Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion

Jade A., Daniel K., Vincent N., Daniel P.

Presented by
Akshay Gopalakrishnan

October 14, 2023

Introduction/Motivation

- Memory fences enable hardware to prevent weak behaviors due to hardware optimizations.
- Therefore, an automated way to ensure fences placed where required is desired.
- Fences are expensive, and therefore, this automated approach must be efficient, using minimal fences wherever possible.
- Previous approaches to this technique are not scalable to large code-bases due to state space explosions.

Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion

Introduction/Motivation

Introduction/Motivation

- Memory fences enable hardware to prevent weak behaviors due to hardware optimizations.
- Therefore, an automated way to ensure fences placed where required is desired.
- Fences are expensive, and therefore, this automated approach must be efficient, using minimal fences wherever possible.
- Previous approaches to this technique are not scalable to large code-bases due to state space explosions.

The authors note that their approach trades precision to scale the automation of fence insertion. It is unclear to me precision in what sense, do they mean that they may place too many fences than required? If that is the case then it should be fine. Otherwise by precision if they mean they do not restore consistency desired, then the approach is dangerous. The authors hopefully clarify what they mean by this.

Main Idea

- Use axiomatic specification of memory model to avoid state space explosion.
- Utilize previous work's concept of critical cycles in execution graphs.
- Statically determine fences to be inserted to guarantee only Sequentially Consistent (SC) behaviors.

Core component 1

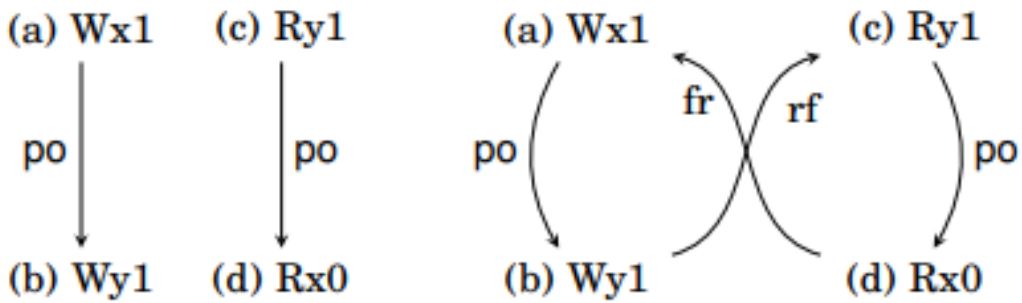
- Executions as graphs using *po*, *rf*, *fr* edges.
- Axiomatic specification of a memory model as acyclic constraints on execution graphs.

mp

T_0	T_1
(a) x ← 1	(c) r1 ← y
(b) y ← 1	(d) r2 ← x

Final state? r1=1 ∧ r2=0

Fig. 4. Message Passing (**mp**).



(a) event structure (b) candidate execution

Fig. 5. Event structure and candidate execution

Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion

Core component 1

Core component 1

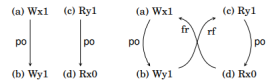
- Executions as graphs using po, rf, fr edges.
- Axiomatic specification of a memory model as acyclic constraints on execution graphs.

mp

T_0	T_1
(a) x ← 1	(c) r1 ← y
(b) y ← 1	(d) r2 ← x

Final state? r1=1 ∧ r2=0

Fig. 4. Message Passing (**mp**).



(a) event structure (b) candidate execution

Fig. 5. Event structure and candidate execution

Traditionally, axiomatic specification has been proven fruitful for scalable verification of concurrent programs. The execution graphs represent a possible execution of the original program. These graphs are typically annotated with information that we do have in a program execution.

Core component 2

Critical cycles in execution graphs:

- Minimal cycles that are at least present in an execution graph.
- Transitive properties of partial orders in graphs ().
- Every model considered at least respects coherence (SC-per location).

Minimal cycle detection:

- (MC1) Per thread, at most 2 memory accesses, connected by a direct edge in the cycle.
- (MC2) Overall, at most 3 accesses to same memory in the cycle.

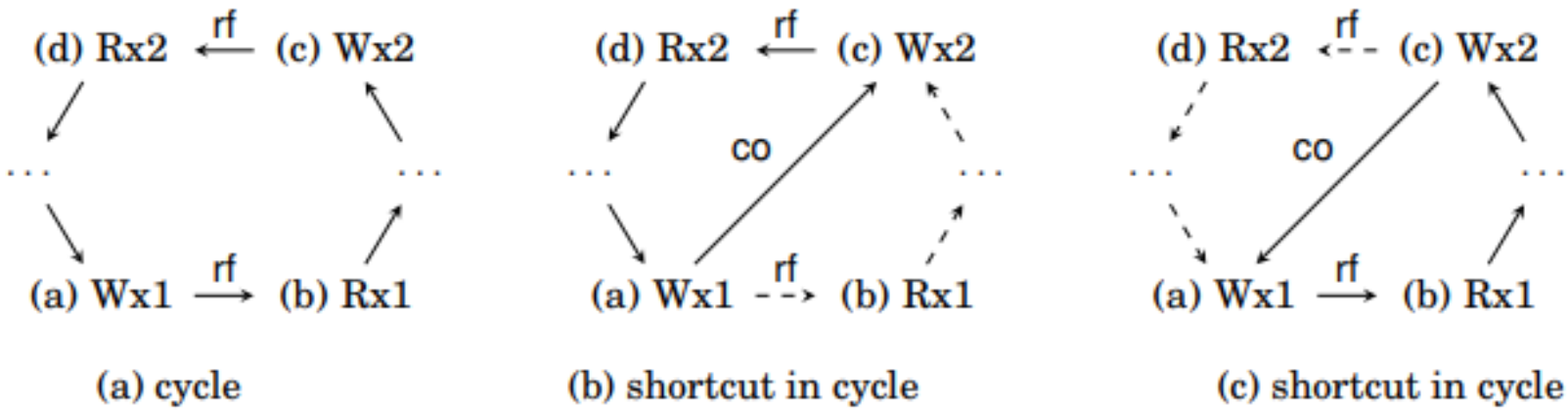
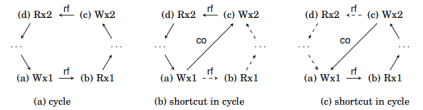


Fig. 6. A cycle in a candidate execution, and two possible shortcuts one can take to form a smaller cycle

Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion

Minimal cycle detection:

- (MC1) Per thread, at most 2 memory accesses, connected by a direct edge in the cycle.
- (MC2) Overall, at most 3 accesses to same memory in the cycle.



(a) cycle (b) shortcut in cycle (c) shortcut in cycle

Fig. 6. A cycle in a candidate execution, and two possible shortcuts one can take to form a smaller cycle

Minimal cycles are identified using primarily the transitive property of $po \cup rf$. There is an additional order on memory writes which is not discussed so far in the paper in detail (viz. coherence order co), which is total over accesses on same memory. These two orders help us identify minimal cycles, which basically boil down to MC1 and MC2. Note that minimal cycles is constrained by length or number of nodes in the cycle.

A delay is *po* or *rf* edge that can cause a weak behavior. A behavior is not SC but in weaker model A if and only if

- (D1) There is at least one cycle with a delay edge.
- (D2) All cycles have a delay.

Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion

A delay is *po* or *rf* edge that can cause a weak behavior. A behavior is not SC but in weaker model A if and only if

- (D1) There is at least one cycle with a delay edge.
- (D2) All cycles have a delay.

A delay is a possible edge that can contribute to a weak behavior. In truth, this edge can also not be a delay, pertaining to an execution having a cycle or not. Its use is solely in the executions that have cycles. This is reminiscent of what we do as crucial reads in an execution.

(D1) is straightforward, as if no delay, the cycle disallows behaviors in both SC and A. (D2) again is based on the same reasoning as D1, as even if one cycle has no delay it is not allowed both in SC and A.

Identifying the delay edges is parametric to the weak memory model A under consideration.

Critical cycles are minimal cycles which are not allowed in SC but in a weaker model A. A cycle is critical for memory model A if and only if

- (CS1) At least one delay in the cycle.
- (CS2) Per thread, at most two accesses, to different locations.
- (CS3) There are at most 3 accesses to same memory location.

nb: Assume all models adhere to coherence (SC-per-location)

Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion

Critical cycles are minimal cycles which are not allowed in SC but in a weaker model A. A cycle is critical for memory model A if and only if

- (CS1) At least one delay in the cycle.
- (CS2) Per thread, at most two accesses, to different locations.
- (CS3) There are at most 3 accesses to same memory location.

nb: Assume all models adhere to coherence (SC-per-location)

CS1 is directly from D1. CS2, if exists with same location, will violate coherence (go casewise using MC1, MC2, D2). CS3 is a little unsure to me too, does not make sense as of now.

Core Component 2

Static detection of critical cycles

- Enumerating all executions and testing for cycles will not scale.
- Goal is to identify places to insert fences to restore SC.
- Instead, map a program to an overapproximation of the program's behaviors, called Abstract Event Graph (aeg).

From C programs to Goto Equivalents

Map a C program to an equivalent C program purely with goto statements.

- Replace conditional branching (if statements) with goto equivalents.
- Replace loop structure (while/for/do statements) with goto equivalents.

```
void thread_1(int input)
{
  int r1;
  x = input;
  if (rand()%2)
    y = 1;
  else
    r1 = z;
  x = 1;
}
```

```
void thread_2()
{
  int r2, r3, r4;
  r2 = y;
  r3 = z;
  r4 = x;
}
```

```
thread_1
    int r1;
    x = input;
    _Bool tmp;
    tmp = rand();
    [!tmp%2] goto 1;
    y = 1;
    goto 2;
1:  r1 = z;
2:  x = 1;
    end_function
```

```
thread_2
    int r2, r3, r4;
    r2 = y;
    r3 = z;
    r4 = x;
    end_function
```

Fig. 9. A C program (left) and its goto-program (right).

Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion

From C programs to Goto Equivalents

From C programs to Goto Equivalents

Map a C program to an equivalent C program purely with goto statements.

- Replace conditional branching (if statements) with goto equivalents.
- Replace loop structure (while/for/do statements) with goto equivalents.

```
void thread_1(int input)
{
  int r1;
  x = input;
  if (rand()%2)
    y = 1;
  else
    r1 = z;
  x = 1;
}
```

```
void thread_2()
{
  int r2, r3, r4;
  r2 = y;
  r3 = z;
  r4 = x;
}
```

```
thread_1
  int r1;
  x = input;
  _Bool tmp;
  tmp = rand();
  [!tmp%2] goto 1;
  y = 1;
  goto 2;
1: r1 = z;
2: x = 1;
  end_function
```

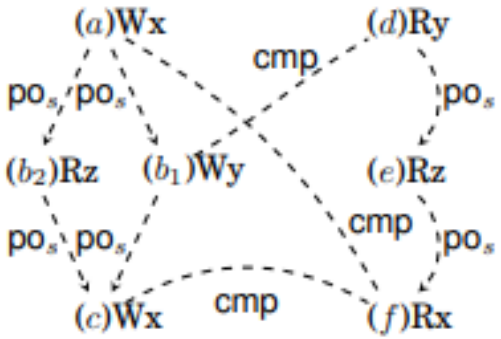
```
thread_2
  int r2, r3, r4;
  r2 = y;
  r3 = z;
  r4 = x;
  end_function
```

Fig. 9. A C program (left) and its goto-program (right).

The choice for converting a program to its goto equivalent is not explained. My guess is that this is rather a choice of convenience in terms of designing the algorithm to derive abstract event graphs. Apart from the feasibility of the algorithm, the only other explaination I have is they mention CProver has the same representation of C programs. Thus, it could simply be reusability of this mapping already done by CProver, or that they would want to use this tool to extend existing work towards model checking (verification).

C-goto Program to aeg

- Extract only the shared memory events, removing local computations.
- Retain syntactic order between memory events (po_s).
- Derive the binary relation relating two events that could cause a data race (*cmp*)



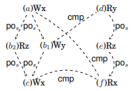
(a) aeg of Figure 9

Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion

C-goto Program to aeg

C-goto Program to aeg

- Extract only the shared memory events, removing local computations.
- Retain syntactic order between memory events (po_s).
- Derive the binary relation relating two events that could cause a data race (cmp)

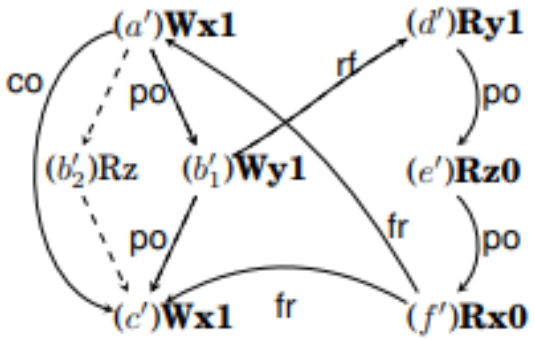


(a) aeg of Figure 9

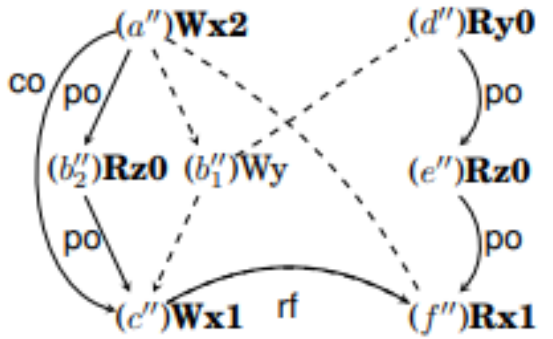
Notice that the read events or write events both do not have concrete values. This is reminiscent of our idea on pre-trace model for memory consistency. The values are made concrete only while adding *rf* and other relations. Although what is not clear is the possible write values that a write can have. However, note that the aeg does represent all possible candidate executions of the original program. This was also proven by them.

aeg to Concrete (candidate) executions

- Give writes and reads concrete values.
- Read concrete values need a write to exist with the same concrete value.
- Add *rf* edges, and concrete *po* edges, taking one control flow path for each branching.
- Lastly, add *co* edges.



(b) ex. with critical cycle



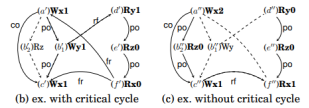
(c) ex. without critical cycle

Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion

└aeg to Concrete (candidate) executions

aeg to Concrete (candidate) executions

- Give writes and reads concrete values.
- Read concrete values need a write to exist with the same concrete value.
- Add *rf* edges, and concrete *po* edges, taking one control flow path for each branching.
- Lastly, add *co* edges.



(b) ex. with critical cycle (c) ex. without critical cycle

This mapping to candidate executions is very much the same way we map pre-traces to candidate executions. The only concern where the difference might be evident is when the branch choice is made, does it depend on the concrete read values or is it also random. If it is the former, then there is a clear difference between our model and theirs. If it is the latter, then their and our representation may just be the same (perhaps discuss with Clark about this).

Fences and Dependencies

- Fences are encoded in aeg as special abstract events (nodes) in the graph (eg: mfence in x86).
- Dependencies are not part of the aeg, but are recorded.

Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion

Fences and Dependencies

Fences and Dependencies

- Fences are encoded in aeg as special abstract events (nodes) in the graph (eg: mfence in x86).
- Dependencies are not part of the aeg, but are recorded.

The choice of not encoding dependencies into the aeg is not clear to me. Perhaps the way to do it would be an edge in the graph, but the edge itself could then complicate our definition of what is a critical cycle and so on. Maybe keeping it separate is a good choice, the aeg is much simpler and the dependency calculation can be used separately to determinte fence insertions.

Loops in aeg

- aeg abstracts away the loop body entirely.
- The read/write values are not concrete, thus the loop's behavior is more generalized.
- To find critical cycles, a loop body must be duplicated at least once(check CS2), which helps us have edges relating events between any two iterations of the loop.

Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion

Loops in aeg

Loops in aeg

- aeg abstracts away the loop body entirely.
- The read/write values are not concrete, thus the loop's behavior is more generalized.
- To find critical cycles, a loop body must be duplicated at least once(check CS2), which helps us have edges relating events between any two iterations of the loop.

This idea is actually very useful and it helps us guarantee that we only need to consider loops as two iterations and nothing more. All of this is on the premise that events do not have concrete values. I am guessing for infinite loops, this may also be the case. But still, have to think on this a bit more.

Critical cycle detection

- Cycle detection done with previous work (Tarjan's Algorithm).
- Algorithm finds all elementary circuits in a directed graph.
- Complexity is $O((V + E) * (C + 1)$, where for vertices (V), edges (E) and circuits (C) in graph.
- Critical cycle detection is done using previous rules.

Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion

Critical cycle detection

Critical cycle detection

- Cycle detection done with previous work (Tarjan's Algorithm).
- Algorithm finds all elementary circuits in a directed graph.
- Complexity is $O((V + E) * (C + 1)$, where for vertices (V), edges (E) and circuits (C) in graph.
- Critical cycle detection is done using previous rules.

Honestly, this algorithm can be parallelized too. Since their main proponent is to speed up fence insertion to make it scalable for bigger programs, a natural extension would be to change the cycle detection algorithm itself.

Identifying fence insertions

- Fence placement is thread-local, between events that partake in a critical cycle and are in the same thread.
- Optimal placement depends on how many potential critical cycles can exist in one aeg, with overlapping cycle paths in the same thread.
- Placing fences that eliminate most cycles would be the way to go.
- However, doing this for multiple threads and assessing is non-trivial.
- To add, there are multiple fence instructions, given different target architectures, each being stronger or weaker than the other.

Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion

Identifying fence insertions

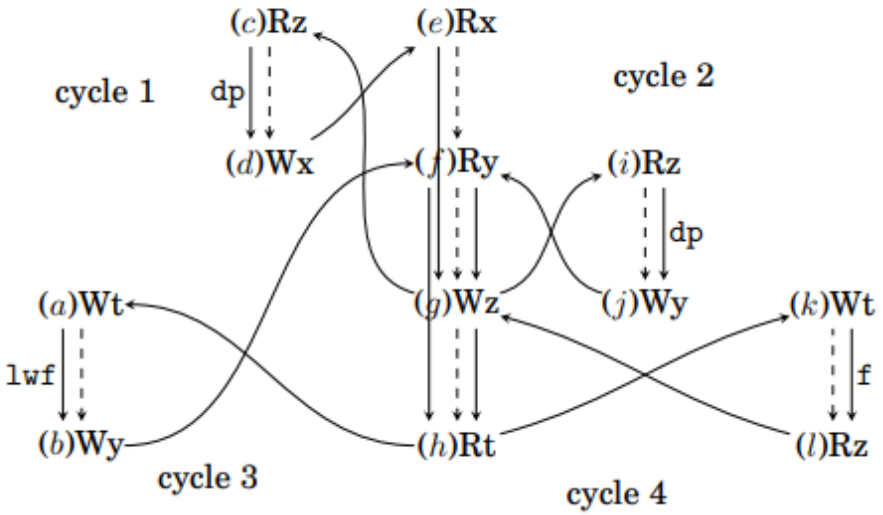
Identifying fence insertions

- Fence placement is thread-local, between events that partake in a critical cycle and are in the same thread.
- Optimal placement depends on how many potential critical cycles can exist in one aeg, with overlapping cycle paths in the same thread.
- Placing fences that eliminate most cycles would be the way to go.
- However, doing this for multiple threads and assessing is non-trivial.
- To add, there are multiple fence instructions, given different target architectures, each being stronger or weaker than the other.

The paper delves into different types of fences as well handling cumulativity. I think cumulativity is actually pretty non-trivial to address, due to dependencies that must exist accross code fragments. I am not sure why they did not just use the dependency calculations done earlier from the program, and use that instead of inserting your own fences again. This is okay as they anyways disable compiler optimizations. So it should ideally be fine.

Optimizing fence insertions (ILP)

- Frame the minimization of fences as an ILP.
- Associate each fence with a cost and calculate the minimization



$$
\begin{aligned}
\textbf{min}\quad & \mathtt{dp}_{(e,g)} + \mathtt{dp}_{(f,h)} + \mathtt{dp}_{(f,g)} + 3 \cdot (\mathtt{f}_{(e,f)} + \mathtt{f}_{(f,g)} + \mathtt{f}_{(g,h)}) \\
& +2 \cdot (\mathtt{lwf}_{(e,f)} + \mathtt{lwf}_{(f,g)} + \mathtt{lwf}_{(g,h)}) \\
\textbf{s.t.}\quad & \text{cycle 1, delay } (e,g)\text{: } \mathtt{dp}_{(e,g)} + \mathtt{f}_{(e,f)} + \mathtt{f}_{(f,g)} + \mathtt{lwf}_{(e,f)} + \mathtt{lwf}_{(f,g)} \geq 1 \\
& \text{cycle 2, delay } (f,g)\text{: } \mathtt{dp}_{(f,g)} + \mathtt{f}_{(f,g)} + \mathtt{lwf}_{(f,g)} \geq 1 \\
& \text{cycle 3, delay } (f,h)\text{: } \mathtt{dp}_{(f,h)} + \mathtt{f}_{(f,g)} + \mathtt{f}_{(g,h)} + \mathtt{lwf}_{(f,g)} + \mathtt{lwf}_{(g,h)} \geq 1 \\
& \text{cycle 4, delay } (g,h)\text{: } \mathtt{f}_{(g,h)} \geq 1
\end{aligned}
$$

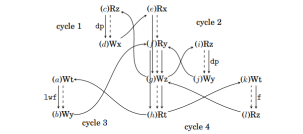
Fig. 16. Example of resolution with between.

Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion

└Optimizing fence insertions (ILP)

Optimizing fence insertions (ILP)

- Frame the minimization of fences as an ILP.
- Associate each fence with a cost and calculate the minimization



$$
\begin{aligned}
\textbf{min}\quad & \mathrm{dp}_{(e,g)} + \mathrm{dp}_{(f,h)} + \mathrm{dp}_{(f,g)} + 3 \cdot (\mathrm{f}_{(e,f)} + \mathrm{f}_{(f,g)} + \mathrm{f}_{(g,h)}) \\
& +2 \cdot (1\mathrm{wf}_{(e,f)} + 1\mathrm{wf}_{(f,g)} + 1\mathrm{wf}_{(g,h)}) \\
\textbf{s.t.}\quad & \textbf{cycle 1, delay } (e,g)\textbf{:}\ \mathrm{dp}_{(e,g)} + \mathrm{f}_{(e,f)} + \mathrm{f}_{(f,g)} + 1\mathrm{wf}_{(e,f)} + 1\mathrm{wf}_{(f,g)} \geq 1 \\
& \textbf{cycle 2, delay } (f,g)\textbf{:}\ \mathrm{dp}_{(f,g)} + \mathrm{f}_{(f,g)} + 1\mathrm{wf}_{(f,g)} \geq 1 \\
& \textbf{cycle 3, delay } (f,h)\textbf{:}\ \mathrm{dp}_{(f,h)} + \mathrm{f}_{(f,g)} + \mathrm{f}_{(g,h)} + 1\mathrm{wf}_{(f,g)} + 1\mathrm{wf}_{(g,h)} \geq 1 \\
& \textbf{cycle 4, delay } (g,h)\textbf{:}\ \mathrm{f}_{(g,h)} \geq 1
\end{aligned}
$$

Fig. 16. Example of resolution with between.

This is a very smart solution to address the minimization problem. However, it turns out to be suboptimal which is observed in the testing phase. I am not sure why it is suboptimal, and perhaps a solution to it might lead to a good publication (talk with Clark, reel in someone else I guess?).

Actual fence/dependency placement

- Fences are inserted into C code as inline assembly (why I wonder).
- Dependencies require some more work to just introduce a dependency (again via inline assembly) in the code.

```
1 r1 = x;
2 r2 = r1+2;
3 r3 = y;
4
```

```
1 r1 = x;
2 asm ("mfence");
3 r2 = r1+2;
4 r3 = y;
```

```
1 r1 = x;
2 r2 = r1+2;
3 asm ("mfence");
4 r3 = y;
```

Fig. 20. Choices for placing a fence.

Testing, comparing with other techniques

- Testing how much time it takes to identify fence placements.
- Testing how many fences are to be added.

	Dek		Pet		Lam		Szy		Par	
LoC	50		37		72		54		96	
dfence	–	–	–	–	–	–	–	–	–	–
memorax	0.4	2	1.4	2	79.1	4	1.3	3	1.1	0
musketeer	**0.0**	**5**	**0.0**	**3**	**0.0**	**8**	**0.0**	**8**	**0.0**	**3**
offence	0.0	2	0.0	2	0.0	8	0.0	8	–	–
pensieve	0.0	16	0.0	6	0.0	24	0.0	22	0.0	7
remmex	0.5	2	0.5	2	2.0	4	1.8	5	–	–
trencher	1.6	2	1.3	2	1.7	4	1.1	8	0.5	1
persist	0.8	2	0.7	2	2.5	8	2.1	8	0.9	1

Fig. 23. All tools on the CLASSIC series for TSO.

- Testing runtime overheads due to placement of fences (M is the proposed implementation).

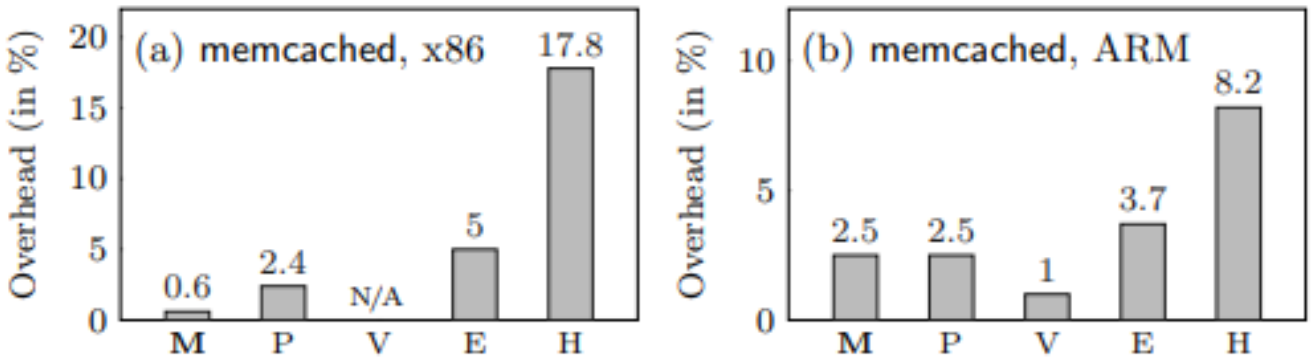


Fig. 26. Runtime overheads due to fences inserted in memcached for each strategy.

Don't Sit on the Fence: A Static Analysis Approach to Automatic Fence Insertion

- Testing runtime overheads due to placement of fences (M is the proposed implementation).

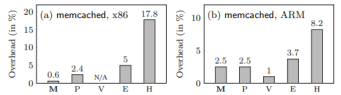


Fig. 26. Runtime overheads due to fences inserted in memcached for each strategy.

It is interesting to note that while fences suggested to add is more in some cases for musketeer, it still shows relatively less overhead. Perhaps the cases for which additional fences were suggested, the case is not covered in the bigger examples of Debian packages.

Conclusion

- Fence insertion techniques not scalable.
- Propose a new technique which statically determines fence placements.
- Relies on overapproximation of program behavior, using abstract event graphs.
- Framed minimal fence insertion as an ILP problem.