satisfies four lemmas. For a graph $G = (V, E, \ell)$ and a cut-set K, where $K \subseteq E$, the function $\mathsf{Refine}(G, K)$ creates a set W_K of additional nodes (fences!), and replaces each

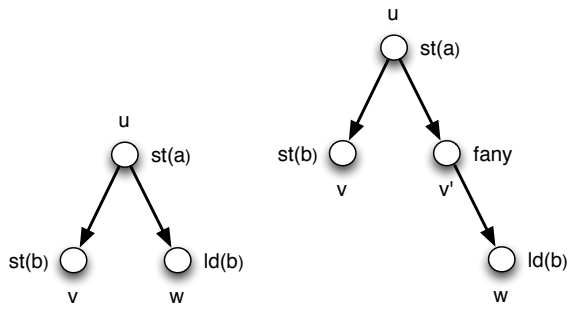

Figure 6. An example graph and its refinement.

$(j_1, j_2) \in K$ with two new edges that, intuitively, insert a fence between j_1 and j_2. The new nodes form a set W_K:

$$W_K \quad ::= \quad \{v_{j_1,j_2} \mid (j_1, j_2) \in K\}$$

where each v_{j_1,j_2} is a fresh node. The output graph is:

$$\begin{array}{rcl} \mathsf{Refine}(G, K) & = & (V \cup W_K,\ (E \setminus K) \cup E_K,\ \ell_K) \\ E_K & = & \{(j_1, v_{j_1,j_2}), (v_{j_1,j_2}, j_2) \mid v_{j_1,j_2} \in W_K\} \\ \ell_K & = & \ell \cup \{(v_{j_1,j_2}, \mathsf{fany}) \mid (j_1, j_2) \in K\} \end{array}$$

Notice that $\mathsf{Refine}(G, \emptyset) = G$.

The Insert **function.** We can now define Insert:

$$\mathsf{Insert}(G, A, O) \quad = \quad \mathsf{Refine}(G, \mathsf{Cut}(G, O \setminus \mathsf{Elim}(G, A, O)))$$

The definition calls three functions as outlined above: first Elim, then Cut, and finally Refine. Both Elim and Cut run in worst-case exponential time, while set difference and Refine run in polynomial time, so we conclude that Insert runs in worst-case exponential time.

Example. Consider the graph $G = (V, E, \ell)$, which is illustrated in Figure **??** (left graph). We have

$$\begin{array}{rcl@{\qquad}rcl} V & = & \{u, v, w\} & \ell(u) & = & \mathsf{st}(a) \\ E & = & \{(u, v), (u, w)\} & \ell(v) & = & \mathsf{st}(b) \\ & & & \ell(w) & = & \mathsf{ld}(b) \end{array}$$

where a, b are distinct addresses. Consider also the set of architecture rules:

$$\begin{array}{rcl} A & = & \{(* \mapsto \mathsf{fany}),\ (\mathsf{fany} \mapsto *), \\ & & (\mathsf{st}(x) \mapsto \mathsf{st}(y)),\ (\mathsf{st}(x) \mapsto \mathsf{ld}(x))\} \end{array}$$

Consider finally the specified execution orders

$$O \quad = \quad \{(u, v), (u, w)\}$$

A run of $\mathsf{Insert}(G, A, O)$ will proceed as follows.

The first step is to call $\mathsf{Elim}(G, A, O)$. This call to Elim will find that $G, A \models \{(u, v)\}$ because we have a single-edge path from u to v, and a rule $(\mathsf{st}(x) \mapsto \mathsf{st}(y)) \in A$ that instantiates to $(\ell(u), \ell(v))$, which is equal to $(\mathsf{st}(a), \mathsf{st}(b))$. Thus, if p is the single-edge path from u to v, we can derive $p, A \vdash u \to v$. The call to Elim will also find that $G, A \not\models \{(u, w)\}$ because we have no rule in A that for the single-edge path p' from u to w enables us to derive $p', A \vdash u \to w$. Note here that the rule $(\mathsf{st}(x) \mapsto \mathsf{ld}(x)) \in A$ requires the two instructions (store and load) work with the same address, while the two labels $\mathsf{st}(a)$ and $\mathsf{st}(b)$ operate on distinct addresses. In summary, we have $\mathsf{Elim}(G, A, O) = \{(u, v)\}$. We can calculate $O \setminus \{(u, v)\} = \{(u, w)\}$.

The second step is to call $\mathsf{Cut}(G, O \setminus \mathsf{Elim}(G, A, O) = \mathsf{Cut}(G, O \setminus \{(u, v)\} = \mathsf{Cut}(G, \{(u, w)\})$. In this case we have $\mathsf{Cut}(G, \{(u, w)\}) = \{(u, w)\}$. The reason is that the one path from u to w has a single edge so we must have that edge in the multi-cut.

The third step is to call

$$\begin{array}{rcl} \mathsf{Insert}(G, A, O) & = & \mathsf{Refine}(G, \mathsf{Cut}(G, O \setminus \mathsf{Elim}(G, A, O))) \\ & = & \mathsf{Refine}(G, \{(u, w)\}) \end{array}$$

The result is illustrated in Figure **??** (right graph). Compared to G, the graph $\mathsf{Insert}(G, A, O)$ has an additional node v', no edge $\{(u, w)\}$, but instead two edges (u, v') and (v', w). We have $\ell(v') = \mathsf{fany}$, that is, the new node is a fence. We can instantiate the rule $(* \mapsto \mathsf{fany}) \in A$ to $(\mathsf{st}(a), \mathsf{fany})$, and we can instantiate the rule $(\mathsf{fany} \mapsto *) \in A$ to $(\mathsf{fany}, \mathsf{ld}(b))$. So for the path p'' with the two edges (u, v') and (v', w), we have that we can derive $p'', A \models (u, w)$.

In summary, the example has two specified execution orders, and one of them is enforced by the architecture without insertion of any fences, while for the other, we inserted a single fence.

Theorem 1. *If* $\{(* \mapsto \mathsf{fany}), (\mathsf{fany} \mapsto *)\} \subseteq A$, *then* $\mathsf{Insert}(G, A, O), A \models O$.

We prove Theorem **??** in Appendix B.

3.2 Multiple Kinds of Fences

Let us now relax the assumption that the only fence available is fany. For example, Figure **??** gives rules for the two fences on ARMv7, namely the weaker fence `dmb st` and the stronger fence `dmb`. When multiple fences are available, the Refine function can choose as weak a fence as possible. The idea is that a weaker fence executes faster than a stronger fence, which is true for the architectures we have considered. Intuitively, we let Refine choose the weakest fence that enforces the relevant declared executions orders.

We will explain how to make the choice in two steps. First let us consider a simple case, which happens to be the one we encountered exclusively in our experiments. For an edge (j_1, j_2) in a cut set K, suppose we have a single element $(i_1, i_2) \in O \setminus \mathsf{Elim}(G, A, O)$ for which (j_1, j_2) is on a path from i_1 to i_2. We must chose a fence that is strong enough to enforce the order (i_1, i_2). Specifically, we need a fence f such that A contains the rules $(L \mapsto f)$ and $(f \mapsto R)$ such that

$$(L \mapsto f) \rhd (i_1, f) \ \wedge \ (f \mapsto R) \rhd (f, i_2)$$

We will choose as weak a fence as possible. For the archictectures we have considered, we can always find a fence that is the weakest among all those that satisfy the above requirement.

Now let us consider the general case. For an edge (j_1, j_2) in a cut set K, suppose we have multiple elements $(i_1, i_2) \in O \setminus \mathsf{Elim}(G, A, O)$ for which (j_1, j_2) is on a path from i_1 to i_2. Here we want the least expensive fence that will enforce *all* of the involved orders. For each such (i_1, i_2) we chose a fence $f_{(i_1,i_2)}$ as described above. Now we need a fence that is at least as strong as those fences $f_{(i_1,i_2)}$. Again, for the archictectures we have considered, we can always find a weakest fence that is at least as strong as each $f_{(i_1,i_2)}$.

Optimality. The optimality of our approach to fence selection depends on the assumption that any two fences will always execute more slowly than any single fence. Consider the case of an architecture like IA64 in Figure **??** and some procedure where we have a load-load order and a store-store order overlapping on a single simple path. Individually the orders can be enforced with an `lfence` *and* an `sfence` but the optimal multi-cut will include an edge shared by both orders. Then to satisfy both orders with a single fence we must select an `mfence`.

4. Implementation

We have implemented our approach in a tool called Parry [**?**] that as input takes a concurrent algorithm written in C/C++, declared execution orders, and a memory model, and as output produces C/C++ with fences. Parry uses Python to orchestrate the three major tasks in fence insertion: control-flow graph generation, order elimination, and fence insertion. We will now explain some details of Parry, particularly a few points that go beyond the fence insertion algorithm that we described in Section 3.

Graph Generation Parry is based on LLVM. First Parry compiles the input source code to LLVM's static-single-assignment (SSA) intermediate representation (IR) along with debugging information. Then Parry generates a control-flow graph of the target procedure using LLVM's `opt` tool. Next, Parry simplifies the control-flow graph by replacing each standard block with a path of instructions. We manipulate the resulting graphs with the graph-tool library [**?**].

We construct a graph in which the only nodes are for load, store, call, and `cmpxchg` instructions. Note that compared to the algorithm in Section 3, we add call instructions to safely account for methods which do not get inlined, and we add the `cmpxchg` instructions because they are frequently used by authors to enforce orders. Indeed, the `cmpxchg` instruction provides compare-and-swap semantics at the LLVM IR level and can act as a full memory fence (like `mfence` on x86 or `dmb` on ARMv7) [**?**].

Compiler Assumptions Parry uses Clang to translate C/C++ into LLVM's intermediate representation and we assume that this translation preserves some key aspects of the code that are of interest to Parry.

We assume that the semantics of a line of C/C++ used to specify an order will be preserved in the intermediate representation generated by Clang. For example, if the programmer expects a store to a certain memory address at a line in the code then we assume that Clang will generate a store to that address for that line. We safely account for the possibility of more than one instruction per line matching the event types of an order by including all matching instructions during the analysis.

We also assume that our fence placements will remain valid after a compiler optimizes the C/C++ code that Parry outputs. That is, we require that the ordered instructions will not be moved past the fences by the compiler. To that end we ensure that all inline assembly instructions inserted by Parry are marked as volatile operations.

Architecture Rules As detailed in Section **??** we have created a set of rules for each architecture that describe the which instructions won't "move" during execution. These rules are necessary for order elimination.

We have included three architectures in Figure **??**, x86, ARMv7 and IA64. The last is for clarity and comparison since our evaluation does not include benchmarks for IA64.

We compiled these rules based on our interpretation of the processor documentation available for each architecture. They are intended to be an over-approximation of the actual architecture behavior. During order elimination they are used to establish preexisting orders without consideration for other types of instructions aside from stores, loads, and fences.

Not included in Figure **??** are instructions that exist in the LLVM IR, like `cmpxchg`, which result in hardware instructions with fence-like semantics. We do account for LLVM's `cmpxchg` as detailed in Section **??**.

Edge Elimination Parry has an initial step that takes place before the main algorithm in Section 3: edge elimination. The idea is to eliminate all edges that are irrelevant to the fence insertion problem. We keep an edge only if for at least one declared execution order, the edge is on a path from the source to the sink of the order *and* the instructions on that path don't enforce the order. After edge elimination, we can implement order elimination for an order (i_1, i_2) simply by checking whether the set of paths from i_1 to i_2 is empty.

Address Equality In some cases, an architecture rule uses a variable twice, such as the rule $(\mathsf{st}(x) \rightarrow \mathsf{ld}(x))$. Our tool only instantiates the rule in case x can be replaced with a variable in LLVM's internal static-single-assignment form. The SSA form guarantees that the value of that variable is the same at both program points. For example, in Figure **??**, `tmp` is a variable, so we can instantiate $(\mathsf{st}(x) \rightarrow \mathsf{ld}(x))$ to $(\mathsf{st}(\texttt{tmp}), \mathsf{ld}(\texttt{tmp}))$.

Fence Insertion Since our analysis is static and we want to minimize the execution of fence instructions, we avoid placing fences in loops unless absolutely necessary. Parry achieves this by finding cycles in the control flow graph. Then it assigns an edge weight to the cycle edges that is one more than twice the incoming edge weight as illustrated earlier in Figure **??**. This ensures that even if many orders from outside the loop overlap inside the loop the linear program will prefer edges outside the loop. This heuristic has value if a loop is executed more than once.

Alternate Fence Placements In many cases there are multiple fence placements that are equivalent according to the multi-cut model. That is, there may be edges with weights on similar paths resulting in the same objective function value from the multi-cut linear program. Our implementation selects the edge closest to the source. Our tool is also able to select alternate cuts where necessary and we discuss our experimental evaluation of equivalent alternative fence placements in Section **??**.

5. Experimental Results

We have evaluated Parry with four classic concurrent algorithms (Dekker, Lamport, Parker, and Peterson) and three transactional memory algorithms (TL2 [**?**], TL2 Eager, and TLRW [**?**]). We downloaded implementations of the classic algorithms from the Musketeer project [**?**]), TL2 and TL2 Eager are from the STAMP project [**?**], and TLRW from the Rochester Software Transactional Memory library [**?**]. TL2 Eager is a variant of TL2. The TLRW implementation is named ByteEager. The three TM algorithms have procedures that are significantly more complex than those of the classic algorithms. For example, TL2 has nearly 400 nodes and more than 250 lines of code in one procedure.

We evaluate all of the algorithms on the x86 and ARMv7 architectures. We chose to work with ARMv7 due to its increasing relevance in all types of computing and its particularly weak memory model, compared to x86.

5.1 Declaration of Execution Orders

We declared execution orders for each of the seven algorithms to benchmark our approach against the author supplied memory fences. We also removed existing fences from the algorithms.

Classic Algorithms Figure **??** shows the orders for the four classic algorithms. We got the orders for Dekker from Lesani's dissertation [**?**] and the orders for Peterson are similar. We got the single order for Parker from a blog post by Dave Dice. Dice wrote that the Parker implementation in the Java Virtual Machine was found to have a bug due to store buffering [**?**]. We defined an order according to the description of the bug to ensure that the store to the shared variable `_counter` is flushed. We defined the orders for Lamport based on an analysis that we detail in an appendix.

TL2 and TL2 Eager Figure **??** shows the orders for TL2 and TL2 Eager which we got from Lesani's dissertation [**?**]. The source code for both algorithms can be found at [**?**].

RSTM ByteEager Figure **??** shows the orders for ByteEager. The source code for RSTM ByteEager algorithm can be found at [**?**]. The orders stem from the work on TLRW by Dice et. al [**?**]. They give a detailed account of critical orders which we use here.

During the process of defining orders for each algorithm we also had to find and remove existing fences to prevent duplication. On closer inspection of the code for the store-store order in the `rollback` procedure of ByteEager we were unable to find any mechanism that might enforce the order.

We contacted the original authors to verify our findings. It became clear that they had built the algorithm for architectures where the order was automatically enforced, namely TSO, and they agreed a fence was necessary. We placed a `dmb st` fence after the source of the order to establish a baseline for comparison, noting that an order definition would have made our communication unnecessary.

Difficulty The only algorithm without orders already defined in some form was Lamport's mutex. Every other algorithm had research, implementation notes, or existing fences from which orders could be derived. We think this means execution order definition is already implicitly taking place during algorithm design but the information is lost as fence placements during implementation.

Further, in the case of ByteEager's `rollback` procedure, we believe that the speed with which the ByteEager authors were able to diagnose the issue suggests that authors and designers will have relatively little difficulty in defining execution orders during algorithm design.

5.2 Parry's Execution Time

Figures **??** and **??** show the wall clock time that Parry's top-level `run` procedure takes to insert fences for the TL2 and ByteEager TM algorithms. Figure **??** shows results where the exponential order elimination algorithm was used and Figure **??** shows results for the linear order elimination algorithm. Notably, the elimination results from the exponential time and linear time order elimination algorithms are identical for all of the evaluated code.

The times were recorded from each stage of Parry's execution, averaged across 100 runs on an Intel Core i5 at 2.4 Ghz with 6GB of RAM with a fully updated version of Ubuntu 14.04 Server.

We include only the TM algorithms here because they are the most complex examples in our evaluation. The size of each procedure for both algorithms in lines of code and control flow graph notes is included in Figure **??**. The lines of code are recorded without account for inlining except in the case of TL2:`TxCommit` where the majority of the instructions are inlined from a procedure call. The largest

		x86	ARM7
Dekker	$8 \xrightarrow{\text{st,ld}} 9$	8:`mfence`	8:`dmb st`
	$13 \xrightarrow{\text{st,ld}} 9$	13:`mfence`	13:`dmb st`
	$25 \xrightarrow{\text{st,ld}} 26$	25:`mfence`	25:`dmb st`
	$30 \xrightarrow{\text{st,ld}} 26$	30:`mfence`	30:`dmb st`
Lamport	$8 \xrightarrow{\text{st,ld}} 9$	8:`mfence`	8:`dmb st`
	$14 \xrightarrow{\text{st,ld}} 15$	14:`mfence`	14:`dmb st`
	$31 \xrightarrow{\text{st,ld}} 32$	31:`mfence`	31:`dmb st`
	$37 \xrightarrow{\text{st,ld}} 38$	37:`mfence`	37:`dmb st`
Parker	$44 \xrightarrow{\text{st,*}} 46$	44:`mfence`	44:`dmb st`
Peterson	$5 \xrightarrow{\text{st,ld}} 7$	5:`mfence`	5:`dmb st`
	$14 \xrightarrow{\text{st,ld}} 16$	14:`mfence`	14:`dmb st`

Figure 7. Orders and fences for four classic algorithms

TL2	LOC	Nodes
`TxLoad`	75	171
`TxStore`	121	236
`TxCommit`	277	398
ByteEager	**LOC**	**Nodes**
`read_ro`	30	64
`read_rw`	32	73
`write_ro`	31	93
`write_rw`	36	122
`rollback`	25	93

Figure 8. Algorithm Procedure Size

case is TL2:`TxCommit` procedure where the control-flow graph has 398 nodes. We have not included TL2 Eager since the size and times were similar to those of regular TL2.

The control-flow graph generation and order elimination account for the majority of the execution time. In the cases where the linear order elimination algorithm is used the graph generation dominates the other parts of our approach. The long execution times for graph generation are caused by a large amount of string manipulation and scanning while working with the LLVM IR in Python. The fence insertion which uses GLPK to run our integer linear program takes little time.

When executing on TL2 for x86 no time is spent on the ILP and a small amount on order elimination. Parry can forgo running the linear program entirely because all of the orders are eliminated. The order elimination only requires a small amount of time because it is immediate for orders where the source and sink instructions exist in an architecture enforced relationship.

5.3 Experiments with the Four Classic Algorithms

For the four classic algorithms, Parry inserted the fences shown in Figure **??**. Notice that each order led to one fence. In each case, the fence is correctly placed and is the best fence possible. We note that Lamport's mutex has two "loops" due to jumps to the start of the algorithm; Parry places a fence directly after the first branch, which is a good choice.

5.4 Transactional Memory Algorithms

For the three TM algorithms, Parry inserted the fences shown in Figure **??** and Figure **??**. We have an opportunity to compare those fences with a baseline. The original authors of the transactional memory algorithms inserted fences or fence macros for particular architectures, which we assume are correct for the intended architectures. From those fence placements, the literature, and in some cases consultation with the original authors, we constructed a baseline for each of x86 and ARMv7. Effectively, we acted as an implementer who selects fences; for example, for TL2 we defined an existing fence macro called `MEMBARSTLD` as `mfence` on x86. Similarly, ByteEager uses a memory fence macro `WBR` and the implicit memory barrier defined by the semantics of the `__sync_*` compiler built-ins [**?**]. One goal of our experiments is to evaluate whether our order definitions and Parry can match the baseline. In the following subsections, we will give a detailed comparison of both the fence placements and of the resulting performance of the TM algorithms on standard benchmarks.

5.5 Impact of Order Elimination

Figure **??** shows that for TL2 on x86, Parry eliminated all 7 orders and inserted no fences at all, while on ARMv7, Parry eliminated no orders. Additionally, Figure **??** shows that for TL2 Eager on x86, Parry eliminated all 4 orders and inserted no fences at all, while on ARMv7, Parry eliminated one order. Finally, Figure **??** shows that for ByteEager on x86, Parry eliminated one order, while on ARMv7, Parry eliminated no orders.

Now let us compare with the baseline. We ask: (1) are there cases where order elimination is necessary to approximate the fence placements from a knowledgeable implementor and (2) can we avoid adding fences altogether using information from the compiler? Our comparison suggests the answer is *yes* in both cases, as we will detail now.

Order elimination prevents the addition of extra fences where the architecture directly enforces an order and an implementer will never insert a fence. For example, load-load orders are automatically enforced on x86. In such cases we can establish a derivation by instantiating an architecture axiom directly and then eliminate the corresponding order. There are also situations like the write procedure for ByteEa-

ger where accounting for the `cmpxchg` instruction prevents the insertion of an additional fence.

Additionally, we have exhibited two instances where fences would likely be inserted by an implementer but which actually require no fence. If TL2 is compiled with the `TL2_EAGER` flag, one fence in `TxCommit` can be eliminated on architectures like ARMv7 since the source of the order will not appear in the control flow graph. If TL2 is compiled by Clang on x86, another fence can be eliminated in `TxCommit` due to generated instructions which allow for a transitive order derivation. In these cases a detailed accounting of the control flow graph is important in determining whether an order is already enforced.

5.6 Performance Benchmarks

We compare the performance of Parry's output with the baseline using six of the benchmarks from the STAMP benchmark suite v0.9.10 [**?**] which is designed for testing transactional memory algorithms. We ran the x86 benchmarks on an Intel Core i5 at 2.4 Ghz with 6GB of RAM with a fully updated version of Ubuntu 14.04 Server. The ARMv7 benchmarks were run on an Exynos 5 Dual at 1.7Ghz with 2GB of RAM with the same operating system.

We compiled the results of each benchmark by taking the arithmetic mean of each over 100 runs and then recording the percentage difference between the baseline version and the Parry version.

Importantly everything was compiled with Clang version 3.3. This is the same version used in Parry to generate IR, control flow graphs. Using the same version of Clang to generate the control flow graphs and to compile the algorithms ensures that the assertions we make about the graphs remain valid for the final compiled output.

5.7 TL2 and TL2 Eager Measurements

Figure **??** shows the fences inserted by Parry alongside the fences placed in the baseline. We associate them by the orders we defined for TL2. For example the last two orders for TL2 correspond with the orders from our running example, `TxCommit`. All of the orders were accounted for by fence macros. This is not surprising given that the authors would have a deep understanding of the algorithm's behavior under weak memory models.

The orders are defined using line numbers which can be referenced in the code which accompanies our project [**?**]. We have included annotations for the instruction types that should be ordered when they appear in the intermediate representation. For example the order between lines 760 and 1413 in `TxCommit` is a store-store order between the store of a value in the write-set and a store to release the lock for the write-set's address. In this case the line numbers appear to be abnormally distant from one another but, due to procedure inlining, they both appear in the control flow graph for `TxCommit`. We have also included a mapping from

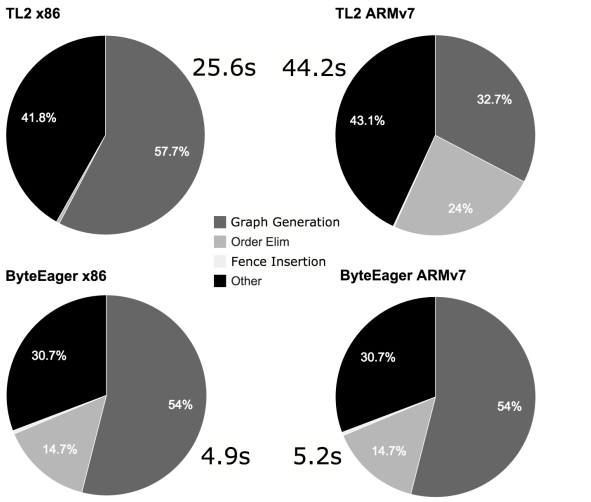

Figure 9. Parry Execution Times, Full Order Elimination

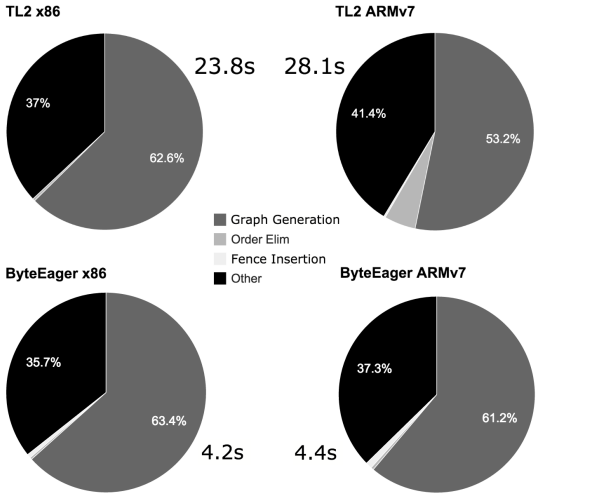

Figure 10. Parry Execution Times, Linear Elimination

TL2		**x86**		**ARM7**
TxLoad	baseline	ours	baseline	ours
2078 $\xrightarrow{\text{ld,ld}}$ 2080	—	—	2078:dmb	2078:dmb
2080 $\xrightarrow{\text{ld,ld}}$ 2082	—	—	2080:dmb	2080:dmb
TxStore	baseline	ours	baseline	ours
1886 $\xrightarrow{\text{ld,ld}}$ 1923	—	—	1920:dmb	1886:dmb
TxCommit	baseline	ours	baseline	ours
1555 $\xrightarrow{\text{st,st}}$ 1625	—	—	1555: ldrex/strex	1555:ldrex/strex
1596 $\xrightarrow{\text{st,st}}$ 1625	—	—	1596: ldrex/strex	1596:ldrex/strex
760 $\xrightarrow{\text{st,st}}$ 1413	—	—	1669:dmb st	1669:dmb st
1413 $\xrightarrow{\text{st,ld}}$ 1679	1679:mfence	—	1679:dmb st	1416:dmb st
TL2 Eager		**x86**		**ARM7**
TxLoad	baseline	ours	baseline	ours
1991 $\xrightarrow{\text{ld,ld}}$ 1993	—	—	2078:dmb	2078:dmb
1993 $\xrightarrow{\text{ld,ld}}$ 1995	—	—	2080:dmb	2080:dmb
TxCommit	baseline	ours	baseline	ours
760 $\xrightarrow{\text{st,st}}$ 1413	—	—	1669:dmb st	—
1413 $\xrightarrow{\text{st,ld}}$ 1679	1679:mfence	—	1679:dmb st	1679:dmb st

Figure 11. Orders and fences for TL2 and TL2 Eager

	lines	orders
`TxCommit`	$760 \xrightarrow{\mathsf{st,st}} 1413$	$\mathsf{st}(\mathsf{addr}) \rightarrow \mathsf{st}(\mathsf{lock})$
	$1413 \xrightarrow{\mathsf{st,ld}} 1679$	$\mathsf{st}(\mathsf{lock}) \rightarrow \mathsf{ld}(x)$

Figure 12. TL2 Lines to Orders

the orders in our running example to the line number orders in Figure **??**.

TL2 x86 The load-load orders in `TxLoad` and `TxStore` and the store-store orders in `TxCommit` should be defined without a fence by an implementer since those instructions can never be seen to move past one another. Parry makes the same determination. As noted earlier we were able to eliminate the final store-load order in `TxCommit` entirely due to Clang's IR output and the transitive order as derived in Figure **??**. It's unlikely that an implementer would have enough information to make the same determination without the help of our tool.

TL2 ARMv7 For both orders in `TxLoad`, the order in `TxStore`, and for the last two orders in `TxCommit`, Parry inserted the same fence as the authors. Parry also matched the authors for the first two orders in `TxCommit`. They begin with `cmpxchg` instructions which compile to a paired `ldrex/strex` on ARMv7 and require no additional fence.

Note, that the result for the last order in `TxCommit` and the only order in `TxStore` are at different line numbers. Both orders have a path with many edges of the same capacity that can all serve to separate the source from the sink in a minimal cut. The algorithm simply chooses the edge closest to the source in this case. Also, the line number where Parry placed the fence for the second order of `TxCommit` is seemingly before the source of the order. This is due to procedure inlining.

TL2 Eager The eager version of TL2 requires fewer execution orders than the full TL2. The orders in `TxLoad` correspond with the same orders for the full TL2. Otherwise, the difference is the first order in `TxCommit`. As noted in Section **??**, when the algorithm is compiled as eager the source of the first order is removed from the control flow graph by the first `ifndef` in Figure **??**. This makes the order unnecessary since it does not appear in that procedure's control flow graph.

Performance Let us consider the performance of the TL2 algorithm; the results for TL2 Eager are similar and we omit them here. The performance results in Figure **??** for TL2 follow intuition quite closely. In the case of x86 we saw a small improvement since we were able to eliminate a fence in `TxCommit`. The improvement is small because `TxCommit` is called infrequently when compared with `TxLoad` and `TxStore` which are the most heavily used procedures in any transactional memory algorithm. Similarly, the results for

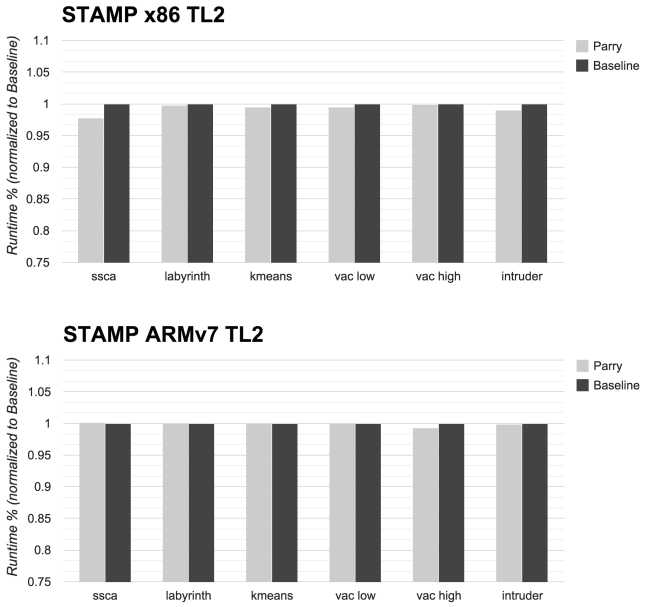

Figure 13. TL2 Performance Benchmarks

ARMv7 for the baseline and Parry are nearly identical since the number and placement of fences are nearly identical.

Alternate Fence Placements To test whether different fence placements that are considered equivalent by the multi-cut model might produce real performance differences we considered five alternate fence placements for the TL2 algorithm on ARMv7. We chose this benchmark because TL2 on ARMv7 has the largest number of alternate placements. This is due to the large control flow graphs for the procedures of TL2 and the weaker memory model of ARMv7.

Our results showed that, for every alternate placement, for all six of the STAMP performance benchmark results, the difference between the default fence placement provided by Parry and the alternate was within $\pm 4\%$. Further only three benchmarks of a total thirty showed greater than a 3% change and two showed greater than 2% change. This suggests that choosing the default is reasonable where performance is concerned.

5.8 RSTM ByteEager Measurements

The table in Figure **??** contains the fence placements for the RSTM ByteEager implementation along with the memory fences already present in the implementation. Again we note that all orders except the last in `rollback` are accounted for in the implementation of the algorithm. We believe this is due to the author's intimate knowledge of the intended behavior and the detailed account of the fence placements for TLRW in [**?**].

x86 The placements here are noteworthy due to the `xchg` instructions present in the baseline version of `read_ro`, `read_rw`, `write_ro`, and `write_rw`. The source for the orders in the `read_ro` and `read_rw` procedures are calls to

ByteEager	x86		ARM7	
read_ro	baseline	ours	baseline	ours
125 $\xrightarrow{st,ld}$ 128	125:xchg	125:mfence	125:ldrex/strex	125:dmb st
read_rw	baseline	ours	baseline	ours
163 $\xrightarrow{st,ld}$ 165	163:xchg	163:mfence	163:ldrex/strex	163:dmb st
write_ro	baseline	ours	baseline	ours
186 $\xrightarrow{st,ld}$ 196	186:xchg	186:xchg	186:ldrex/strex	186:ldrex/strex
write_rw	baseline	ours	baseline	ours
228 $\xrightarrow{st,ld}$ 238	228:xchg	228:xchg	228:ldrex/strex	228:ldrex/strex
rollback	baseline	ours	baseline	ours
261 $\xrightarrow{st,st}$ 265	—	—	266:dmb st	266:dmb st

Figure 14. Orders and fences for RSTM ByteEager

```
1658 void set_read_byte(uint32_t id) {
1659 #if defined(BASELINE)
1660   __sync_lock_test_and_set(&r[id], 1u)
1661 #else
1662   // NOTE: no WBR macro
1663   r[id] = 1;
1664 #endif
1665 }
```

Figure 15. RSTM

another procedure which is not inlined and which uses the `__sync_lock_test_and_set` compiler built-in. When targeting x86 this built-in compiles to the `xchg` instruction which includes an implicit `lock` prefix. This prevents other stores and loads from moving past the `xchg` [**?** , p. 160]. As a result it can do the same job as the `mfence` that Parry inserts. For our benchmarks we compared the version using the compiler built-in and a modified version which relies on Parry to insert the proper fence as illustrated in Figure **??**. This accurately reflects how an author might rely on an order definition to enforce the correct behavior without the compiler built-in. It also mirrors the definition for SPARC in the original source (not depicted in Figure **??**) which is identical to the version we use to test Parry, except that it includes an explicit `WBR` fence macro.

For `write_ro` and `write_rw` the author's implementation uses the `__sync_bool_compare_and_swap` built-in at the source of the orders. This built-in translates to a `cmpxchg` LLVM IR instruction with a sequential consistency ordering qualifier which in turn compiles to the `xchg` instruction. Parry accounts for the `cmpxchg` by treating it as a fence per the LLVM documentation [**?**], and consequently it does not add an additional fence to these procedures.

ARMv7 The placements are the same for `read_ro` and `read_rw` as they are for x86 but here the compiler built-in results in a paired `ldrex/strex`. Since we have implemented our own simple read, Parry again inserts a qualified `dmb st`. As with x86, when considering the `write_ro` and `write_rw` procedures Parry accounts for the `cmpxchg` instruction and does not add an additional fence.

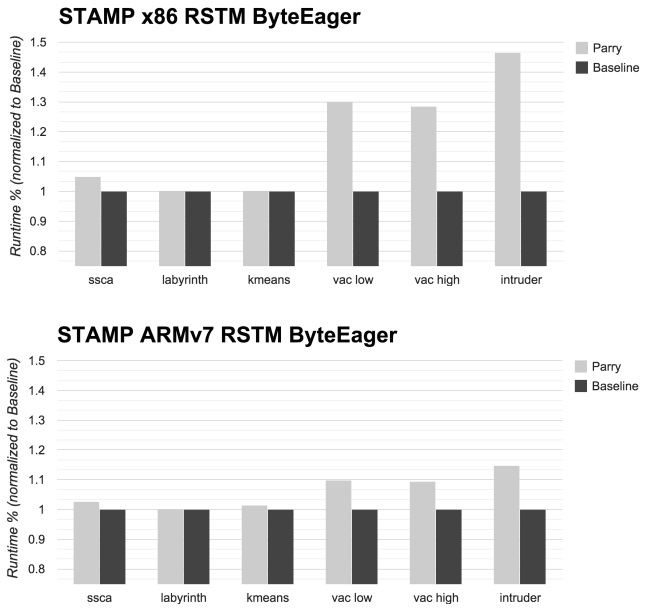

Figure 16. RSTM ByteEager Performance Benchmarks

In the `rollback` procedure a store-store order is required to preserve the expected TLRW behavior when aborting a transaction. We were unable to find anything that would enforce a store-store order in the original implementation. As discussed we contacted the original authors and determined that they did not consider store-store orders when building the algorithm. We placed an appropriate fence for the baseline which Parry matched.

Performance On x86 we expected to see similar results for both Parry and the baseline across benchmarks but in some cases Parry's compiled output was almost 1.5 times slower. Upon further inspection the overhead is due to the modifications we made in Figure **??**. Both the `mfence` and `xchg` instructions can act as an appropriate memory fence for the orders in `read_ro` and `read_rw` and they have similar execution costs to the best of our knowledge. They differ, in that `xchg` also handles the store to memory where our modifications perform the store using a separate instruction.

To test this, we removed the `mfence` from the version which relies on a store and fence. The benchmark results with just a store instruction were comparable to the original using the `xchg`. This suggests that, in these benchmarks, the `xchg` is only marginally more expensive than the store. In future work we could address this by using the `xchg` to enforce orders by replacing the store entirely.

On ARMv7 we see a smaller difference though clearly the same issue exists here: the store and the `dmb st` fence are more expensive than the paired `ldrex/strex`.

For both architectures the fence inserted by Parry is only subtly different from the code generated by the compiler built-in but the performance impact is significant. We believe

this highlights the importance of finding optimal placements and fences types wherever possible.

6. Related Work

We divide the related work into three groups: sequential consistency, inference, and other related works. The papers solve problems that are somewhat different than our problem, yet we will discuss how their techniques may be relevant to solve our problem.

Sequential consistency. Many authors have presented approaches to insert fences that enforce sequential consistency, including Lee et al. [**?**], Fang et al. [**?**], and Alglave et al. [**?**].

Inference. An alternative to specification of execution orders is *inference* of execution orders. The idea of inference is somewhat different from *type inference*, which can be understood as articulating program invariants. In the case of inference of execution orders, the challenge is to articulate assumptions needed to prove a correctness property.

Kuperstein, Vechev, and Yahav [**?**], presented promising work on inference; they infer execution orders from a program, a correctness property, and a memory model. Their approach first runs a whole-program state-space exploration algorithm that produces a logical formula, then solves the formula to get a set of execution orders, and finally uses those orders to insert fences. Their approach to enforce an order (i_1, i_2) is to insert a fence right after i_1 or right before i_2. The whole-program nature of their approach means that while the inserted fences are sound in the given context, they may be unsound in a different context. Still, their approach can give worthwhile feedback to an algorithm designer who tries to specify a set of orders that are sufficient to prove correctness. We note that our choice of correctness property (opacity) of transactional transactional memory algorithms is currently beyond the capabilities of the Kuperstein-Vechev-Yahav approach. What is needed here is a more powerful language for specifying correctness properties along with a suitable generalization of the approach. This can be an exciting direction for future work.

Kuperstein et al. [**?**], Meshman et al. [**?**], and Dan et al. [**?**] have presented approaches to a related inference problem, allowing degrees of infinite-state programs, but seemingly without execution orders as an intermediate step towards fences. Their approaches can likely be recast as inference of execution orders. Again, a direction for future work is to make their specification languages more powerful to enable specification of correctness of TM algorithms.

Liu et al. [**?**] presented an execution-based approach to inference, in which they run the program on a memory model and then use the traces to infer fences. This technique can likely be recast to infer execution orders instead.

Other. Execution orders can also be seen as restrictions on the possible executions of a program. In this way declared orders are similar to previous work on using annotations to restrict scheduling for correctness [**?**] and for testing [**?**]. Our work differs in its focus on the restriction of the possible executions due to instruction reordering (or the appearance thereof) as opposed to the restriction of the possible executions due to thread scheduling.

7. Conclusion

We have presented a declarative approach to specifying and enforcing the assumptions made by concurrent algorithms. For three TM algorithms, our tool inserted fences that are competitive with those inserted by the original authors. We believe execution orders are a natural abstraction that can help algorithm designers reason about their algorithms independently of memory models.

Acknowledgments

We thank the Michael Scott, Mike Spear, Luke D'Alessandro for answering questions about their ByteEager TLRW algorithm implementation. We thank Todd Millstein and Matt Brown for their feedback on drafts of this paper. We were partially supported by the NSF Expeditions in Computing Award CCF-0926127.

```
7  void* thr1(void * arg) {
8    flag1 = 1;
9    while (flag2 >= 1) {
10     if (turn != 0) {
11       flag1 = 0;
12       while (turn != 0) {};
13       flag1 = 1;
14     }
15   }
16   // begin: critical section
17   //x = 0;
18   //assert(x<=0);
19   // end: critical section
20   turn = 1;
21   flag1 = 0;
22 }
23
24 void* thr2(void * arg) {
25   flag2 = 1;
26   while (flag1 >= 1) {
27     if (turn != 1) {
28       flag2 = 0;
29       while (turn != 1) {};
30       flag2 = 1;
31     }
32   }
33   // begin: critical section
34   //x = 1;
35   //assert(x>=1);
36   // end: critical section
37   turn = 1;
38   flag2 = 0;
39 }
```

$$O_{\mathsf{Dekker}} = \{\, \mathsf{st}(\mathsf{flag1}) \rightarrow \mathsf{ld}(\mathsf{flag2}),\ \mathsf{st}(\mathsf{flag2}) \rightarrow \mathsf{ld}(\mathsf{flag1}) \,\}$$

Figure 17. Dekker's Mutex

Appendix A: Classic Concurrent Algorithms

In this appendix we include the source code for each of the "classic" algorithms that was used in our experiments. Each implementation comes from the Musketeer benchmarks and is reproduced here line-for-line. This is intended to serve as a reference for the line numbers detailed in the fence placement results tables in the main paper.

Dekker's Mutex

In Figure **??** we have the two procedures used to simulate process interaction in the Musketeer Dekker implementation. Note that `flag1`, `flag2` and `turn` are shared. Recall that the orders we have defined are store-load orders between lines 8 and 9, 13 and 9, 25 and 26, and 30 and 26. Note that the orders along back edges from the stores to `flag1` and `flag2` at the end of the `while` loops are required for the same reason as the order between the first stores to `flag1` and `flag2` and the loads just below them.

```
42 void park() {
43   if (_counter > 0) {
44     _counter = 0;
45     // mfence needed here
46     return;
47   }
48   if (mutex_trylock(&__unbuffered_mutex) !=
         0) return;
49   if (_counter > 0) { // no wait needed
50     _counter = 0;
51     mutex_unlock(__unbuffered_mutex);
52     return;
53   }
54   __unbuffered_did_park=1;
55   cond_wait(__unbuffered_cond,
       __unbuffered_mutex);
56   _counter = 0;
57   mutex_unlock(__unbuffered_mutex);
58 }
```

$$O_{\mathsf{Parker}} = \{\, \mathsf{st}(\mathsf{counter}) \rightarrow *\}$$

Figure 18. Parker

```
4  void* thr1(void * arg) {
5    flag1 = 1;
6    turn = 1;
7    do {} while (flag2==1 && turn==1);
8    // begin: critical section
9    // end: critical section
10   flag1 = 0;
11 }
12
13 void* thr2(void * arg) {
14   flag2 = 1;
15   turn = 0;
16   do {} while (flag1==1 && turn==0);
17   // begin: critical section
18   // end: critical section
19   flag2 = 0;
20 }
```

$$O_{\mathsf{Peterson}} = \{\, \mathsf{st}(\mathsf{flag1}) \rightarrow \mathsf{ld}(\mathsf{flag2}),\ \mathsf{st}(\mathsf{flag2}) \rightarrow \mathsf{ld}(\mathsf{flag1}) \,\}$$

Figure 19. Peterson's Mutex

Parker

In Figure **??** we have the procedure which contains the bug in the Parker implementation in the JVM. Note that `_counter` is shared. Recall that the order we have defined is a store-any order between lines 44 and 46.

Peterson's Mutex

In Figure **??** we have the two procedures used to simulate process interaction in the Musketeer Peterson implementation. Note that `flag1`, `flag2` and `turn` are shared. Recall that the orders we have defined are both store-load orders between lines 5 and 7 and also between lines 14 and 16.

```
5  void thr1() {
6   L0:
7    b1 = 1;
8    x = 1;
9    if (y != 0) {
10     b1 = 0;
11     goto L0;
12   }
13
14   y = 1;
15   if (x != 1) {
16     b1 = 0;
17
18     if (y != 1) {
19       goto L0;
20     }
21   }
22   // begin
23   // end
24   y = 0;
25   b1 = 0;
26 }
```

```
28 void thr2() {
29  L1:
30   b2 = 1;
31   x = 2;
32   if (y != 0) {
33     b2 = 0;
34     goto L1;
35   }
36
37   y = 2;
38   if (x != 2) {
39     b2 = 0;
40
41     if (y != 2) {
42       goto L1;
43     }
44   }
45   // begin
46   // end
47   y = 0;
48   b2 = 0;
49 }
```

$$O_{\mathsf{Lamport}} = \{\, \mathsf{st(x)} \to \mathsf{ld(y)}, \mathsf{st(y)} \to \mathsf{ld(x)} \,\}$$

Figure 20. Lamport's Mutex

Lamport's Mutex

Here we detail two example executions for Lamport's mutual exclusion algorithm. They illustrate the need for at least two orders in the implementation included in Musketeer's "classic" benchmarks. The relevant source code appears in Figure **??**. Note that `x` and `y` are shared. The first order is between the stores to `x` on line 8 and the loads of `y` on line 9. The second order is between the stores to `y` on line 14 and the loads of `x` on line 15.

For the example executions in Figures **??** and **??** we use $\mathsf{ld(y)} : 0$ to denote a 0 valued result loaded from the address represented by y, $\mathsf{st(y}, 1)$ to denote a store of the value 1 to the same address, and *enter* to denote the point at which a process enters the critical section. On the right margin we note where the loads correspond with `if` statements.

To see that the first order is necessary consider the execution in Figure **??**, where only the stores to x move past the guards that check if y is equal to 0.

Separately, if the stores to y are allowed to pass the `if` statements that check that value of x, the execution in Figure **??** is possible.

thr1	thr2 :	
$\mathsf{ld(y)} : 0$		`if` : $y \neq 0$
	$\mathsf{ld(y)} : 0$	`if` : $y \neq 0$
$\mathsf{st(x}, 1)$		
$\mathsf{st(y}, 1)$		
$\mathsf{ld}(x) : 1$		`if` : $x \neq 1$
enter		
	$\mathsf{st(x}, 2)$	
	$\mathsf{st(y}, 2)$	
	$\mathsf{ld}(x) : 2$	`if` : $x \neq 2$
	enter	

Figure 21. Lamport's Mutex, Bad Execution 1

thr1	thr2 :	
$\mathsf{st(x}, 1)$		
$\mathsf{ld(y)} : 0$		`if` : $y \neq 0$
$\mathsf{ld(x)} : 1$		`if` : $x \neq 1$
	$\mathsf{st(x}, 2)$	
	$\mathsf{ld(y)} : 0$	`if` : $y \neq 0$
	$\mathsf{st(y}, 2)$	
	$\mathsf{ld(x)} : 2$	`if` : $x \neq 2$
	enter	
$\mathsf{st(y}, 1)$		
enter		

Figure 22. Lamport's Mutex, Bad Execution 2

Appendix B: Correctness Proof

The lift **function.** We first define the helper function lift that later will enable us to state some properties succinctly. Let $G = (V, E, \ell)$ and let $K \subseteq E$, and suppose we have i_1, i_2 such that $p \in \mathsf{paths}(\mathsf{Refine}(G, K), i_1, i_2)$. We define $\mathsf{lift}(p)$ to be the corresponding path in G, that is, the path that for each pair of edges $(j_1, v_{j_1,j_2}), (v_{j_1,j_2}, j_2)$ in p instead has the edge $(j_1, j_2) \in K$. Notice that $\mathsf{lift}(p) \in \mathsf{paths}(G, i_1, i_2)$.

We prove the correctness of Insert in five steps. First we present four lemmas and then the main result (Theorem **??**). Each of the four lemmas states a key property of the Refine function. Before each lemma we will give an informal explanation that uses the following terminology. For a call $\mathsf{Refine}(G, A, O)$, we will refer to G as the *original* graph and we will refer to $\mathsf{Refine}(G, A, O)$ as the *refined* graph. Now let us move on to the four lemmas.

Intuitively, Lemma **??** says that reasoning about a path in the original graph carries over to the corresponding path in the refined graph.

Lemma 2. *Suppose* $G = (V, E, \ell)$ *and* $K \subseteq E$ *and* $(i_1, i_2) \in (V \times V)$ *and* $p \in \mathsf{paths}(\mathsf{Refine}(G, K), i_1, i_2)$. *If* $\mathsf{lift}(p), A \vdash i_1 \to i_2$, *then* $p, A \vdash i_1 \to i_2$.

Proof. We proceed by induction on the derivation of

$$\mathsf{lift}(p), A \vdash i_1 \to i_2.$$

We have two cases based on the last rule used in the derivation.

If the last rule used is the instantiation rule, then we have that i_1 can reach i_2 in $\mathsf{lift}(p)$, and we have $(L \mapsto R) \in A$, and $(L \mapsto R) \rhd (\ell(i_1), \ell(i_2))$. From $p \in \mathsf{paths}(\mathsf{Refine}(G, K), i_1, i_2)$, we have that i_1 can reach i_2 in p, so we can use the instantiation rule to derive $p, A \vdash i_1 \to i_2$.

If the last rule is the transitivity rule, then we can find j such that we can derive $\mathsf{lift}(p), A \vdash i_1 \to j$ and $\mathsf{lift}(p), A \vdash j \to i_2$. From the induction hypothesis, we have $p, A \vdash i_1 \to j$ and $p, A \vdash j \to i_2$. Now we use the transivity rule to derive $p, A \vdash i_1 \to i_2$. □

Intuitively, Lemma **??** says that reasoning about the original graph carries over to the refined graph.

Lemma 3. *Suppose $G = (V, E, \ell)$ and $K \subseteq E$. If $G, A \models O$, then $\mathsf{Refine}(G, K), A \models O$.*

Proof. Suppose $(i_1, i_2) \in O$, and let

$$p \in \mathsf{paths}(\mathsf{Refine}(G, K), i_1, i_2).$$

We have $\mathsf{lift}(p) \in \mathsf{paths}(G, i_1, i_2)$, so from $G, A \models O$, we have $\mathsf{lift}(p), A \vdash i_1 \to i_2$, so from Lemma **??**, we have $p, A \vdash i_1 \to i_2$. □

Intuitively, Lemma **??** says that certain paths in the refined graph must contain a fence.

Lemma 4. $\forall (i_1, i_2) \in O :$
$\forall p \in \mathsf{paths}(\mathsf{Refine}(G, \mathsf{Cut}(G, O)), i_1, i_2) : \exists j \in W_{\mathsf{Cut}(G,O)} :$
j is on p.

Proof. Let $K = \mathsf{Cut}(G, O)$ and suppose also that $p \in \mathsf{paths}(\mathsf{Refine}(G, K), i_1, i_2)$. Notice $\mathsf{lift}(p) \in \mathsf{paths}(G, i_1, i_2)$. From the displayed Formula (**??**), we have $\mathsf{paths}((V, E \setminus K, \ell), i_1, i_2) = \emptyset$, so $\mathsf{lift}(p)$ contains at least one edge that is also an element of K. Let (j_1, j_2) be such an edge. The call $\mathsf{Refine}(G, \mathsf{Cut}(G, O))$ returns a graph that, among other things, adds a node v_{j_1,j_2} and replaces (j_1, j_2) with two edges. Notice that the node v_{j_1,j_2} is on p. Additionally, from the definition of W_K we have $v_{j_1,j_2} \in W_K$. So, we can choose $j = v_{j_1,j_2}$. □

Intuitively, Lemma **??** says that, with an appropriate assumption about the fence fany, the refined graph contains sufficient fences to enforce O.

Lemma 5. *If $\{(* \mapsto \mathsf{fany}), (\mathsf{fany} \mapsto *)\} \subseteq A$, then $\mathsf{Refine}(G, \mathsf{Cut}(G, O)), A \models O$.*

Proof. Suppose $(i_1, i_2) \in O$ and let furthermore $p \in \mathsf{paths}(\mathsf{Refine}(G, \mathsf{Cut}(G, O)), i_1, i_2)$. From Lemma **??**, we have that we can find $j \in W_{\mathsf{Cut}(G,O)}$ such that j is on p. In particular, i_1 can reach j in p, and j can reach i_2 in p, and $\ell(j) = \mathsf{fany}$. From $\ell(j) = \mathsf{fany}$ we have $(* \mapsto \mathsf{fany}) \rhd (\ell(i_1), \ell(j))$ and $(\mathsf{fany} \mapsto *) \rhd (\ell(j), \ell(i_2))$. From $\{(* \mapsto \mathsf{fany}), (\mathsf{fany} \mapsto *)\} \subseteq A$, we have that we can use the instantiation rule to get $p, A \vdash i_1 \to j$ and $p, A \vdash j \to i_2$. Now we use transitivity to conclude $p, A \vdash i_1 \to i_2$. □

Now we are ready to prove the main result. Like Lemma **??**, also Theorem **??** says that with an appropriate assumption about the fence fany, the refined graph contains sufficient fences to enforce O. The difference is that Lemma **??** is only about Cut and Refine, while Theorem **??** is about the entire definition of Insert, which also uses Elim.

Theorem 1. *If $\{(* \mapsto \mathsf{fany}), (\mathsf{fany} \mapsto *)\} \subseteq A$, then $\mathsf{Insert}(G, A, O), A \models O$.*

Proof. Suppose $\{(* \mapsto \mathsf{fany}), (\mathsf{fany} \mapsto *)\} \subseteq A$ and let $G = (V, E, \ell)$. From the displayed Formula (**??**), we have $G, A \models \mathsf{Elim}(G, A, O)$, and from the displayed Formula (**??**), we have $\mathsf{Cut}(G, O \setminus \mathsf{Elim}(G, A, O)) \subseteq E$. When we apply Lemma **??** to those two properties, we get

$$\mathsf{Insert}(G, A, O), A \models \mathsf{Elim}(G, A, O). \tag{3}$$

From Lemma **??**, we have:

$$\mathsf{Insert}(G, A, O), A \models O \setminus \mathsf{Elim}(G, A, O). \tag{4}$$

From the displayed Property (**??**), we have $\mathsf{Elim}(G, A, O) \subseteq O$, so:

$$\mathsf{Elim}(G, A, O) \cup (O \setminus \mathsf{Elim}(G, A, O)) = O. \tag{5}$$

From the displayed Formulas (**??**)–(**??**), we can conclude that $\mathsf{Insert}(G, A, O), A \models O$. □