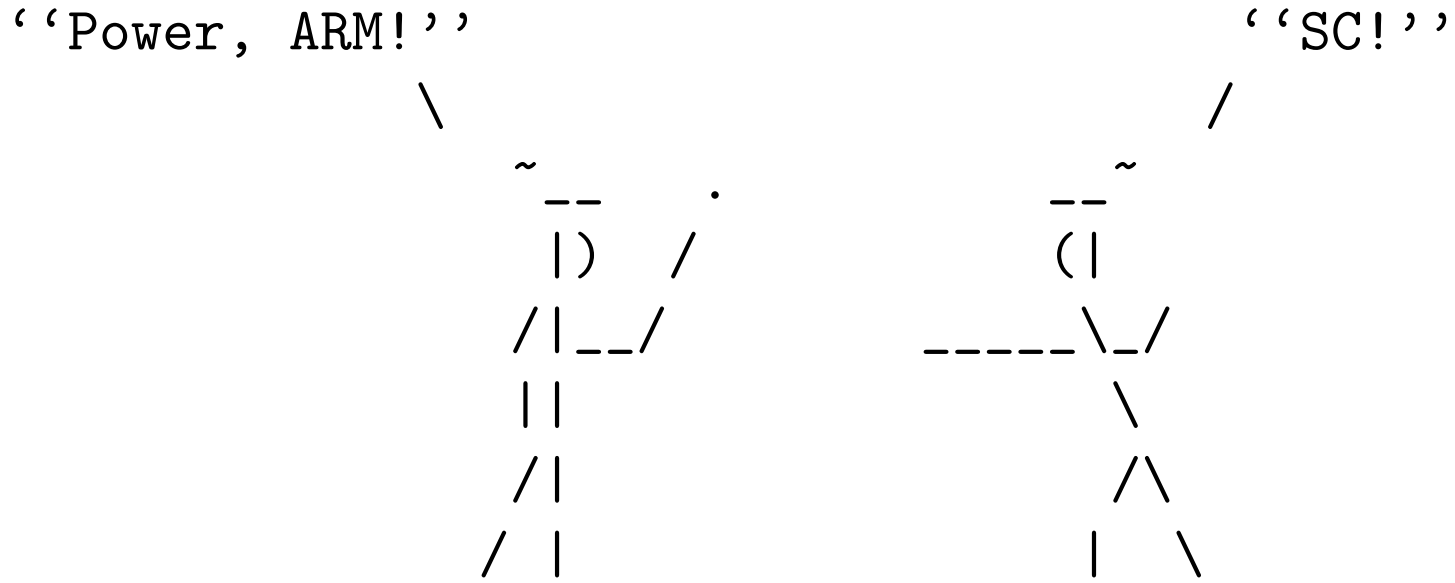
Don't sit on the fence: A static analysis approach to automatic fence insertion



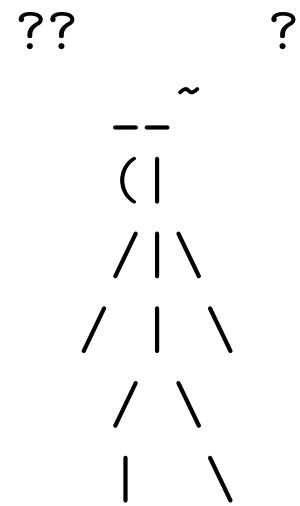
Jade Alglave, Daniel Kroening, **Vincent Nimal**, Daniel Poetzl

University of Oxford

Weak Memory Consistency

```
1 volatile int data = 0, flag = 0;

3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }
```

```
8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert( flag_set ⇒ data_upd );
12 }
```



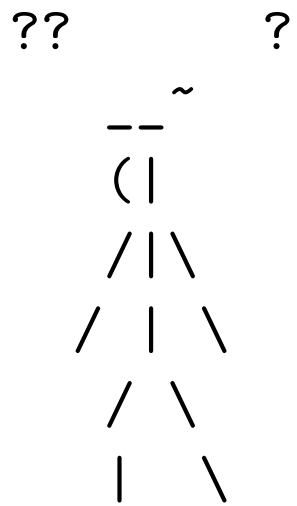
Weak Memory Consistency

```
1 volatile int data = 0, flag = 0;

3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }
```

```
8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert( flag_set ⇒ data_upd );
12 }
```

Sequential Consistency: (SC; interleavings semantics)

data=1;



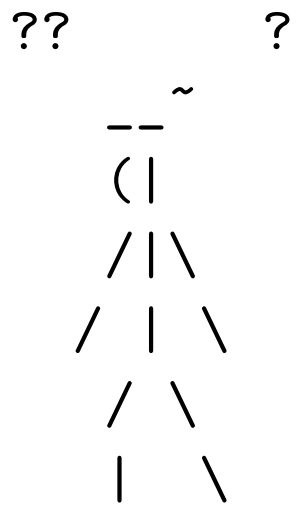
Weak Memory Consistency

```
1 volatile int data = 0, flag = 0;

3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }
```

```
8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert( flag_set ⇒ data_upd );
12 }
```

Sequential Consistency: (SC; interleavings semantics)

data=1; flag=1;



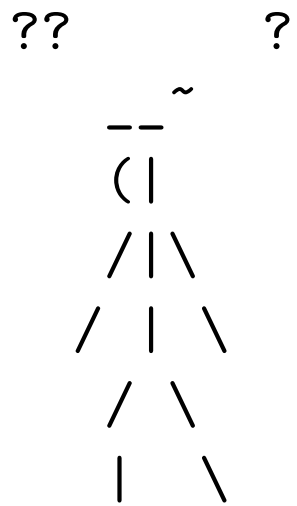
Weak Memory Consistency

```
1 volatile int data = 0, flag = 0;

3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }
```

```
8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert( flag_set ⇒ data_upd );
12 }
```

Sequential Consistency: (SC; interleavings semantics)

data=1; flag=1; flag_set=1;



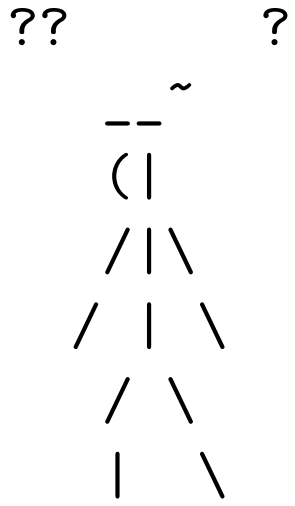
Weak Memory Consistency

```
1 volatile int data = 0, flag = 0;

3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }
```

```
8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert( flag_set ⇒ data_upd );
12 }
```

Sequential Consistency: (SC; interleavings semantics)

data=1; flag=1; flag_set=1; data_upd=1;



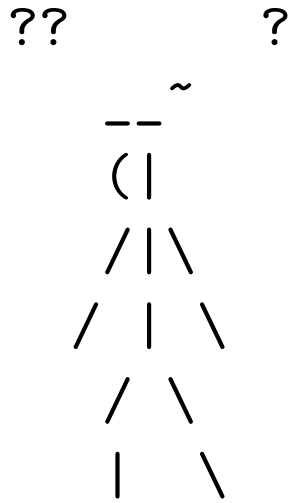
Weak Memory Consistency

```
1 volatile int data = 0, flag = 0;

3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }
```

```
8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert( flag_set ⇒ data_upd );
12 }
```

Sequential Consistency: (SC; interleavings semantics)

data=1; flag=1; flag_set=1; data_upd=1;



Weak Memory Consistency

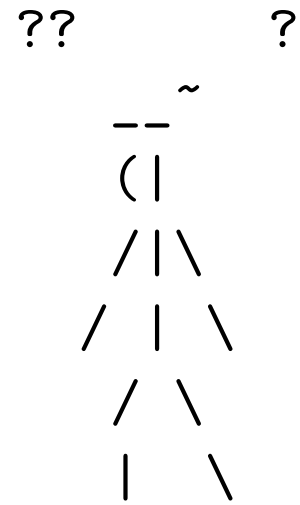
```
1 volatile int data = 0, flag = 0;

3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }
```

```
8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert( flag_set ⇒ data_upd );
12 }
```

Sequential Consistency: (SC; interleavings semantics)

data=1; flag=1; flag_set=1; data_upd=1;

data=1; flag_set=0; flag=1; data_upd=1;



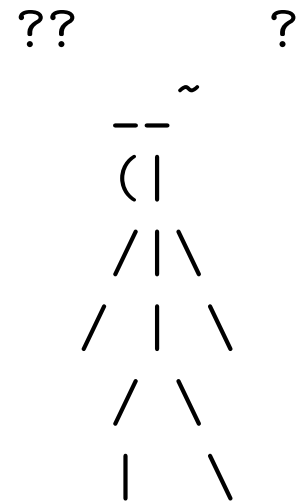
Weak Memory Consistency

```
1 volatile int data = 0, flag = 0;

3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }
```

```
8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert( flag_set ⇒ data_upd );
12 }
```

Sequential Consistency: (SC; interleavings semantics)

data=1; flag=1; flag_set=1; data_upd=1;
data=1; flag_set=0; flag=1; data_upd=1;
data=1; flag_set=0; data_upd=1; flag=1;



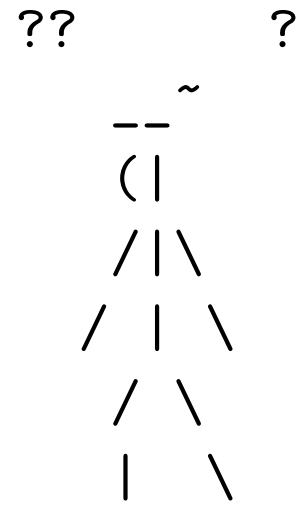
Weak Memory Consistency

```
1 volatile int data = 0, flag = 0;

3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }
```

```
8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert( flag_set ⇒ data_upd );
12 }
```

Sequential Consistency: (SC; interleavings semantics)

data=1; flag=1; flag_set=1; data_upd=1;
data=1; flag_set=0; flag=1; data_upd=1;
data=1; flag_set=0; data_upd=1; flag=1;
flag_set=0; data_upd=0; data=1; flag=1;



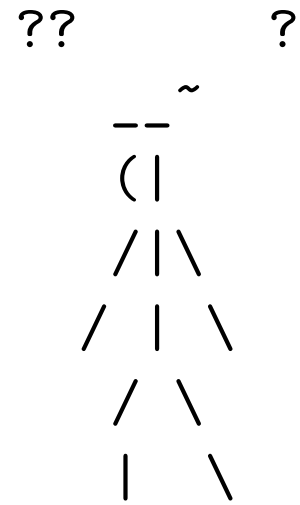
Weak Memory Consistency

```
1 volatile int data = 0, flag = 0;

3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }
```

```
8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert( flag_set ⇒ data_upd );
12 }
```

Sequential Consistency: (SC; interleavings semantics)

data=1; flag=1; flag_set=1; data_upd=1;
data=1; flag_set=0; flag=1; data_upd=1;
data=1; flag_set=0; data_upd=1; flag=1;
flag_set=0; data_upd=0; data=1; flag=1;
flag_set=0; data=1; data_upd=1; flag=1;



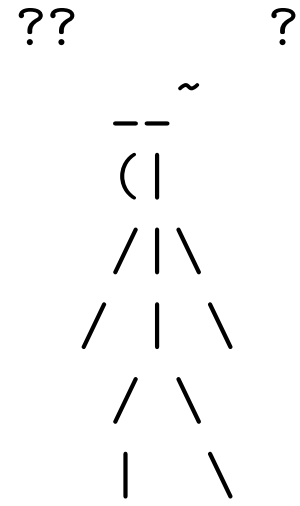
Weak Memory Consistency

```
1 volatile int data = 0, flag = 0;

3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }
```

```
8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert( flag_set ⇒ data_upd );
12 }
```

Sequential Consistency: (SC; interleavings semantics)

data=1; flag=1; flag_set=1; data_upd=1;
data=1; flag_set=0; flag=1; data_upd=1;
data=1; flag_set=0; data_upd=1; flag=1;
flag_set=0; data_upd=0; data=1; flag=1;
flag_set=0; data=1; data_upd=1; flag=1;
flag_set=0; data=1; flag=1; data_upd=1;



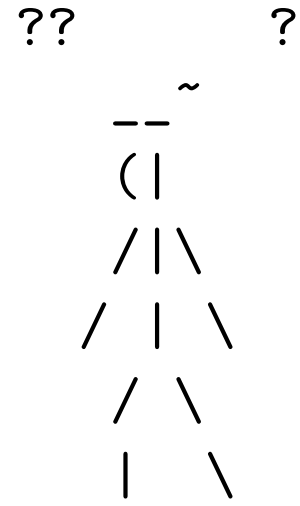
Weak Memory Consistency

```
1 volatile int data = 0, flag = 0;

3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }
```

```
8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert( flag_set ⇒ data_upd );
12 }
```

Sequential Consistency: (SC; interleavings semantics)

data=1; flag=1; flag_set=1; data_upd=1;
data=1; flag_set=0; flag=1; data_upd=1;
data=1; flag_set=0; data_upd=1; flag=1;
flag_set=0; data_upd=0; data=1; flag=1;
flag_set=0; data=1; data_upd=1; flag=1;
flag_set=0; data=1; flag=1; data_upd=1;



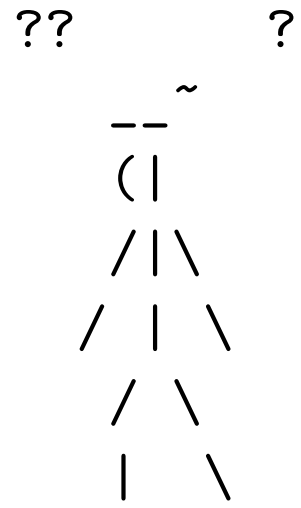
Weak Memory Consistency

```
1 volatile int data = 0, flag = 0;

3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }
```

```
8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert( flag_set ⇒ data_upd );
12 }
```

Sequential Consistency: (SC; interleavings semantics)

data=1; flag=1; flag_set=1; data_upd=1;
data=1; flag_set=0; flag=1; data_upd=1;
data=1; flag_set=0; data_upd=1; flag=1;
flag_set=0; data_upd=0; data=1; flag=1;
flag_set=0; data=1; data_upd=1; flag=1;
flag_set=0; data=1; flag=1; data_upd=1;



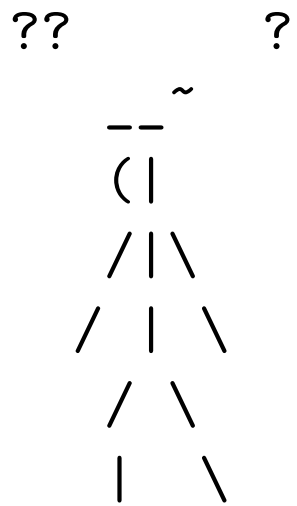
Note that flag_set⇒data_upd is always verified

Weak Memory Consistency

```
1 volatile int data = 0, flag = 0;

3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }
```

```
8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert( flag_set ⇒ data_upd );
12 }
```

On Power or ARM, assertion failed!



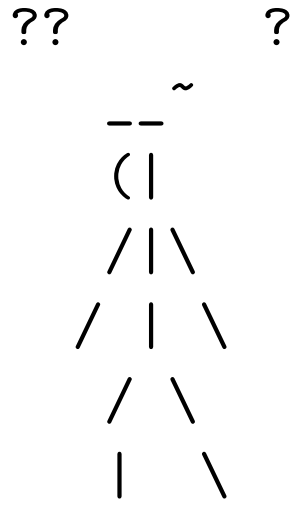
Weak Memory Consistency

```
1 volatile int data = 0, flag = 0;

3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }
```

```
8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert( flag_set ⇒ data_upd );
12 }
```

On Power or ARM, assertion failed!

Modern multicore architectures (e.g. x86, ARM)
weaker than Sequential Consistency
i.e., can exhibit more behaviours than SC



Weak Memory Consistency

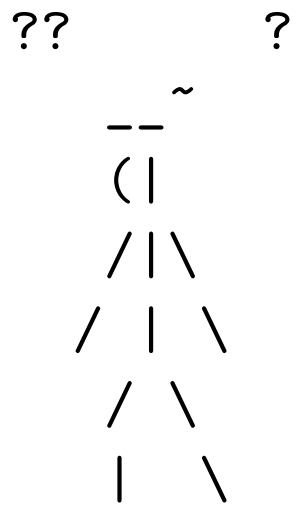
```
1 volatile int data = 0, flag = 0;

3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }
```

```
8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert( flag_set ⇒ data_upd );
12 }
```

On Power or ARM, assertion failed!

Modern multicore architectures (e.g. x86, ARM) **weaker** than Sequential Consistency
i.e., can exhibit more behaviours than SC

Writes (amongst other things) can be **reordered** by the processor for performance reasons

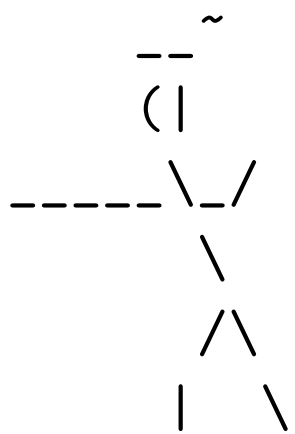


Restoring Sequential Consistency (our objective)

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   __asm__ ("lwsync");
6   flag = 1;
7 }
```

```
9 void* thread_1 (void* arg) {
10  int flag_set = flag;
11  __asm__ ("lwsync");
12  int data_upd = data;
13  assert( flag_set ⇒ data_upd );
14 }
```

Concurrent programming with **interleavings** in mind is simpler than with processor specifics



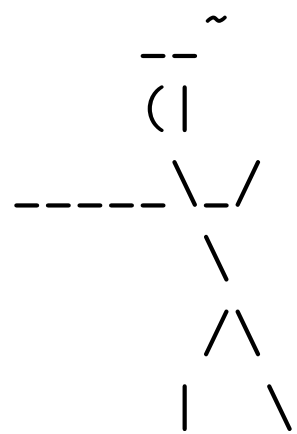
Restoring Sequential Consistency (our objective)

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   __asm__ ("lwsync");
6   flag = 1;
7 }
```

```
9 void* thread_1 (void* arg) {
10  int flag_set = flag;
11  __asm__ ("lwsync");
12  int data_upd = data;
13  assert( flag_set ⇒ data_upd );
14 }
```

Concurrent programming with **interleavings** in mind is simpler than with processor specifics

Placing synchronisation mechanisms (fences, dependencies) to restore SC



Restoring Sequential Consistency (our objective)

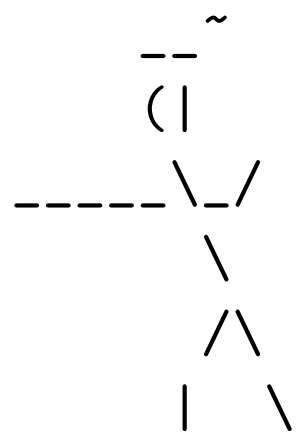
```
3 void* thread_0 (void* arg) {
4   data = 1;
5   __asm__ ("lwsync");
6   flag = 1;
7 }
```

```
9 void* thread_1 (void* arg) {
10  int flag_set = flag;
11  __asm__ ("lwsync");
12  int data_upd = data;
13  assert( flag_set ⇒ data_upd );
14 }
```

Concurrent programming with **interleavings** in mind is simpler than with processor specifics

Placing synchronisation mechanisms (fences, dependencies) to restore SC

Weak behaviours no longer permitted



Restoring Sequential Consistency (our objective)

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   __asm__ ("lwsync");
6   flag = 1;
7 }
```

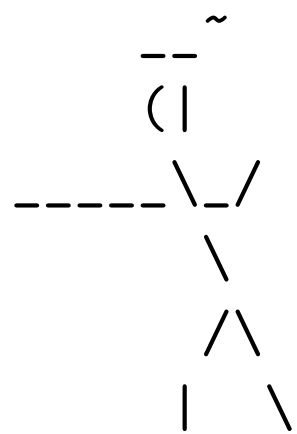
```
9 void* thread_1 (void* arg) {
10  int flag_set = flag;
11  __asm__ ("lwsync");
12  int data_upd = data;
13  assert( flag_set ⇒ data_upd );
14 }
```

Concurrent programming with **interleavings** in mind is simpler than with processor specifics

Placing synchronisation mechanisms (fences, dependencies) to restore SC

Weak behaviours no longer permitted

Soundly: *all* the weak behaviours forbidden



Restoring Sequential Consistency (our objective)

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   __asm__ ("lwsync");
6   flag = 1;
7 }
```

```
9  void* thread_1 (void* arg) {
10   int flag_set = flag;
11   __asm__ ("lwsync");
12   int data_upd = data;
13   assert( flag_set ⇒ data_upd );
14 }
```

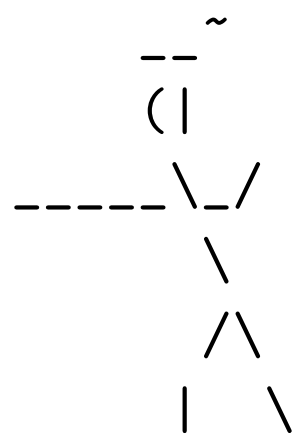
Concurrent programming with **interleavings** in mind is simpler than with processor specifics

Placing synchronisation mechanisms (fences, dependencies) to restore SC

Weak behaviours no longer permitted

Soundly: *all* the weak behaviours forbidden

Adequately: only preventing reorderings of accesses affecting semantics



An automated synchronisation synthesis tool (our solution)

```
[vincent@localhost musketeer]$ ./musketeer

* *     musketeer 0.37     * *

              ~__
               |)
              /|_____
             /  |
              /|
             / |

Usage:                          Purpose:

 musketeer [-?] [-h] [--help] show help

Main options:

 --mm <tso,pso,rmo,power>      detects all the fences to insert for a weak
                               memory model

Alternative methods:
```

We offer a tool, **musketeer**, that

An automated synchronisation synthesis tool (our solution)

```
[vincent@localhost musketeer]$ ./musketeer

* *     musketeer 0.37     * *

              ~__
               |)
              /|_____
             /  |
              /|
             / |

Usage:                          Purpose:

 musketeer [-?] [-h] [--help] show help

Main options:

 --mm <tso,pso,rmo,power>      detects all the fences to insert for a weak
                               memory model

Alternative methods:
```

We offer a tool, **musketeer**, that

- analyses concurrent C programs for TSO/x86 to Power/ARM

An automated synchronisation synthesis tool (our solution)

```
[vincent@localhost musketeer]$ ./musketeer

* *     musketeer 0.37     * *

              ~__
               |)
              /|_____
             /  |
              /|
             / |

Usage:                          Purpose:

 musketeer [-?] [-h] [--help] show help

Main options:

 --mm <tso,pso,rmo,power>     detects all the fences to insert for a weak
                              memory model

Alternative methods:
```

We offer a tool, **musketeer**, that

- analyses concurrent C programs for TSO/x86 to Power/ARM
- **synthesises** fences and dependencies for restoring SC

An automated synchronisation synthesis tool (our solution)

```
[vincent@localhost musketeer]$ ./musketeer

* *     musketeer 0.37     * *

              ~__
               |)
              /|_____
             /  |
              /|
             / |

Usage:                          Purpose:

 musketeer [-?] [-h] [--help] show help

Main options:

 --mm <tso,pso,rmo,power>      detects all the fences to insert for a weak
                               memory model

Alternative methods:
```

We offer a tool, **musketeer**, that

- analyses concurrent C programs for TSO/x86 to Power/ARM
- **synthesises** fences and dependencies for restoring SC
- **minimises** the runtime impact of these fences

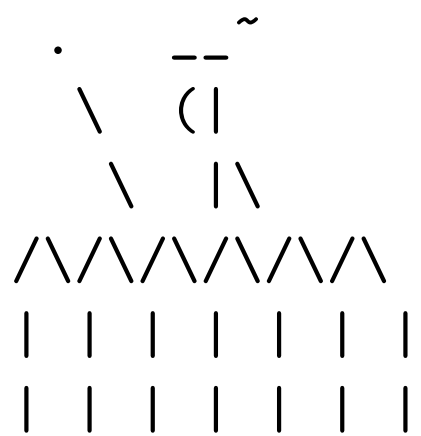
An automated synchronisation synthesis tool (our solution)

```
[vincent@localhost musketeer]$ ./musketeer

* *     musketeer 0.37     * *

              ~__
               |)
              /|_____
             /  |
              /|
             / |

Usage:                          Purpose:

 musketeer [-?] [-h] [--help] show help

Main options:

 --mm <tso,pso,rmo,power>      detects all the fences to insert for a weak
                               memory model

Alternative methods:
```

We offer a tool, **musketeer**, that

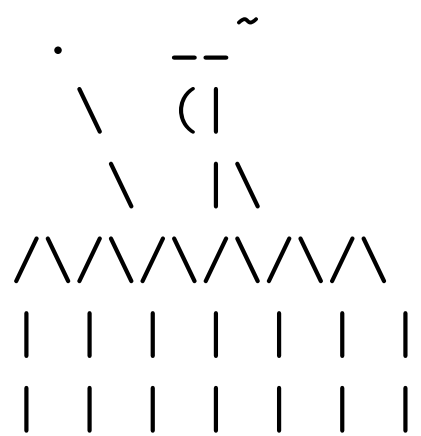
- analyses concurrent C programs for TSO/x86 to Power/ARM
- **synthesises** fences and dependencies for restoring SC
- **minimises** the runtime impact of these fences
- performs a source-to-source transformation in an **automated** manner

Why automating synthesis? (our motivation)



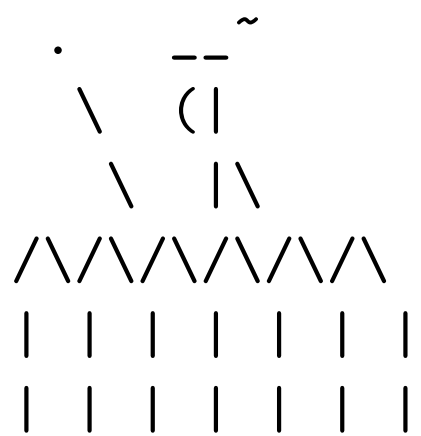
Why automating synthesis? (our motivation)

- Inserting manually requires lots of work



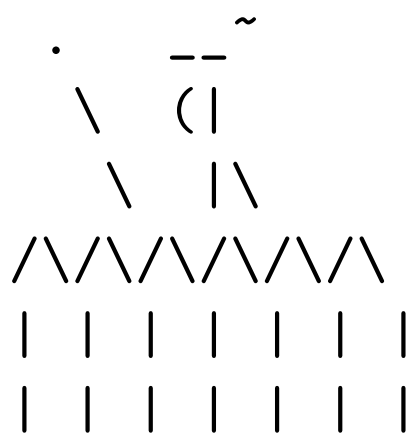
Why automating synthesis? (our motivation)

- Inserting manually requires lots of work
- The semantics of memory fences can be subtle



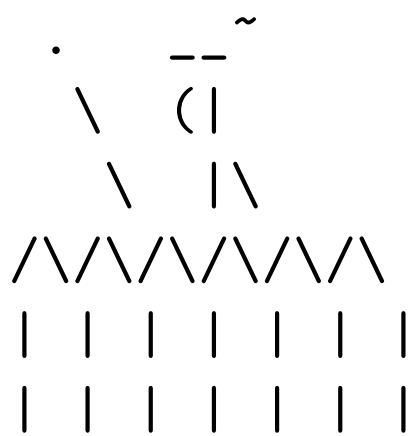
Why automating synthesis? (our motivation)

- Inserting manually requires lots of work
- The semantics of memory fences can be subtle
- Architectures allow different reorderings



Why automating synthesis? (our motivation)

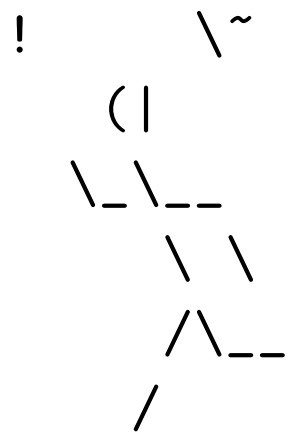
- Inserting manually requires lots of work
- The semantics of memory fences can be subtle
- Architectures allow different reorderings



restored order	ARM fences	Power fences
store-read	dsb	sync
store-store	dsb	sync, lwsync
read-store	dsb, dp	sync, lwsync, dp
read-read	dsb, dp, bcc;isb	sync, lwsync, dp, bcc;isync

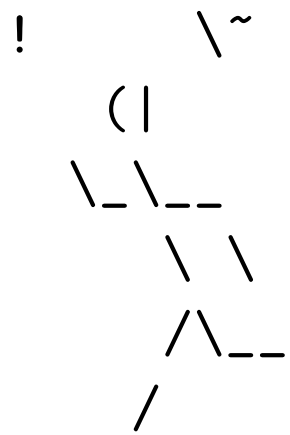
Figure: Effect of memory barriers on delayed pairs of events [Alglave et al., CAV'10].

Why optimising? (our motivation)



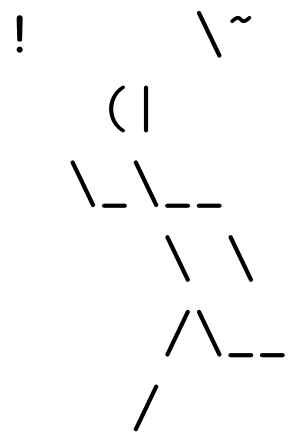
Why optimising? (our motivation)

- Fences are slow!



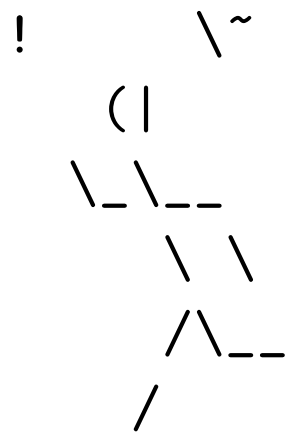
Why optimising? (our motivation)

- Fences are slow!
- Benign reorderings (i.e., which do not affect the semantics) improve performance



Why optimising? (our motivation)

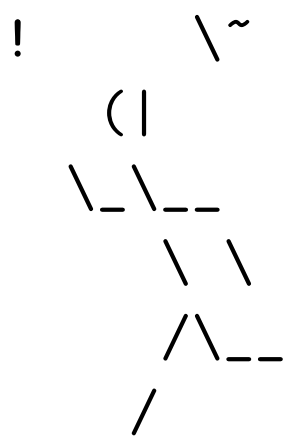
- Fences are slow!
- Benign reorderings (i.e., which do not affect the semantics) improve performance
- Some types of fences are cheaper than others (lwsync, isync...)

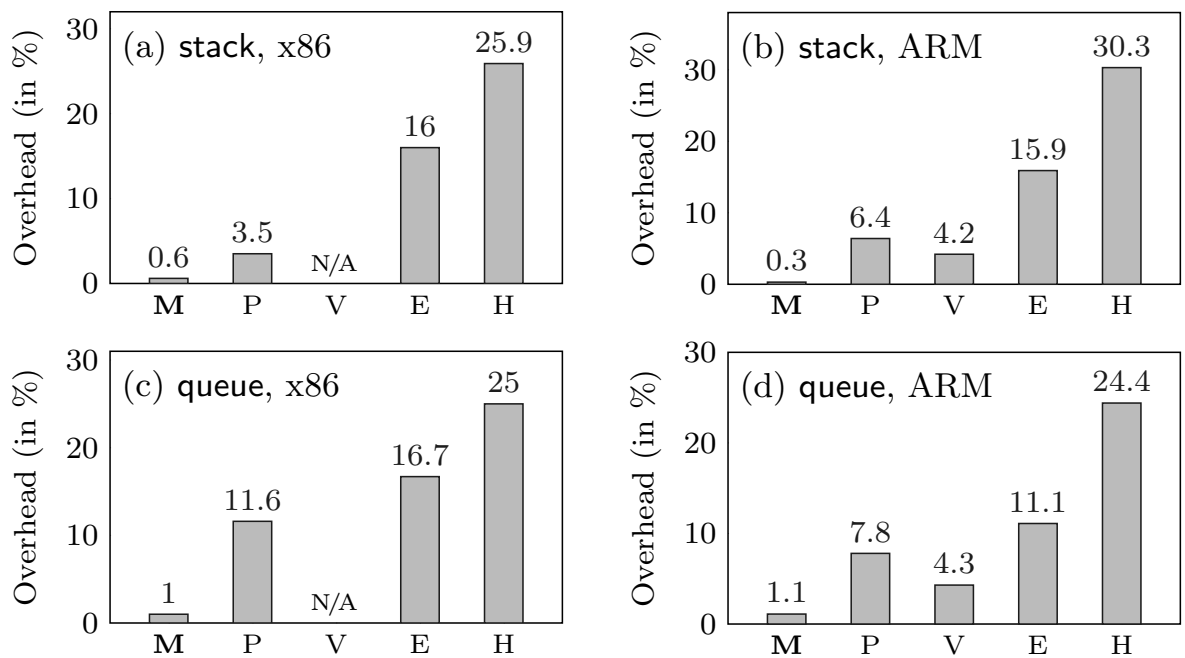


Why optimising? (our motivation)

- Fences are slow!
- Benign reorderings (i.e., which do not affect the semantics) improve performance
- Some types of fences are cheaper than others (lwsync, isync...)

O original
M musketeer (our tool)
P pensieve
V volatile (unsound)
E escape analysis
H heap and static mem.

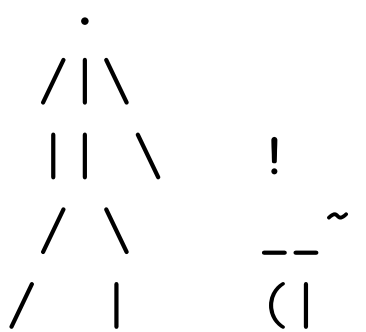




Average overhead of execution time after fence insertion

Executions and Critical Cycles

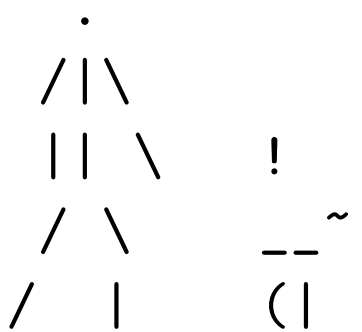
```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11   assert (   flag_set   ⇒ data_upd );
12 }
```



Executions and Critical Cycles

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert ( flag_set ⇒ data_upd );
12 }
```

(4) Wdata1

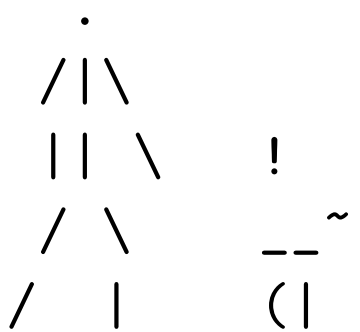


Executions and Critical Cycles

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11   assert (   flag_set   ⇒ data_upd );
12 }
```

(4) Wdata1

(5) Wflag1

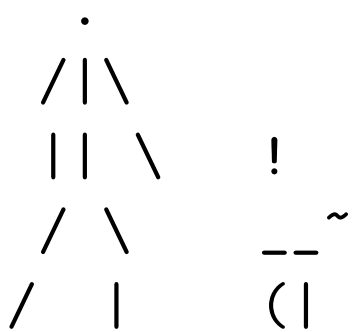


Executions and Critical Cycles

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert ( flag_set ⇒ data_upd );
12 }
```

(4) Wdata1 (9) Rflag1

(5) Wflag1

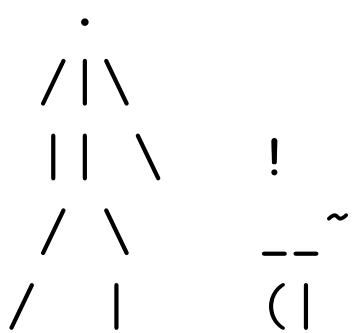


Executions and Critical Cycles

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert ( flag_set ⇒ data_upd );
12 }
```

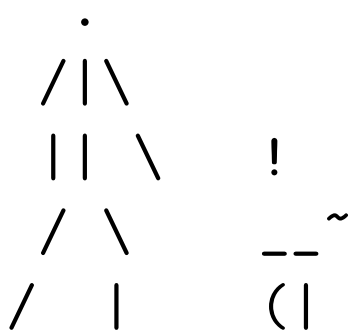
(4) Wdata1 (9) Rflag1

(5) Wflag1 (10) Rdata0



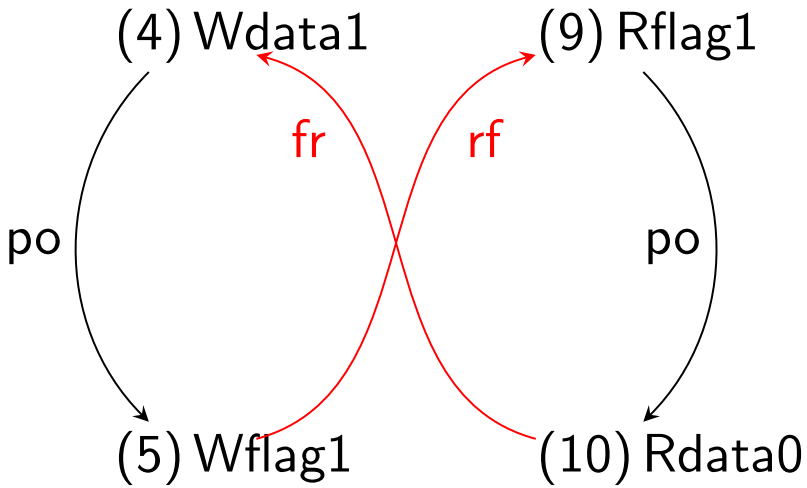
Executions and Critical Cycles

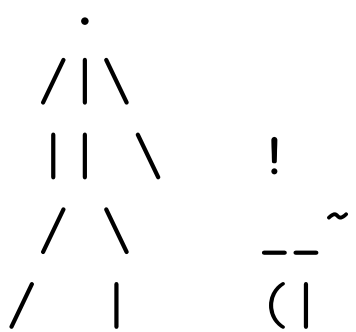
```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert ( flag_set ⇒ data_upd );
12 }
```



Executions and Critical Cycles

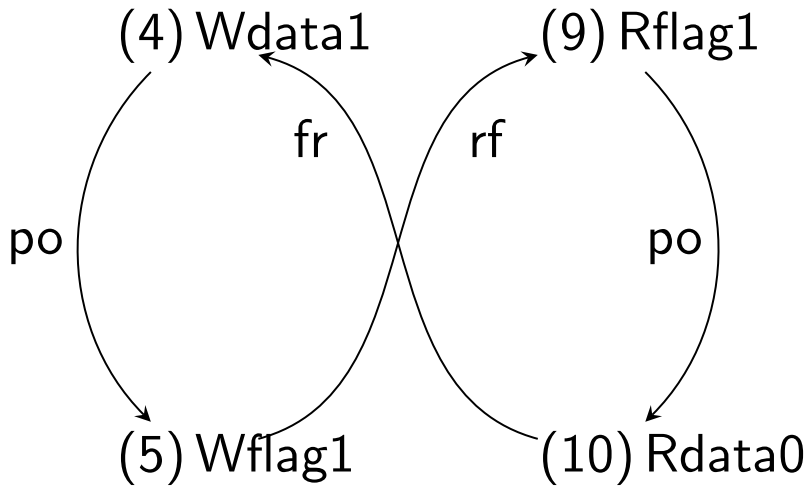
```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert ( flag_set ⇒ data_upd );
12 }
```



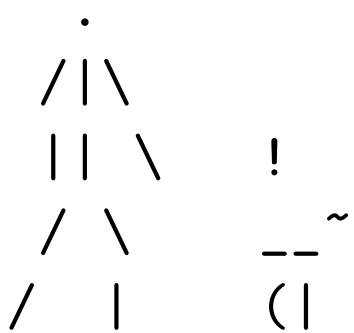


Executions and Critical Cycles

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert ( flag_set ⇒ data_upd );
12 }
```

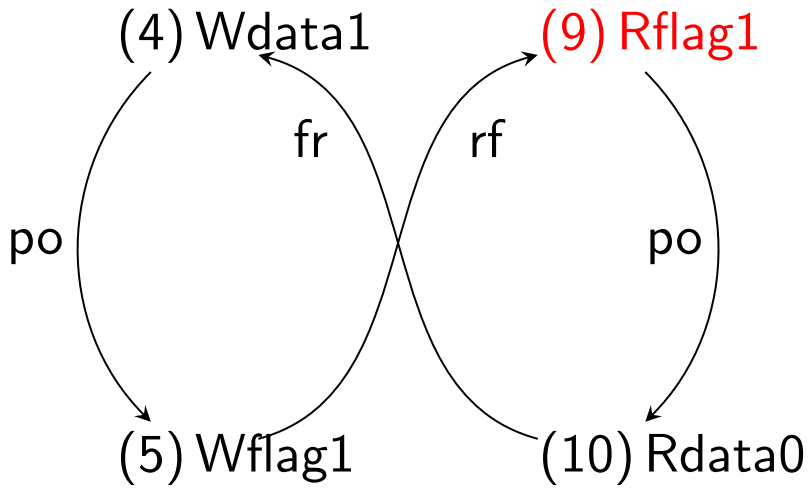


- Under SC, orders enforced by the processor;



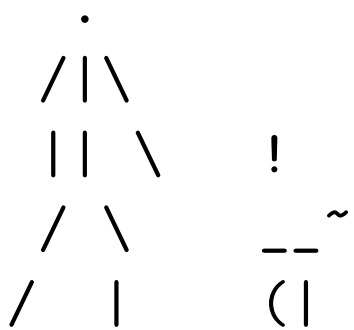
Executions and Critical Cycles

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert ( flag_set ⇒ data_upd );
12 }
```



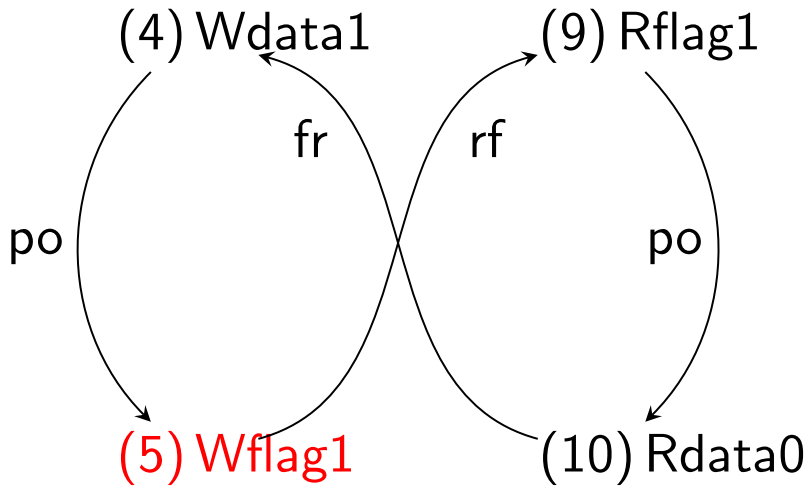
- Under SC, orders enforced by the processor;

(9)

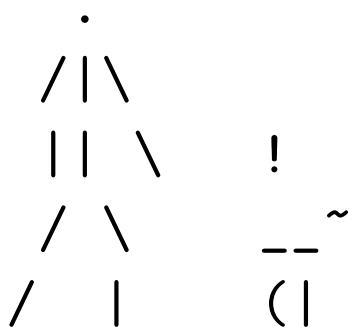


Executions and Critical Cycles

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert ( flag_set ⇒ data_upd );
12 }
```

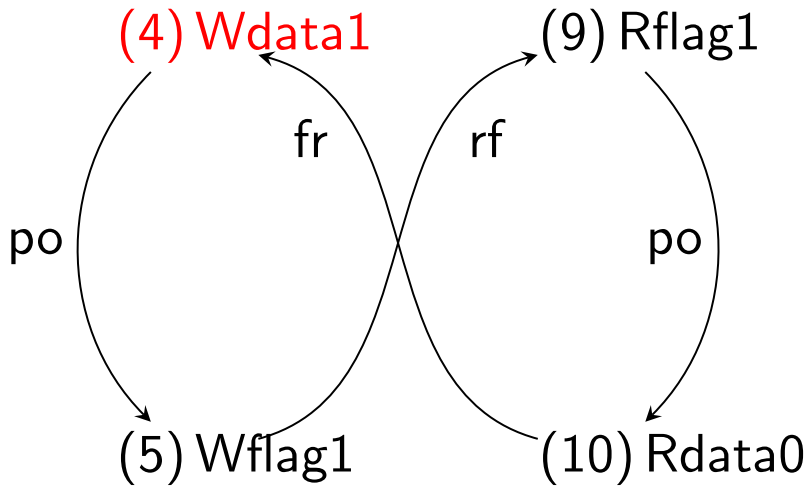


- Under SC, orders enforced by the processor;
  (5)<(9)

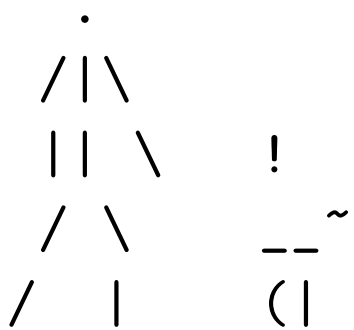


Executions and Critical Cycles

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert ( flag_set ⇒ data_upd );
12 }
```

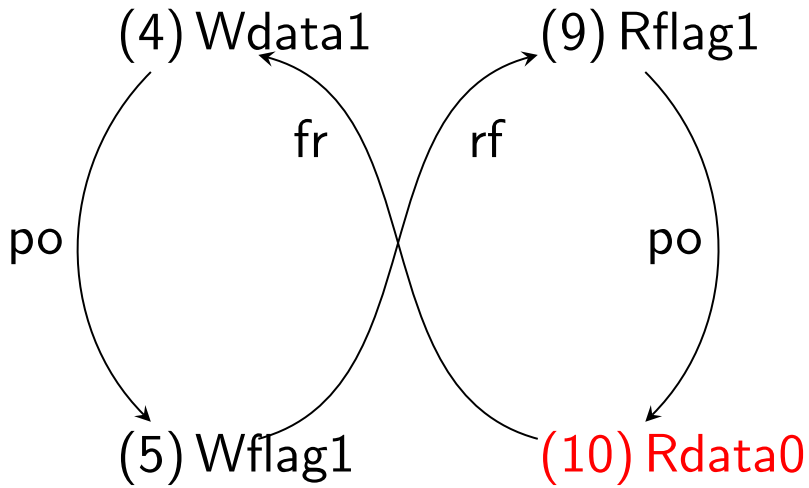


- Under SC, orders enforced by the processor;
  (4)<(5)<(9)

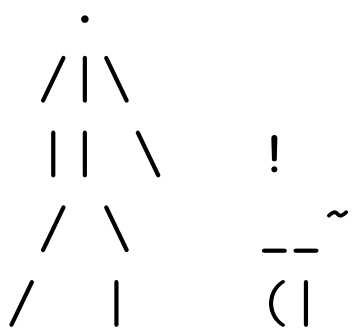


Executions and Critical Cycles

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert ( flag_set ⇒ data_upd );
12 }
```

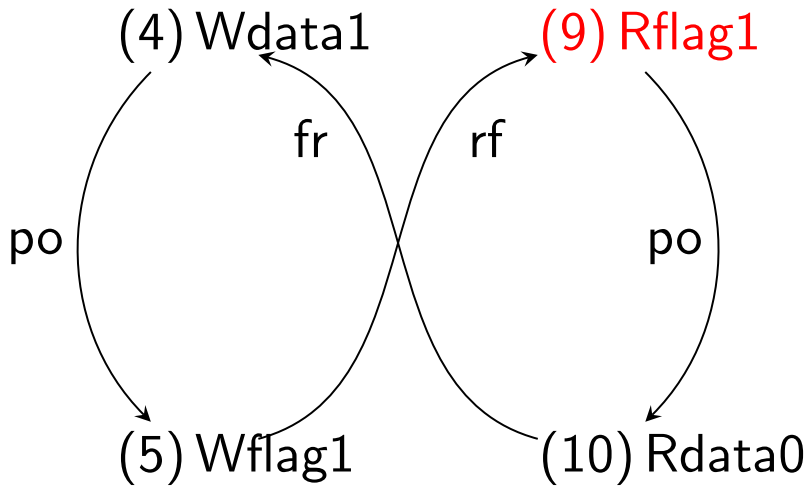


- Under SC, orders enforced by the processor;
  (10)<(4)<(5)<(9)

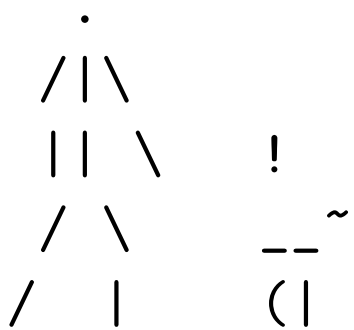


Executions and Critical Cycles

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert ( flag_set ⇒ data_upd );
12 }
```

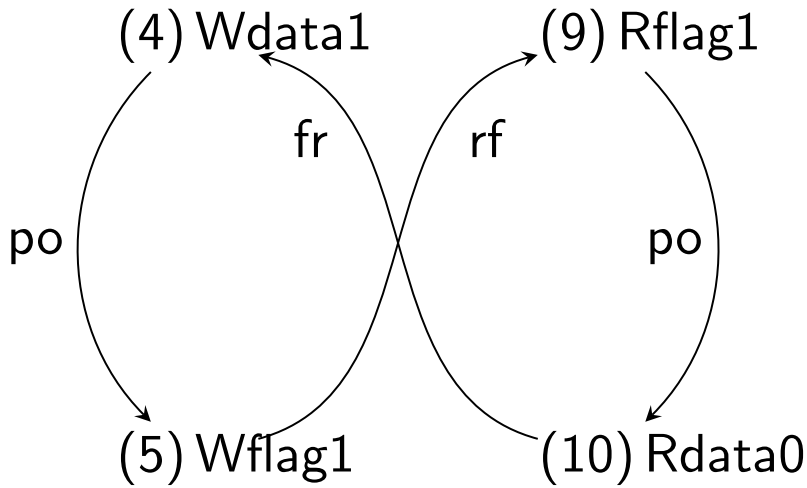


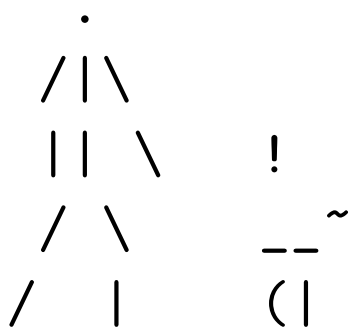
- Under SC, orders enforced by the processor;
  (9)<(10)<(4)<(5)<(9)



Executions and Critical Cycles

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert ( flag_set ⇒ data_upd );
12 }
```

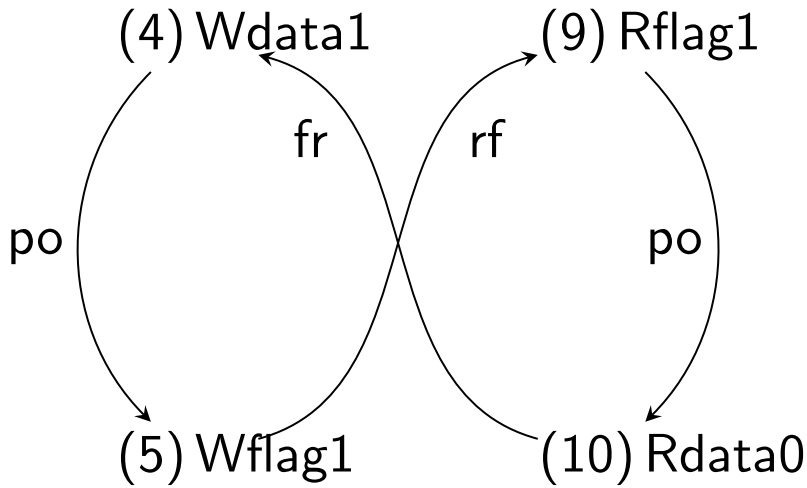


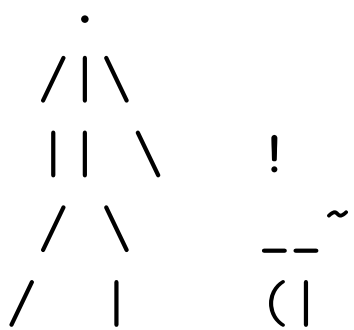


- Under SC, orders enforced by the processor;
  (9)<(10)<(4)<(5)<(9)
  contradiction in the *global-happen-before*;

Executions and Critical Cycles

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert ( flag_set ⇒ data_upd );
12 }
```



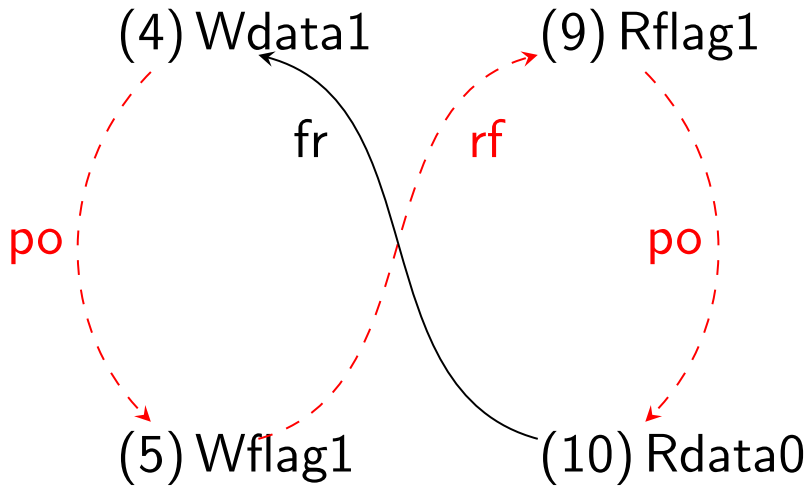


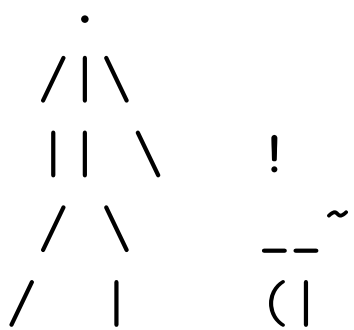
- Under SC, orders enforced by the processor;
  $(9)<(10)<(4)<(5)<(9)$
  contradiction in the *global-happen-before*;

cycle existence ⇒ execution impossible

Executions and Critical Cycles

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert ( flag_set ⇒ data_upd );
12 }
```





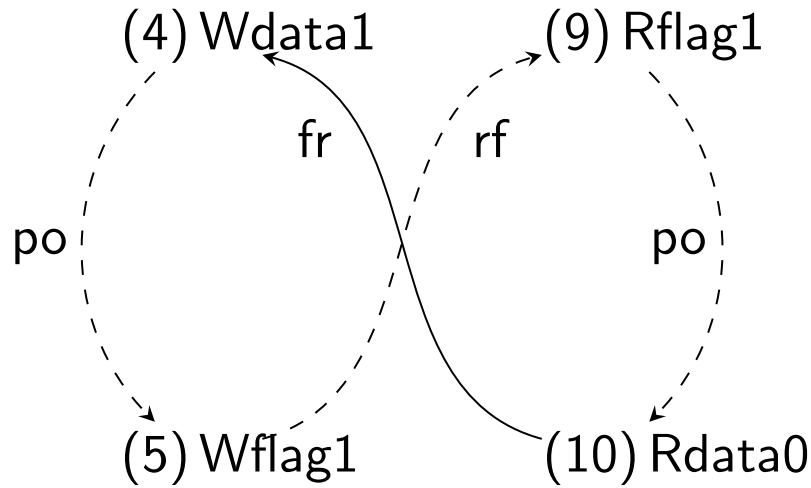
- Under SC, orders enforced by the processor;
  $(9)<(10)<(4)<(5)<(9)$
  contradiction in the *global-happen-before*;

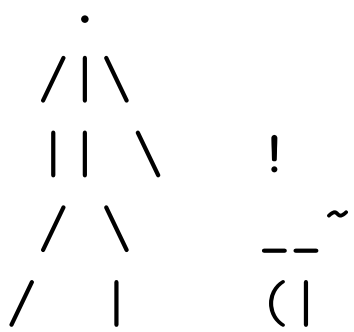
  cycle existence ⇒ execution impossible

- Under Power, some orders are relaxed;

Executions and Critical Cycles

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert ( flag_set ⇒ data_upd );
12 }
```





- Under SC, orders enforced by the processor;
  $(9)<(10)<(4)<(5)<(9)$
  contradiction in the *global-happen-before*;

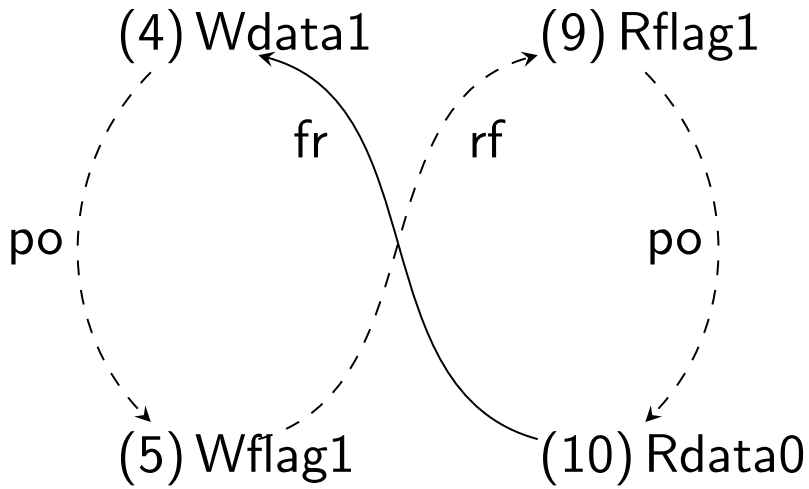
  cycle existence ⇒ execution impossible

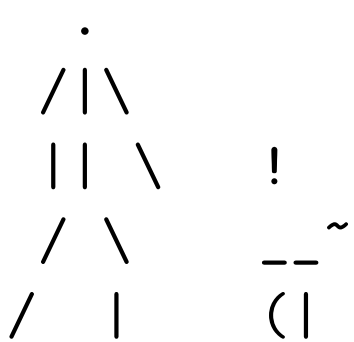
- Under Power, some orders are relaxed;
  $(9)<(5)<(10)<(4)$

  no critical cycle ⇒ execution possible

Executions and Critical Cycles

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert ( flag_set ⇒ data_upd );
12 }
```

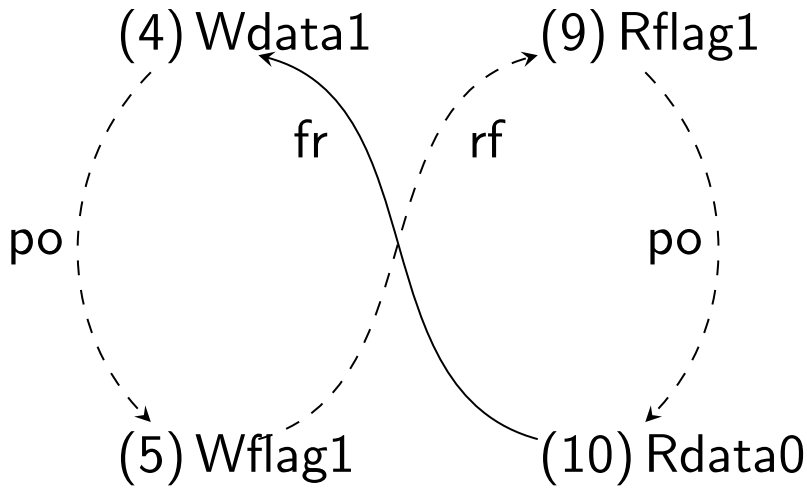


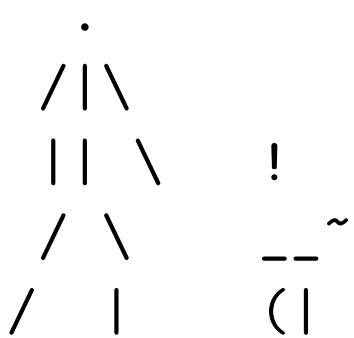


- We target all the cycles for SC that are not cycles for Power

Executions and Critical Cycles

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert ( flag_set ⇒ data_upd );
12 }
```



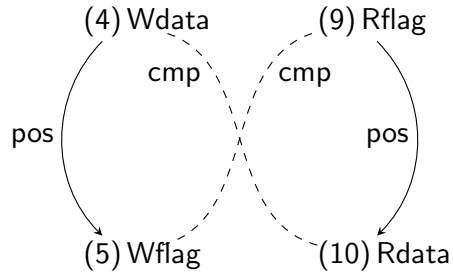


- We target all the cycles for SC that are not cycles for Power
- We modify the code so that they would also be cycles for Power

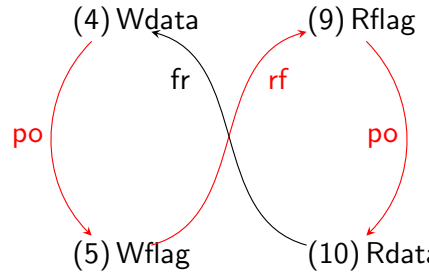
Automatic source-to-source transformation

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   flag = 1;
6 }

8 void* thread_1 (void* arg) {
9   int flag_set = flag;
10  int data_upd = data;
11  assert ( flag_set ⇒ data_upd );
12 }
```

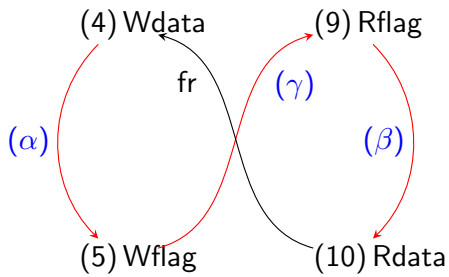
(1) program



(2) CFG static abstraction

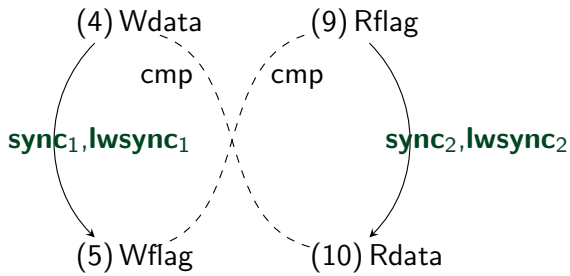


(3) find critical cycles



min($\mathbf{sync}_1 + \mathbf{sync}_2$) × cost(f)
+ ($\mathbf{lwsync}_1 + \mathbf{lwsync}_2$) × cost(lwf)
s.t.
(α): $\mathbf{sync}_1 + \mathbf{lwsync}_1 \geq 1$
(β): $\mathbf{sync}_2 + \mathbf{lwsync}_2 \geq 1$
(γ): $\mathbf{sync}_1 + \mathbf{lwsync}_1 + \mathbf{sync}_2 + \mathbf{lwsync}_2 \geq 1$

(4) Integer Linear Prog.



$\mathbf{sync}_1 = 0$, $\mathbf{lwsync}_1 = 1$,
$\mathbf{sync}_2 = 0$, $\mathbf{lwsync}_2 = 1$

lwsync in thread_0 lines 4 and 5
lwsync in thread_1 lines 9 and 10

(5) solve ILP with GLPK

```
3 void* thread_0 (void* arg) {
4   data = 1;
5   __asm__ ("lwsync");
6   flag = 1;
7 }

9 void* thread_1 (void* arg) {
10  int flag_set = flag;
11  __asm__ ("lwsync");
12  int data_upd = data;
13  assert ( flag_set ⇒ data_upd );
14 }
```

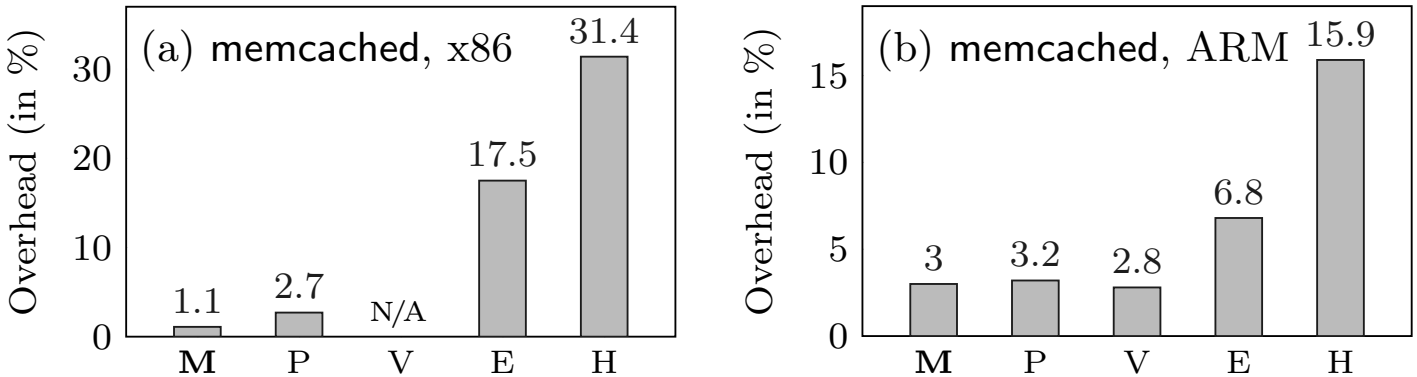
(6) insert fences

Experiments

1. classic examples: mutex algorithms (Dekker, Lamport, Szymanski...)
2. parametric examples
3. Debian executables (>700): in particular, memcached (high-performance caching system used e.g. by Facebook)

			x86		Power	
	LoC	build time	fences	time	fences	time
memcached	9944	59.3s	3	13.9s	70	89.9s
lingot	2894	56.8s	0	5.3s	5	5.3s
weborf	2097	65.8s	0	0.7s	0	0.7s
timemachine	1336	25.4s	2	0.8s	16	0.8s
see	2626	51.6s	0	1.4s	0	1.5s
blktrace	1567	41.7s	0	6.5s	–	timeout
ptunnel	1249	2.1s	2	95.0s	–	timeout
proxsmtpd	2024	53.5s	0	0.1s	0	0.1s
ghostess	2684	51.7s	0	25.9s	0	25.9s
dnshistory	1516	107.4s	1	29.4s	9	64.9s

Performance impact on memcached



Average overhead of execution time after fence insertion

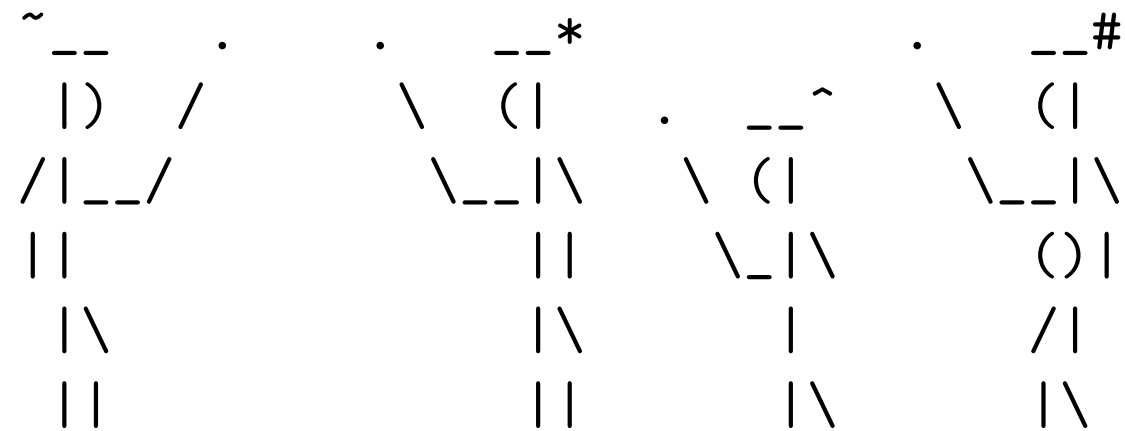
	memcached on x86	memcached on ARM	pfscan on x86	pfscan on ARM
(O)	[29.461; 29.699]	[16.248; 16.553]	[15.013; 15.034]	[19.066; 19.096]
(M)	[29.776; 30.049]	[16.748; 17.023]	[15.073; 15.097]	[19.101; 19.128]
(P)	[30.259; 30.517]	[16.795; 17.063]	[15.083; 15.107]	[19.411; 19.437]
(V)	N/A	[15.074; 15.097]	N/A	[15.083; 15.107]
(E)	[34.631; 34.883]	[17.402; 17.636]	[15.086; 15.118]	[19.659; 19.684]
(H)	[38.751; 39.050]	[18.916; 19.098]	[15.270; 15.293]	[34.422; 34.457]

Confidence intervals for N=100, α=5%

We also computed Student's T-tests for checking statistical significance

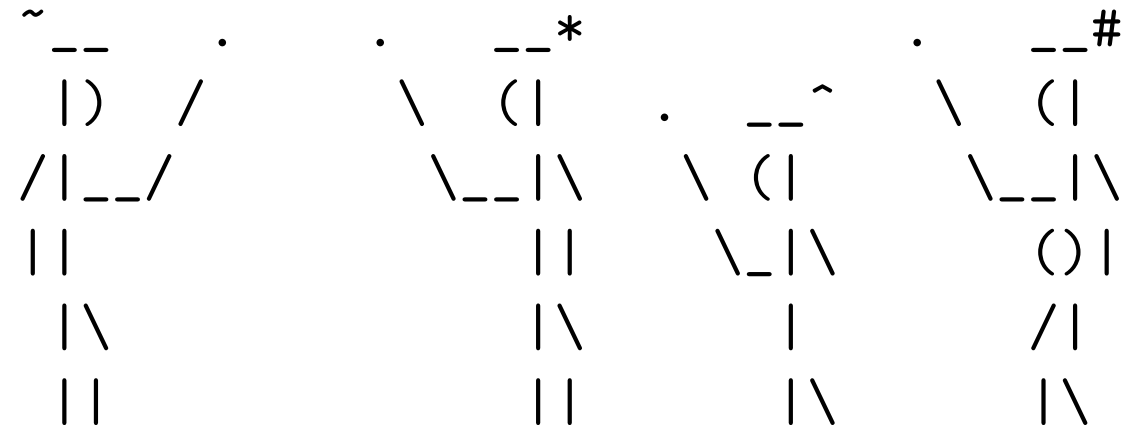
Limitations of musketeer

- musketeer works on a sound over-approximation



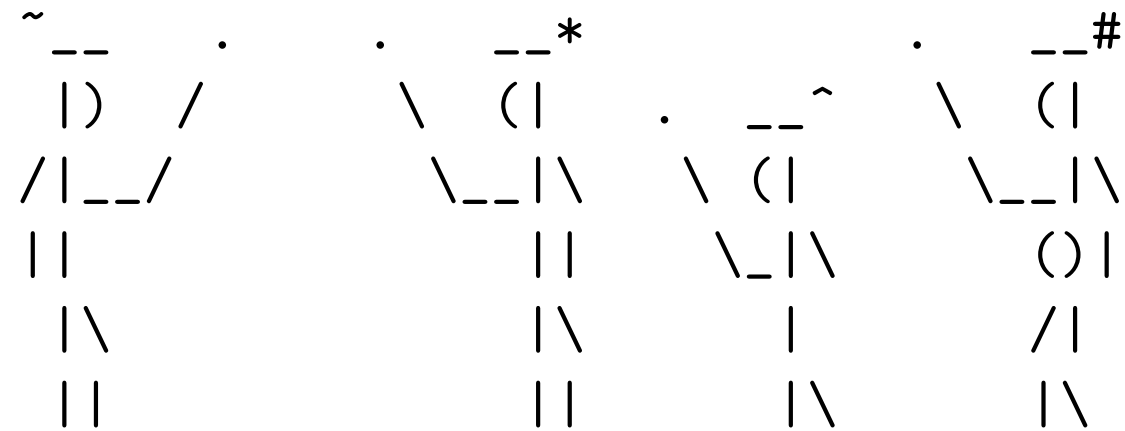
Limitations of musketeer

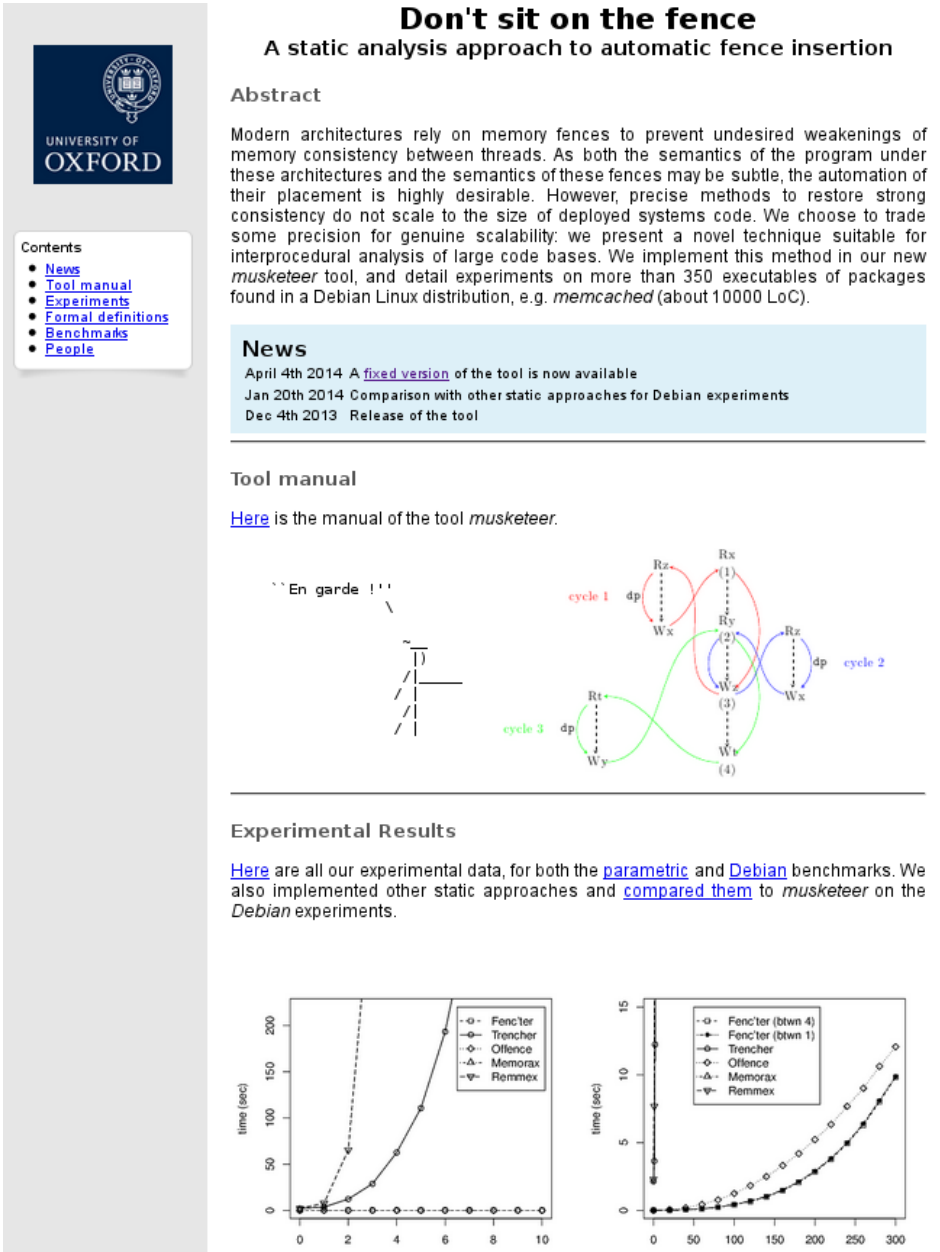
- musketeer works on a sound over-approximation
  - no evaluation of the branching conditions
  - no synchronisation analysis
  - abstraction of the loops



Summary of other existing tools

	tool	*group*	*architecture*
precise	blender, fender	(Kuperstein et al.)	x86 ... RMO
	memorax	(Abdulla et al.)	x86
	offence	(Alglave et al.)	x86 ... Power
	remmex	(Linden and Wolper)	x86, PSO
	trencher	(Bouajjani et al.)	x86
dynamic	dfence	(Liu et al.)	x86





Don't sit on the fence

A static analysis approach to automatic fence insertion

Abstract

Modern architectures rely on memory fences to prevent undesired weakenings of memory consistency between threads. As both the semantics of the program under these architectures and the semantics of these fences may be subtle, the automation of their placement is highly desirable. However, precise methods to restore strong consistency do not scale to the size of deployed systems code. We choose to trade some precision for genuine scalability: we present a novel technique suitable for interprocedural analysis of large code bases. We implement this method in our new *musketeer* tool, and detail experiments on more than 350 executables of packages found in a Debian Linux distribution, e.g. *memcached* (about 10000 LoC).

Contents

- News
- Tool manual
- Experiments
- Formal definitions
- Benchmarks
- People

News

April 4th 2014 A fixed version of the tool is now available

Jan 20th 2014 Comparison with other static approaches for Debian experiments

Dec 4th 2013 Release of the tool

Tool manual

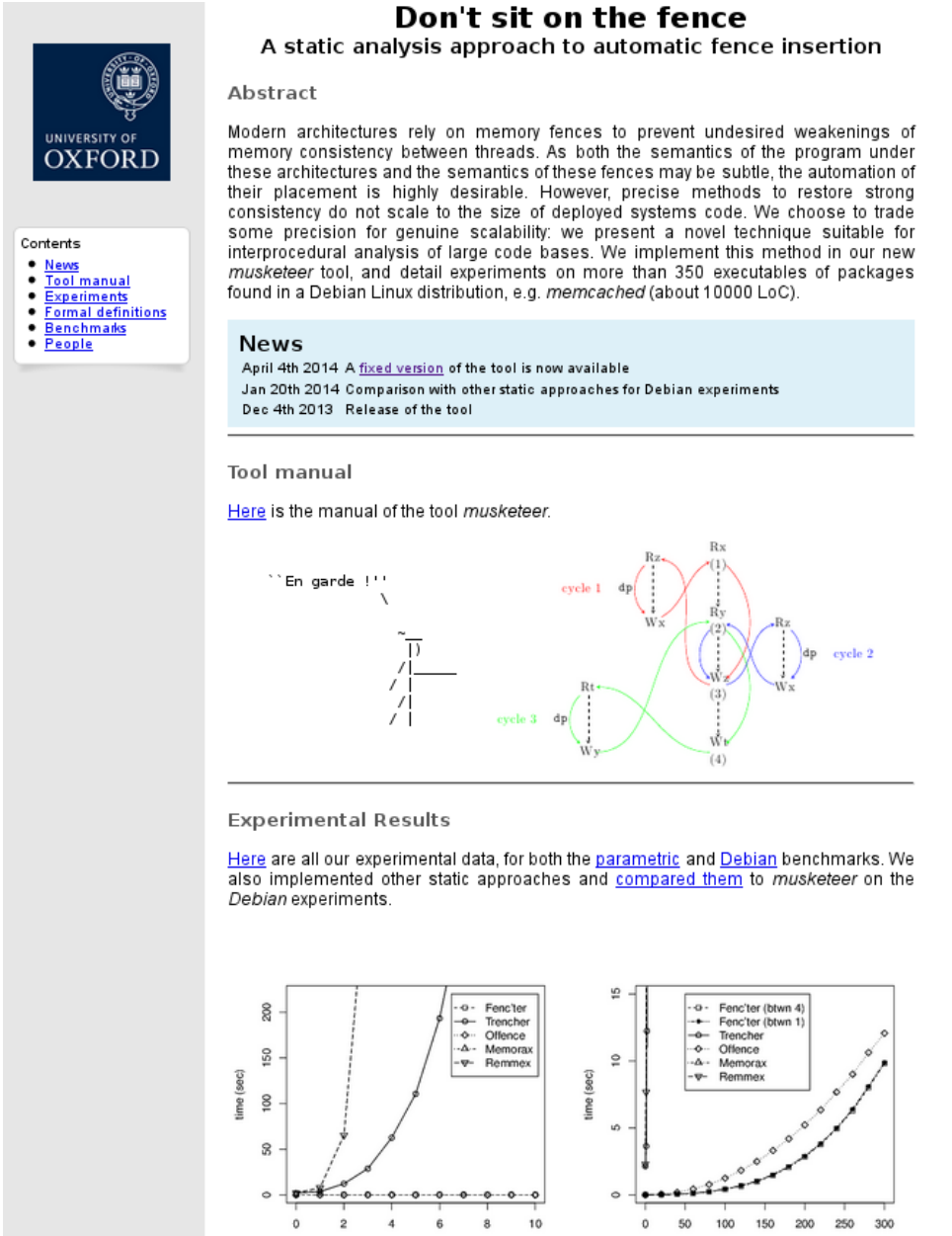
Here is the manual of the tool *musketeer*.

Experimental Results

Here are all our experimental data, for both the parametric and Debian benchmarks. We also implemented other static approaches and compared them to *musketeer* on the *Debian* experiments.

www.cprover.org/wmm/musketeer

- tool and manual
- benchmarks
- additional experiments



Don't sit on the fence

A static analysis approach to automatic fence insertion

Abstract

Modern architectures rely on memory fences to prevent undesired weakenings of memory consistency between threads. As both the semantics of the program under these architectures and the semantics of these fences may be subtle, the automation of their placement is highly desirable. However, precise methods to restore strong consistency do not scale to the size of deployed systems code. We choose to trade some precision for genuine scalability: we present a novel technique suitable for interprocedural analysis of large code bases. We implement this method in our new *musketeer* tool, and detail experiments on more than 350 executables of packages found in a Debian Linux distribution, e.g. *memcached* (about 10000 LoC).

Contents

- News
- Tool manual
- Experiments
- Formal definitions
- Benchmarks
- People

News

April 4th 2014 A fixed version of the tool is now available

Jan 20th 2014 Comparison with other static approaches for Debian experiments

Dec 4th 2013 Release of the tool

Tool manual

Here is the manual of the tool *musketeer*.

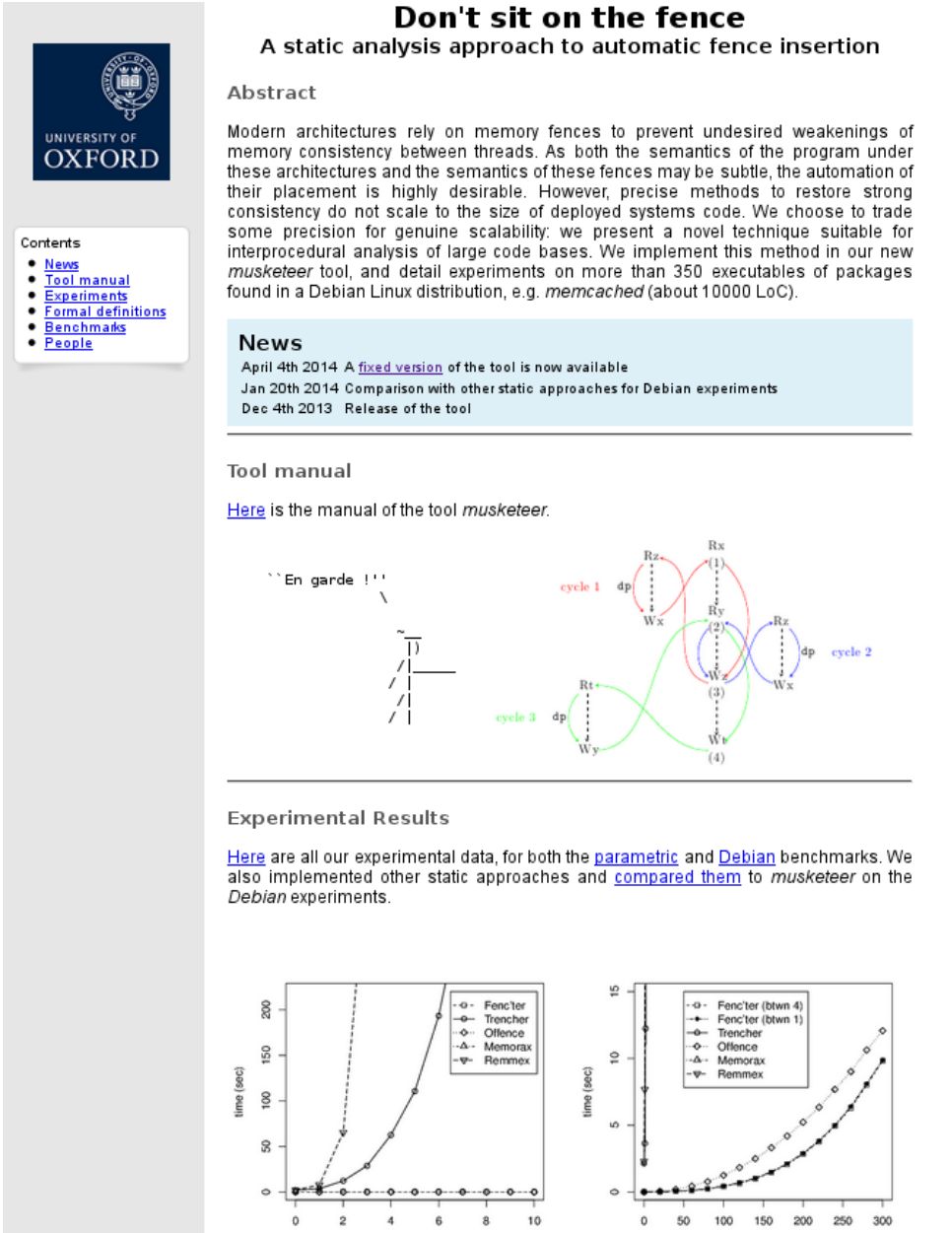
Experimental Results

Here are all our experimental data, for both the parametric and Debian benchmarks. We also implemented other static approaches and compared them to *musketeer* on the *Debian* experiments.

www.cprover.org/wmm/musketeer

- tool and manual
- benchmarks
- additional experiments

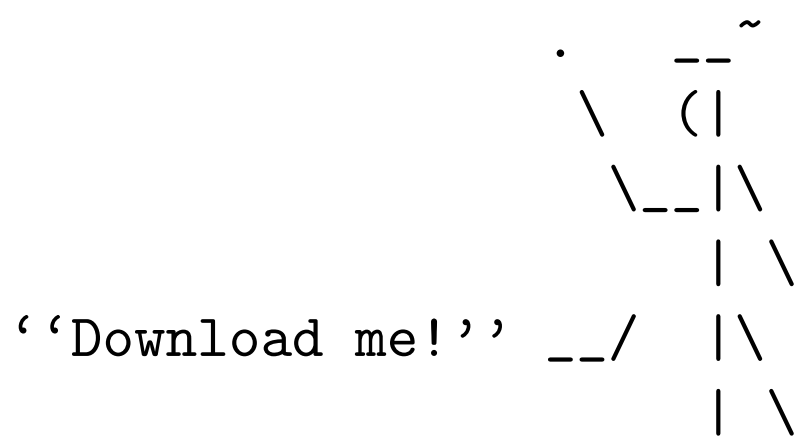
For more information, please do not hesitate to contact us to vincent.nimal@cs.ox.ac.uk



www.cprover.org/wmm/musketeer

- tool and manual
- benchmarks
- additional experiments

For more information, please do not hesitate to contact us to vincent.nimal@cs.ox.ac.uk



Alternative static methods

- (O) original (no additional fence inserted)
- (M) musketeer (our tool)
- (P) pensieve (in essence: all the pairs with communications)
- (E) escape analysis (in essence: all the pairs)
- (H) fences around heap and static variable accesses

Statistical evaluation of the experiments (1/2)

	memcached on x86	memcached on ARM	pfscan on x86	pfscan on ARM
(O)	[29.461; 29.699]	[16.248; 16.553]	[15.013; 15.034]	[19.066; 19.096]
(M)	[29.776; 30.049]	[16.748; 17.023]	[15.073; 15.097]	[19.101; 19.128]
(P)	[30.259; 30.517]	[16.795; 17.063]	[15.083; 15.107]	[19.411; 19.437]
(V)	N/A	[15.074; 15.097]	N/A	[15.083; 15.107]
(E)	[34.631; 34.883]	[17.402; 17.636]	[15.086; 15.118]	[19.659; 19.684]
(H)	[38.751; 39.050]	[18.916; 19.098]	[15.270; 15.293]	[34.422; 34.457]

Confidence intervals for N=100, α=5%

	memcached on x86	memcached on ARM	pfscan on x86	pfscan on ARM
(M) vs. (P)	5.008	0.440	1.158	32.576

Student T-test of (M) vs. (P) for N=100, α=5%

t-value with 198 degrees of freedom at α=5% is 1.972

Statistical evaluation of the experiments (2/2)

	stack on x86	stack on ARM	queue on x86	queue on ARM
(O)	[9.757; 9.798]	[11.291; 11.369]	[11.947; 11.978]	[20.441; 20.634]
(M)	[9.818; 9.850]	[11.316; 11.408]	[12.067; 12.099]	[20.687; 20.857]
(P)	[10.077; 10.155]	[11.995; 12.109]	[13.339; 13.373]	[22.035; 22.240]
(V)	N/A	[11.779; 11.834]	N/A	[21.334; 21.526]
(E)	[11.316; 11.360]	[13.071; 13.200]	[13.949, 13.981]	[22.722; 22.903]
(H)	[12.286; 12.325]	[14.676; 14.844]	[14.941, 14.963]	[25.468; 25.633]

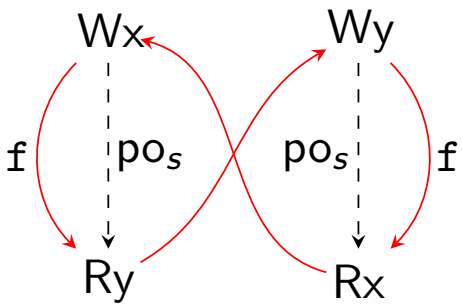
Confidence intervals for data structure experiments for N=100, α=5%

Variables and cost function

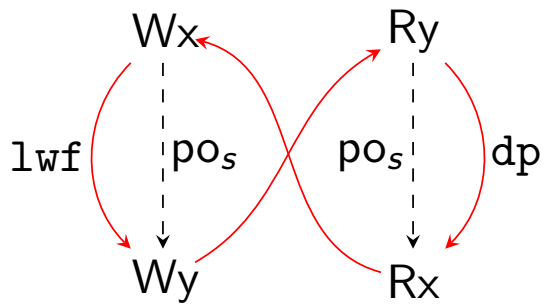
variables: $[\text{type of fence}]_{[\text{edge}]}$

$$\sum_{e \in \text{po}_s^+} \texttt{dp}_e * \text{cost}(\texttt{dp}) + \sum_{e' \in \text{po}_s} \texttt{dp}_{e'} * \text{cost}(\texttt{dp}) + \texttt{lwf}_{e'} * \text{cost}(\texttt{lwf})$$

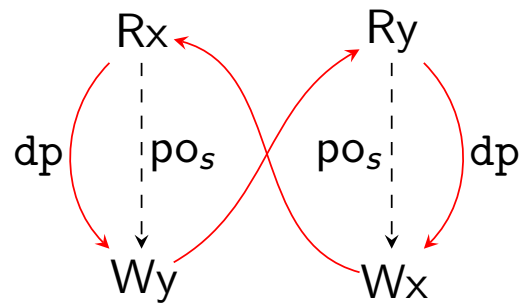
$$+\texttt{cf}_{e'} * \text{cost}(\texttt{cf}) + \texttt{f}_{e'} * \text{cost}(\texttt{f}) + \texttt{br}_{e'} * \text{cost}(\texttt{br}).$$



cost(cycle) = 6
(a) store buffering



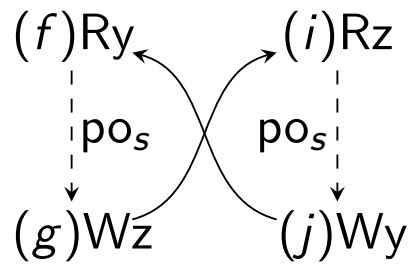
cost(cycle) = 3
(b) message passing



cost(cycle) = 2
(c) load buffering

Constraints

- Constraints ensure that each relevant delayed pair is prevented $\Rightarrow$ soundness.
- Suppose first that relaxed pairs are in po_s.



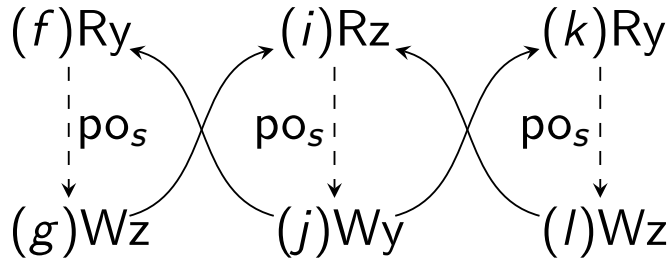
$$\textbf{min} \quad \mathtt{dp}_{(i,j)} + \mathtt{dp}_{(f,g)} + 3 * (\mathtt{f}_{(i,j)} + \mathtt{f}_{(f,g)}) + 2 * (\mathtt{lwf}_{(i,j)} + \mathtt{lwf}_{(f,g)})$$

$$\textbf{s.t.} \quad \text{delay } (f,g): \quad \mathtt{dp}_{(f,g)} + \mathtt{lwf}_{(f,g)} + \mathtt{f}_{(f,g)} \geq 1$$

$$\text{delay } (i,j): \quad \mathtt{dp}_{(i,j)} + \mathtt{lwf}_{(i,j)} + \mathtt{f}_{(i,j)} \geq 1$$

Constraints: Entangled Cycles

- Variables are shared between the constraints. A fence between a pair can affect another pair.

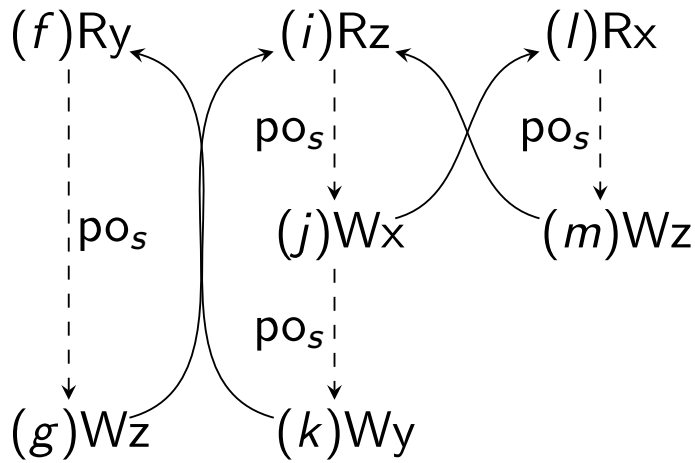


$$\textbf{min} \quad \mathtt{dp}_{(i,j)} + \mathtt{dp}_{(f,g)} + \mathtt{dp}_{(k,l)} + 3 * \left(\mathtt{f}_{(i,j)} + \mathtt{f}_{(f,g)} + \mathtt{f}_{(k,l)}\right) + 2 * \left(\mathtt{lwf}_{(i,j)} + \mathtt{lwf}_{(f,g)}\right) + \mathtt{lwf}_{(k,l)}$$

$$\begin{aligned} \textbf{s.t.} \quad & \text{delay } (f,g): && \mathtt{dp}_{(f,g)} + \mathtt{lwf}_{(f,g)} + \mathtt{f}_{(f,g)} \geq 1 \\ & \text{delay } (i,j): && \mathtt{dp}_{(i,j)} + \mathtt{lwf}_{(i,j)} + \mathtt{f}_{(i,j)} \geq 1 \\ & \text{delay } (k,l): && \mathtt{dp}_{(k,l)} + \mathtt{lwf}_{(k,l)} + \mathtt{f}_{(k,l)} \geq 1 \end{aligned}$$

Constraints: relaxed pairs are in po_s^+

- Relaxed pairs can be in po_s^+. To entangle cycles, we represent each po_s in the relaxed po_s^+.

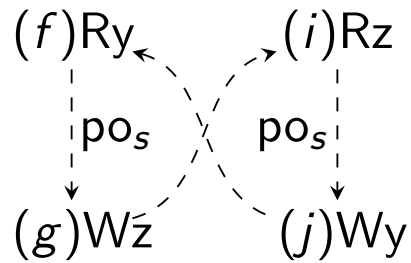


$$
\begin{aligned}
\textbf{min}\quad & \mathtt{dp}_{(i,j)} + \mathtt{dp}_{(f,g)} + \mathtt{dp}_{(j,k)} + \mathtt{dp}_{(l,m)} \\
& +3 * (\mathtt{f}_{(i,j)} + \mathtt{f}_{(f,g)} + \mathtt{f}_{(j,k)} + \mathtt{f}_{(l,m)}) \\
& +2 * (\mathtt{lwf}_{(i,j)} + \mathtt{lwf}_{(f,g)}) + \mathtt{lwf}_{(j,k)} + \mathtt{lwf}_{(l,m)})
\end{aligned}
$$

$$
\begin{aligned}
\textbf{s.t.}\quad & \text{delay } (f,g): && \mathtt{dp}_{(f,g)} + \mathtt{lwf}_{(f,g)} + \mathtt{f}_{(f,g)} \geq 1 \\
& \text{delay } (i,j): && \mathtt{dp}_{(i,j)} + \mathtt{lwf}_{(i,j)} + \mathtt{f}_{(i,j)} \geq 1 \\
& \text{delay } (i,k): && \mathtt{dp}_{(i,k)} + \mathtt{lwf}_{(i,j)} + \mathtt{f}_{(i,j)} + \mathtt{lwf}_{(j,k)} + \mathtt{f}_{(j,k)} \geq 1 \\
& \text{delay } (l,m): && \mathtt{dp}_{(l,m)} + \mathtt{lwf}_{(l,m)} + \mathtt{f}_{(l,m)} \geq 1
\end{aligned}
$$

Constraints: Cumulativity

- External rf can be reordered. Only lwsync and sync can fix.

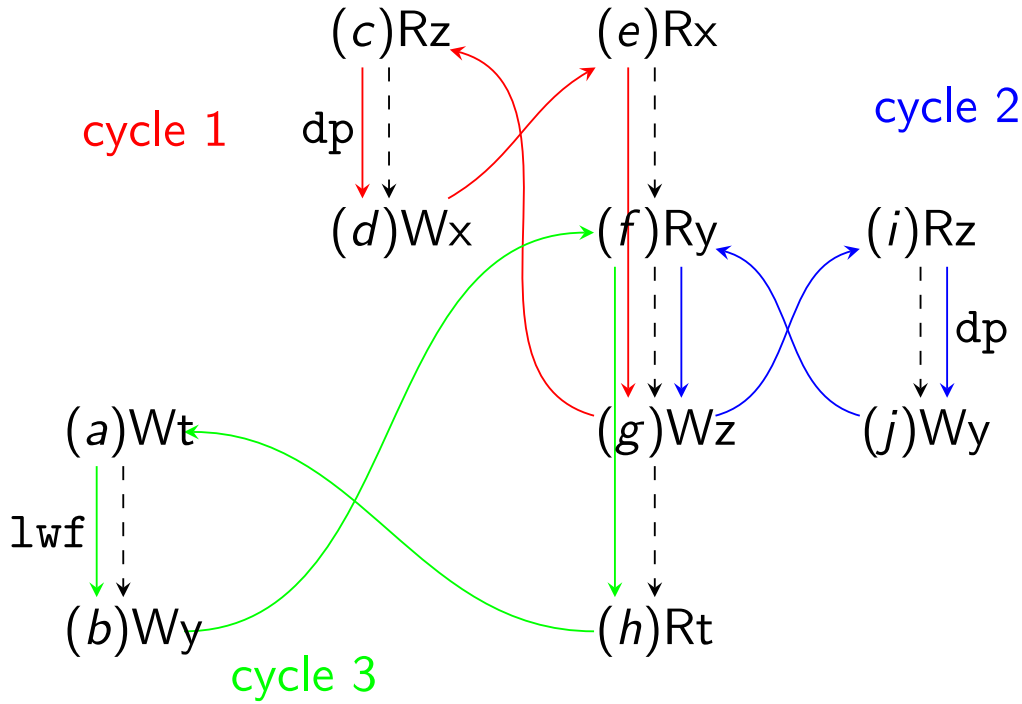


$$\textbf{min} \quad \mathtt{dp}_{(i,j)} + \mathtt{dp}_{(f,g)} + 3 * (\mathtt{f}_{(i,j)} + \mathtt{f}_{(f,g)}) + 2 * (\mathtt{lwf}_{(i,j)} + \mathtt{lwf}_{(f,g)})$$

s.t.

$$\begin{aligned}
&\text{delay } (f,g): && \mathtt{dp}_{(f,g)} + \mathtt{lwf}_{(f,g)} + \mathtt{f}_{(f,g)} \geq 1 \\
&\text{delay } (i,j): && \mathtt{dp}_{(i,j)} + \mathtt{lwf}_{(i,j)} + \mathtt{f}_{(i,j)} \geq 1 \\
&\text{delay } (j,f): && \mathtt{lwf}_{(f,g)} + \mathtt{f}_{(f,g)} + \mathtt{lwf}_{(i,j)} + \mathtt{f}_{(i,j)} \geq 1 \\
&\text{delay } (g,i): && \mathtt{lwf}_{(f,g)} + \mathtt{f}_{(f,g)} + \mathtt{lwf}_{(i,j)} + \mathtt{f}_{(i,j)} \geq 1
\end{aligned}$$

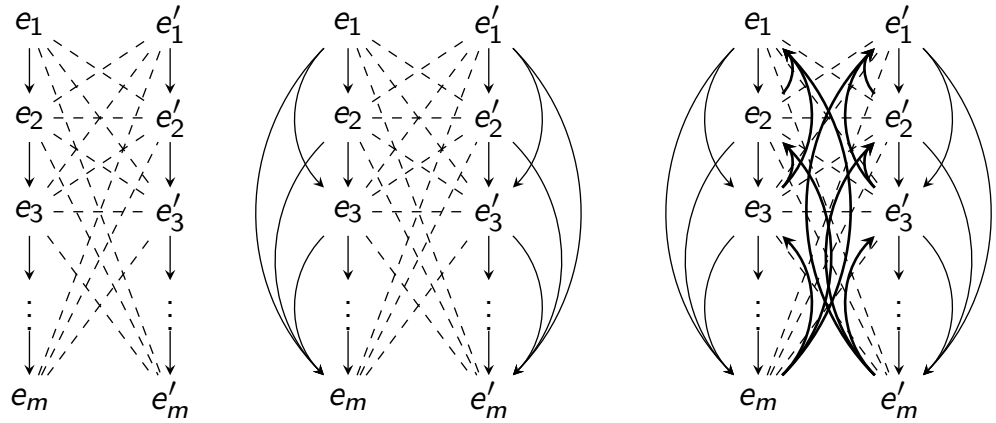
Cases where musketeer is more precise than trace-based



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

Figure: Example of resolution with btwn.

Complexity



$\sum_{i=2}^{n} \binom{n}{i} (A_m^2)^i$,

i.e. $o(m^{2n})$

step	complexity
Constructing the graph	$O(\#\mathbb{E}^2)$
Finding all the cycles	$O((\#\mathrm{po}_s + \#\,\mathrm{cmp} + \#\mathbb{E}) * \#C)$, with $\#C$ polynomial in $\#\mathbb{E}$ but exponential in $\#thds$
Constructing the ILP	linear (constant for variables, linear for constraints)
Solving ILP	NP (but fast in practice)
Inserting in the source	constant

Additional encodings

btwn	ILP variables	number of variables	complexity
btwn_1	po_s in the critical cycles	$\sum_{d \in \text{delays}} \#\text{soc}(d)$	$O(1)$
btwn_2	po_s in the intersections of pairs of critical cycles	$\sum_{d \in \text{delays} \cap \bigcup_{j \neq k} \max(\text{ctn}(C_j) \cap \text{ctn}(C_k))} \#\text{soc}(d)$	$O(\#\text{cycles}^2)$
btwn_3	po_s^+ at the intersections of any set of critical cycles	$\leq \# \bigcup_{j \neq k} \max(\text{ctn}(C_j) \cap \text{ctn}(C_k))$	$O(2^{\#\text{cycles}})$
btwn_4	po_s^+ at the intersections of any pair of critical cycles	$\leq \# \bigcup_{j \neq k} \max(\text{ctn}(C_j) \cap \text{ctn}(C_k))$	$O(\#\text{cycles}^2)$

Evaluation of the additional encodings