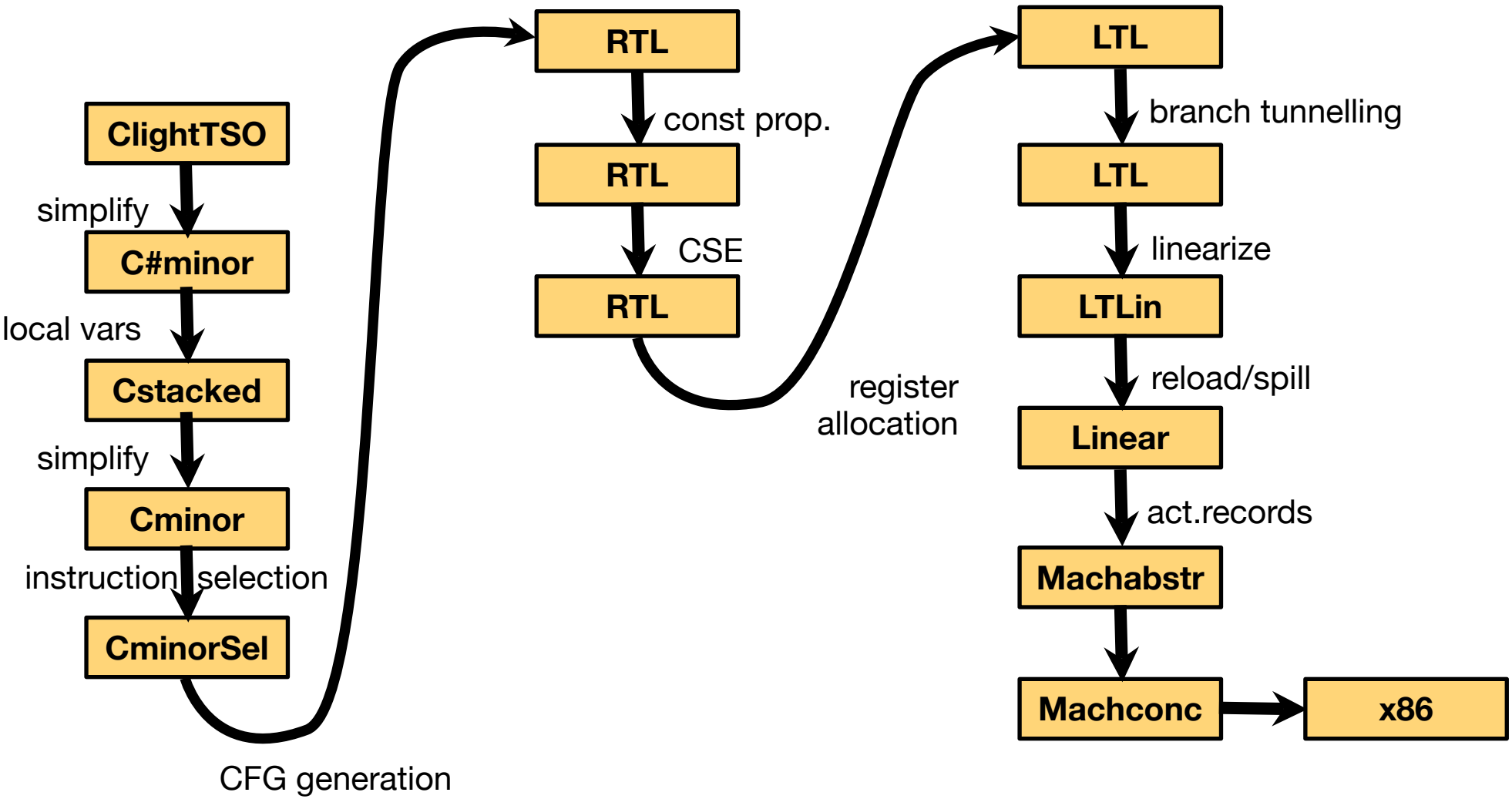
Verifying fence elimination optimisations

Viktor Vafeiadis, MPI-SWS
Francesco Zappa Nardelli, INRIA

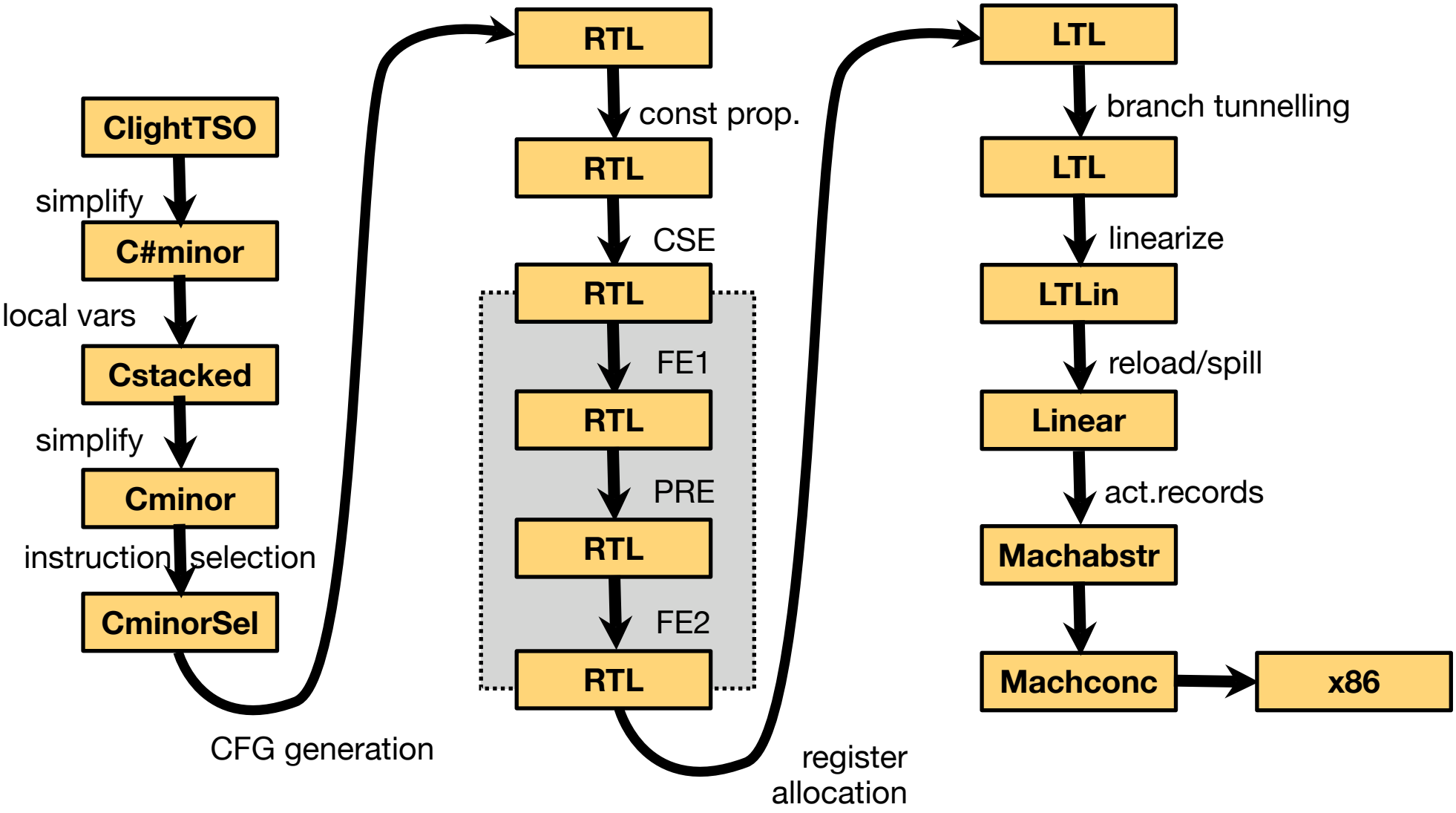
http://www.cl.cam.ac.uk/~pes20/CompCertTSO

CompCertTSO



[POPL 2011]

CompCertTSO + fence optimisations



Language semantics

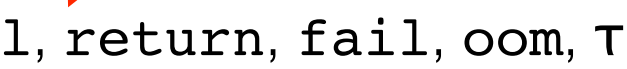
The semantics of all the CompCertTSO languages is defined by:

– a type of programs, prg

– a type of states, $states$

– a set of initial states for each program, $\mathsf{init} \in prg \to \mathbb{P}(states)$

– a transition relation, $\to \in \mathbb{P}(states \times event \times states)$



Traces

– *Infinite sequences* of call & return events;

– *Finite sequences* of call & return events ending with:

end: successful termination,

inftau: infinite execution that stops performing visible events

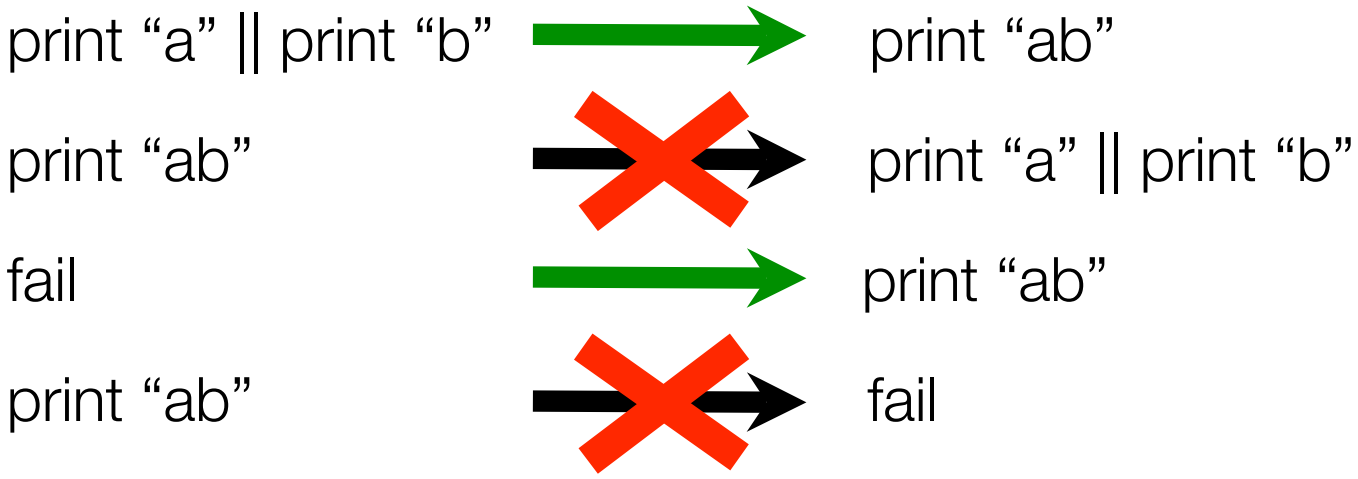
oom: execution runs out of memory

$$
\begin{aligned}
\mathsf{traces}(p) \stackrel{\text{def}}{=} \quad & \{\ell \cdot \mathsf{end} \mid \exists s \in \mathsf{init}(p).\ \exists s'.\ s \stackrel{\ell}{\Rightarrow} s' \wedge s' \not\rightarrow\} \\
& \cup \{\ell \cdot tr \mid \exists s \in \mathsf{init}(p).\ \exists s'.\ s \stackrel{\ell \cdot \mathtt{fail}}{\Longrightarrow} s'\} \\
& \cup \{\ell \cdot \mathsf{inftau} \mid \exists s \in \mathsf{init}(p).\ \exists s'.\ s \stackrel{\ell}{\Rightarrow} s' \wedge \mathsf{inftau}(s')\} \\
& \cup \{\ell \cdot \mathsf{oom} \mid \exists s \in \mathsf{init}(p).\ \exists s'.\ s \stackrel{\ell}{\Rightarrow} s'\} \\
& \cup \{tr \mid \exists s \in \mathsf{init}(p).\ s \text{ can do the infinite trace } tr\}
\end{aligned}
$$

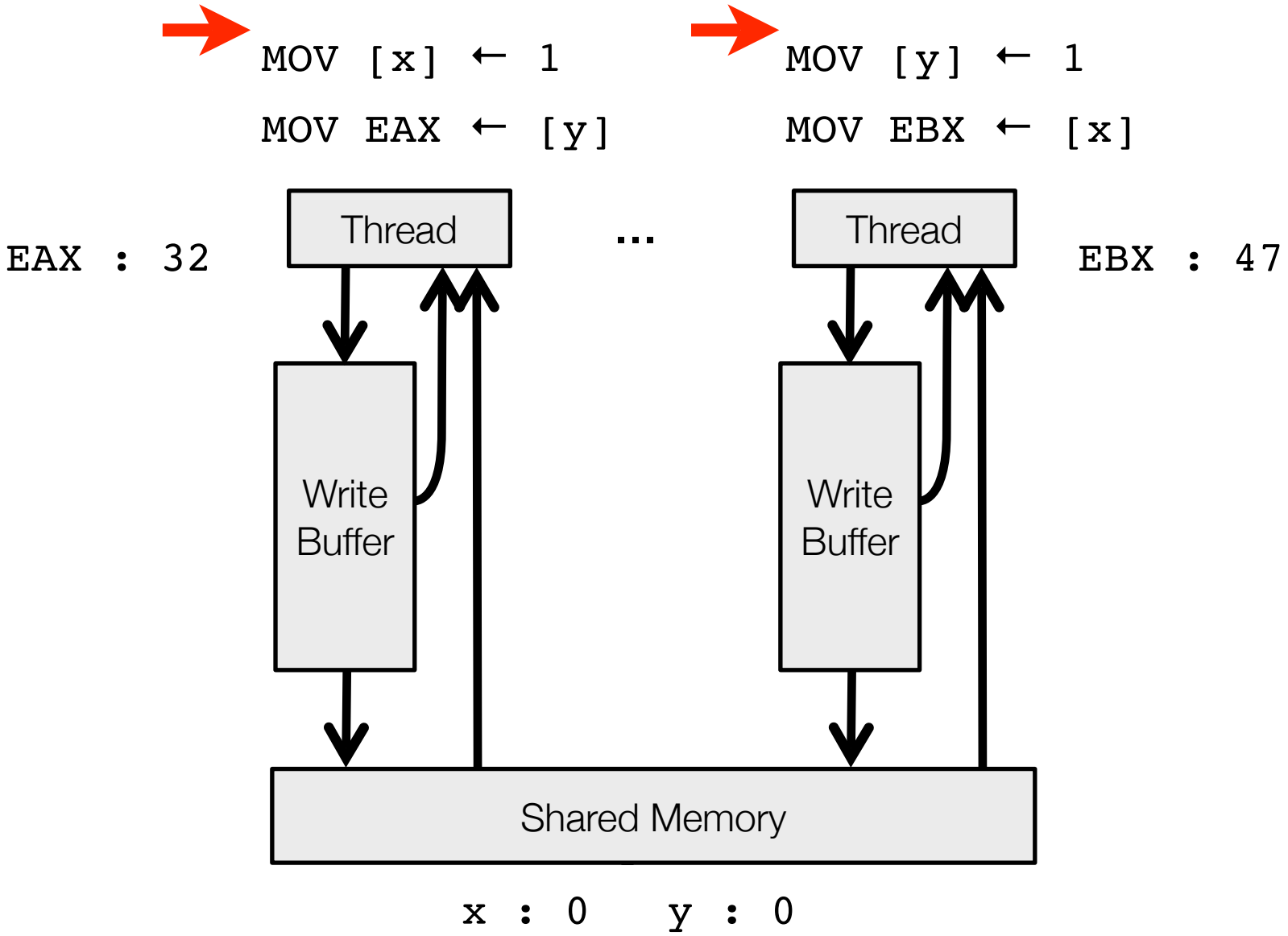
NB: Erroneous computations become undefined after the first error.

Compiler correctness

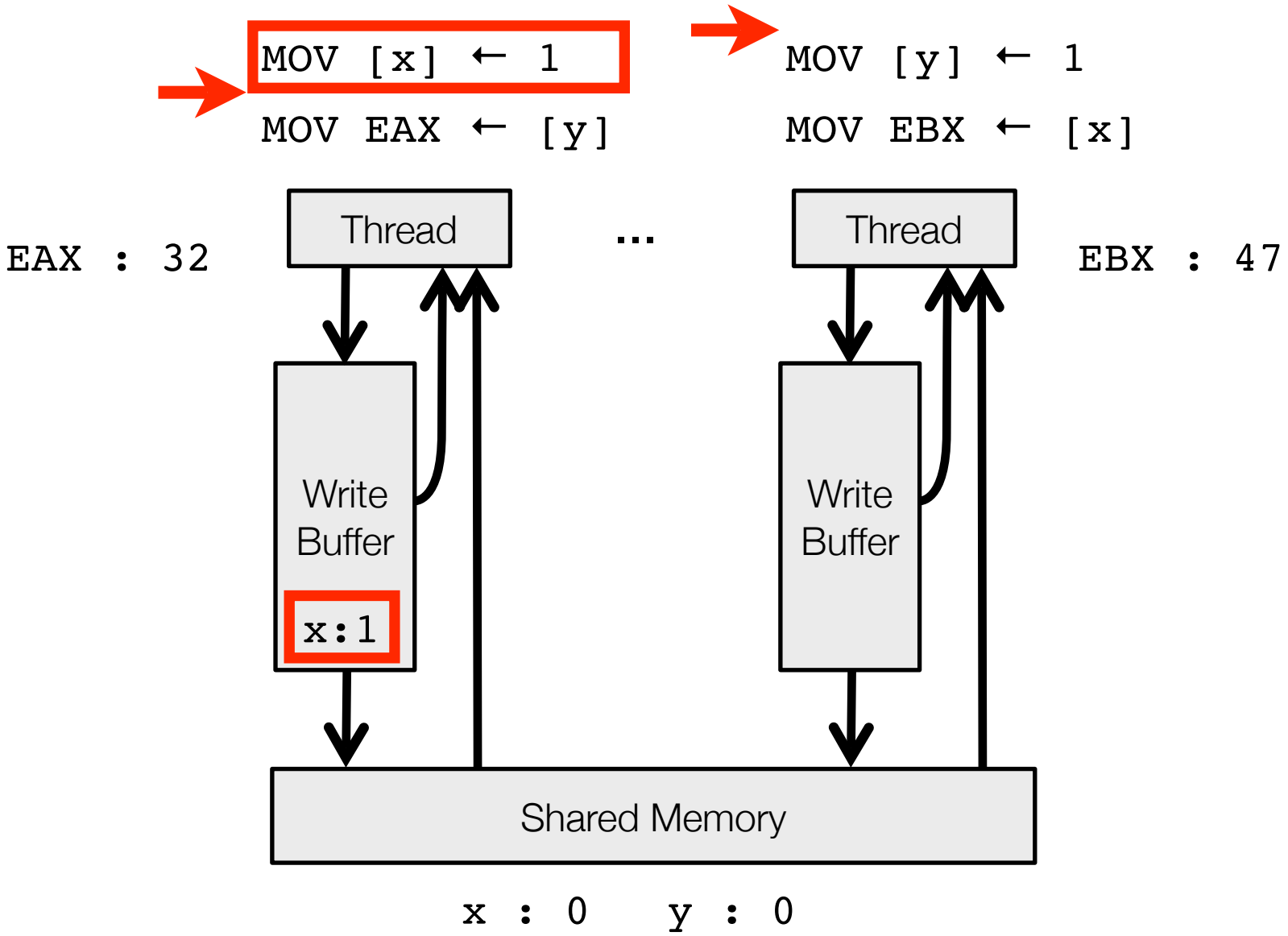
$$\text{traces}(\text{source\_program}) \supseteq \text{traces}(\text{target\_program})$$



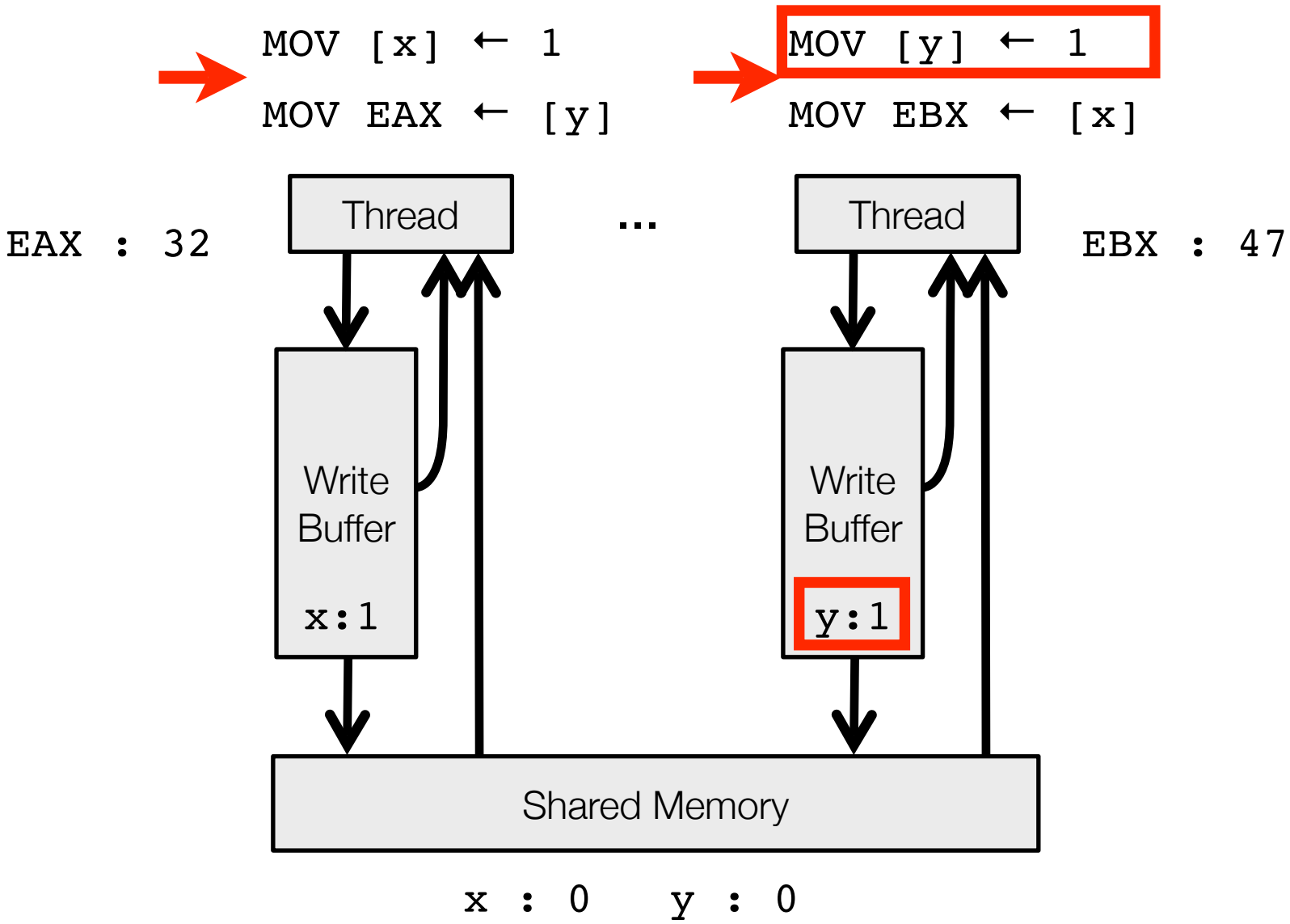
Store buffering



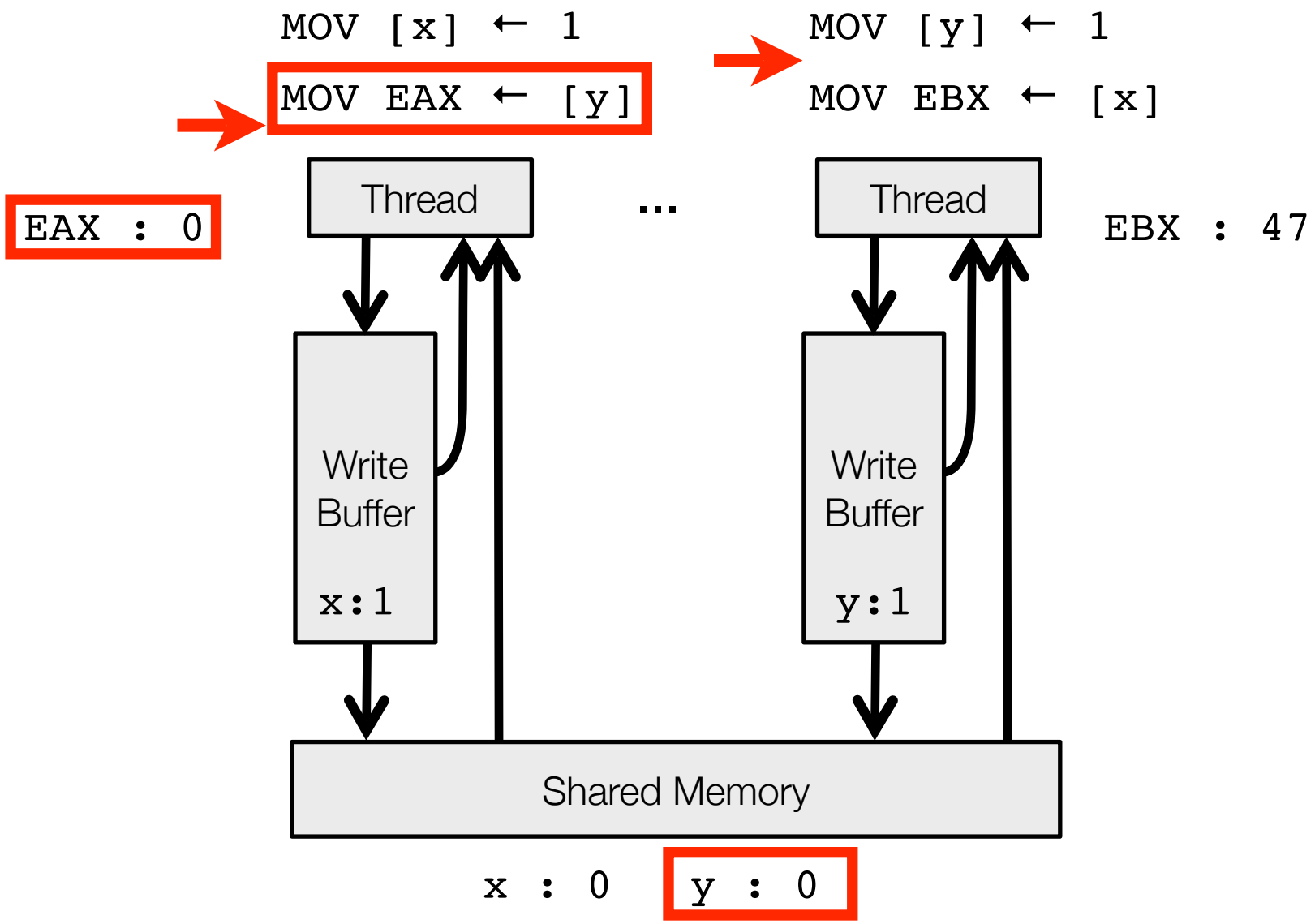
Store buffering



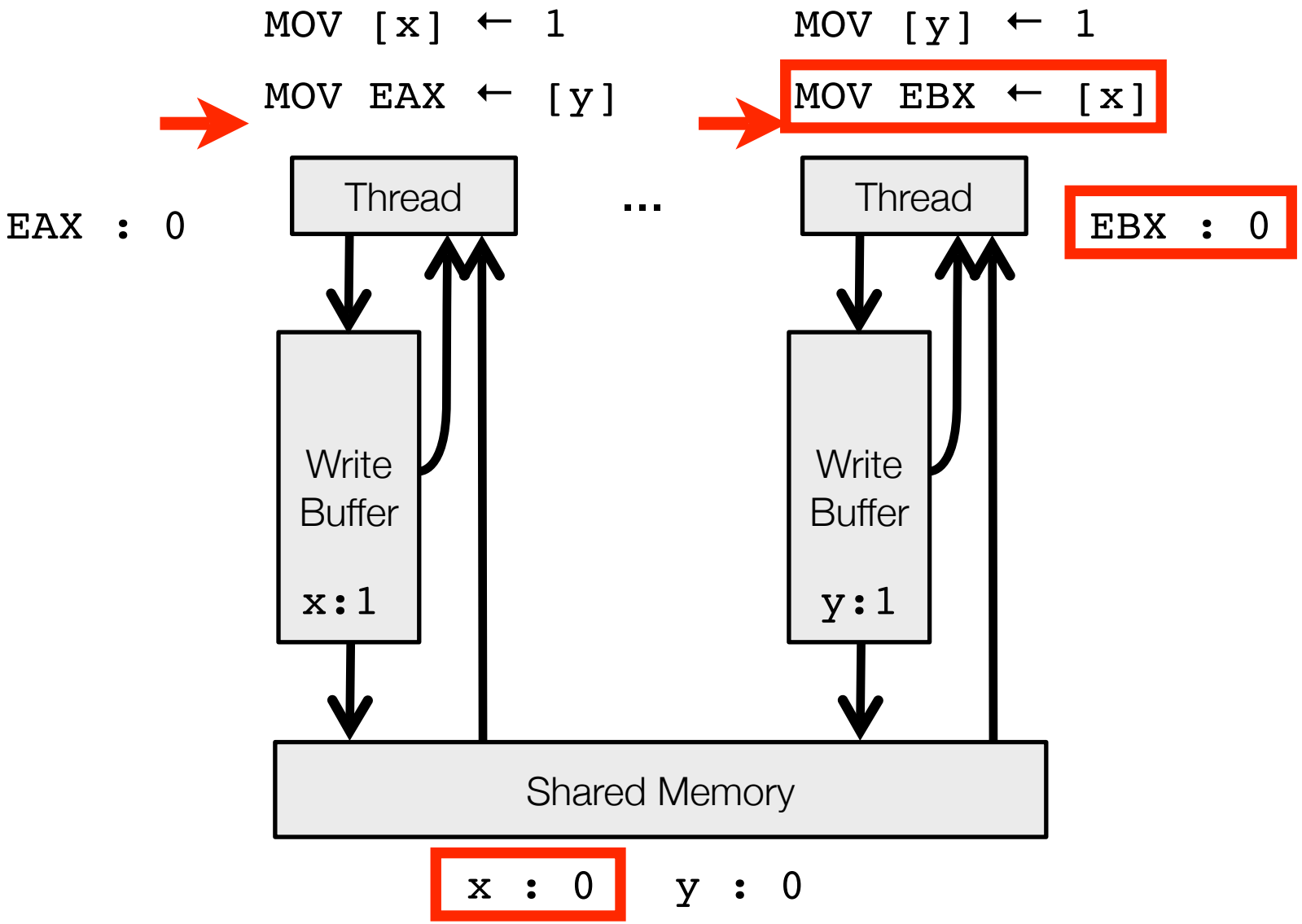
Store buffering



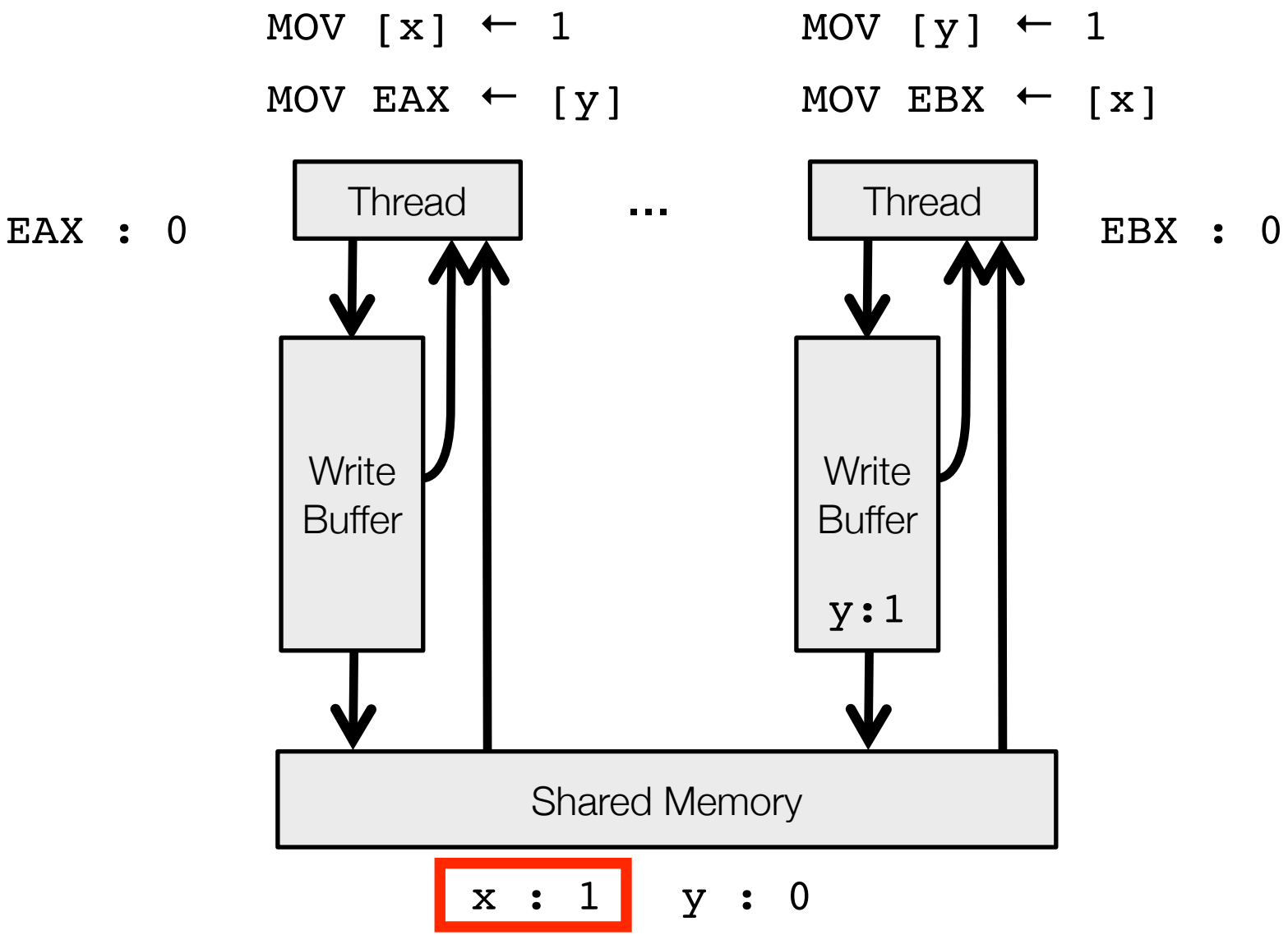
Store buffering



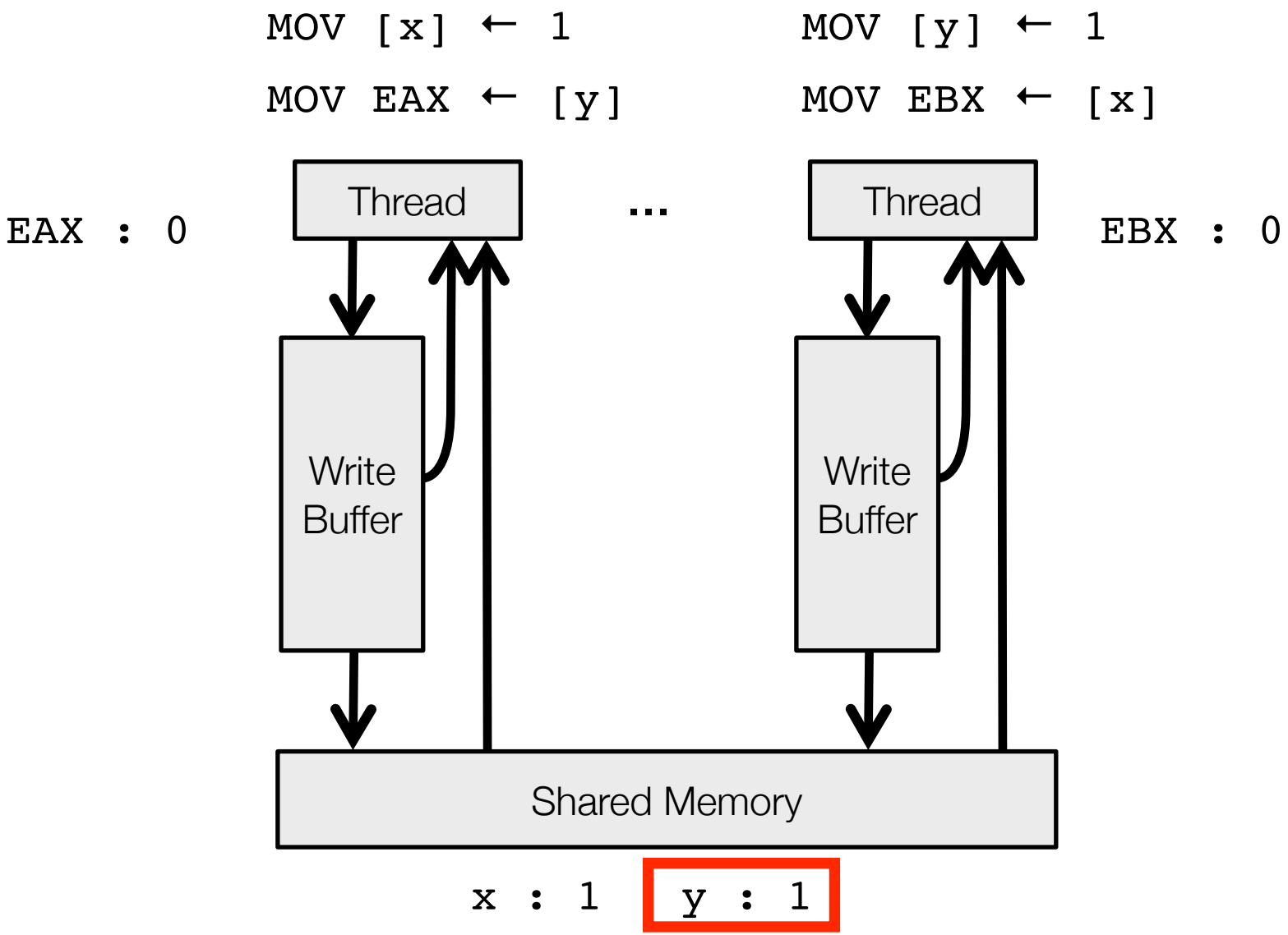
Store buffering



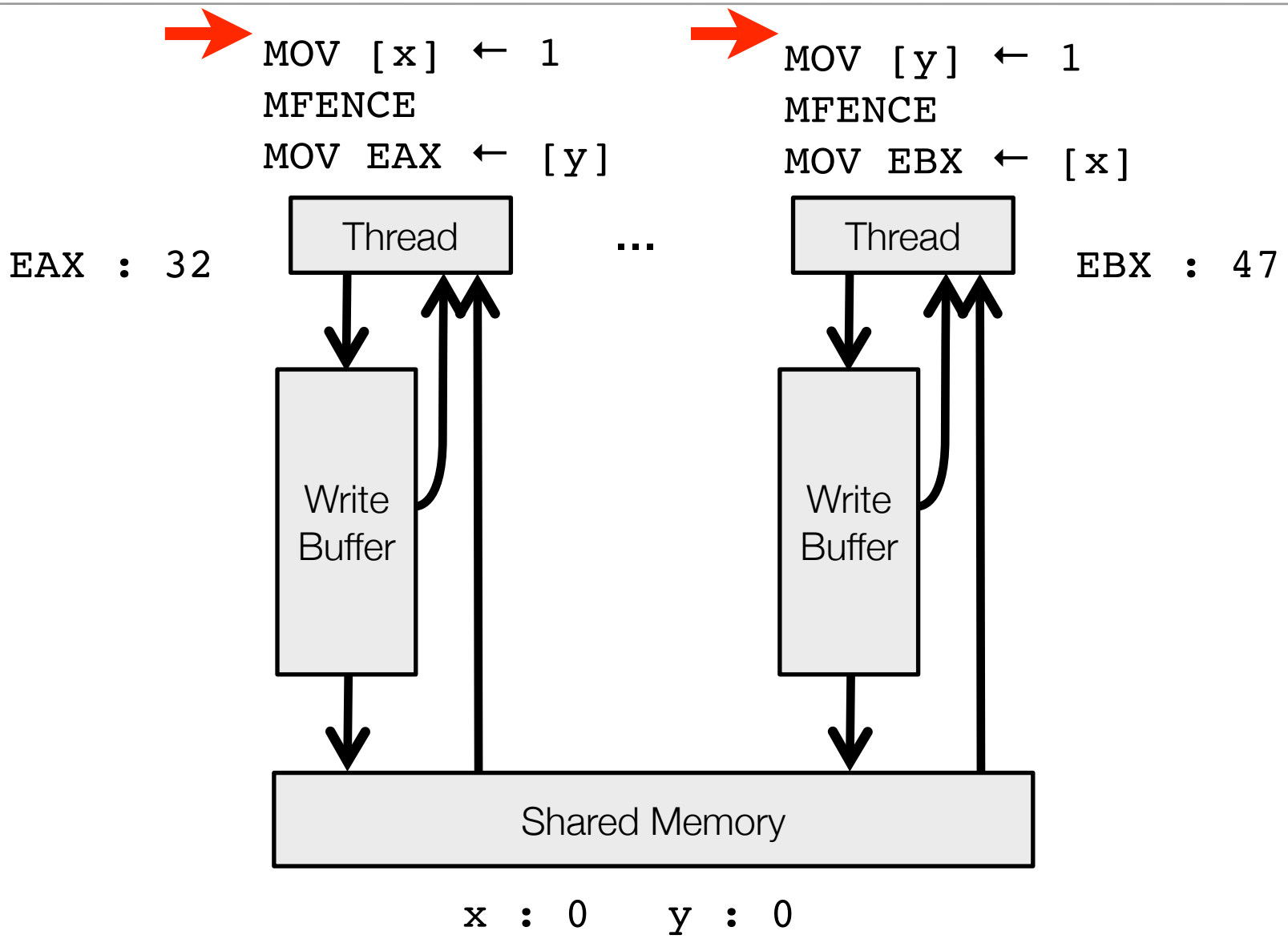
Store buffering



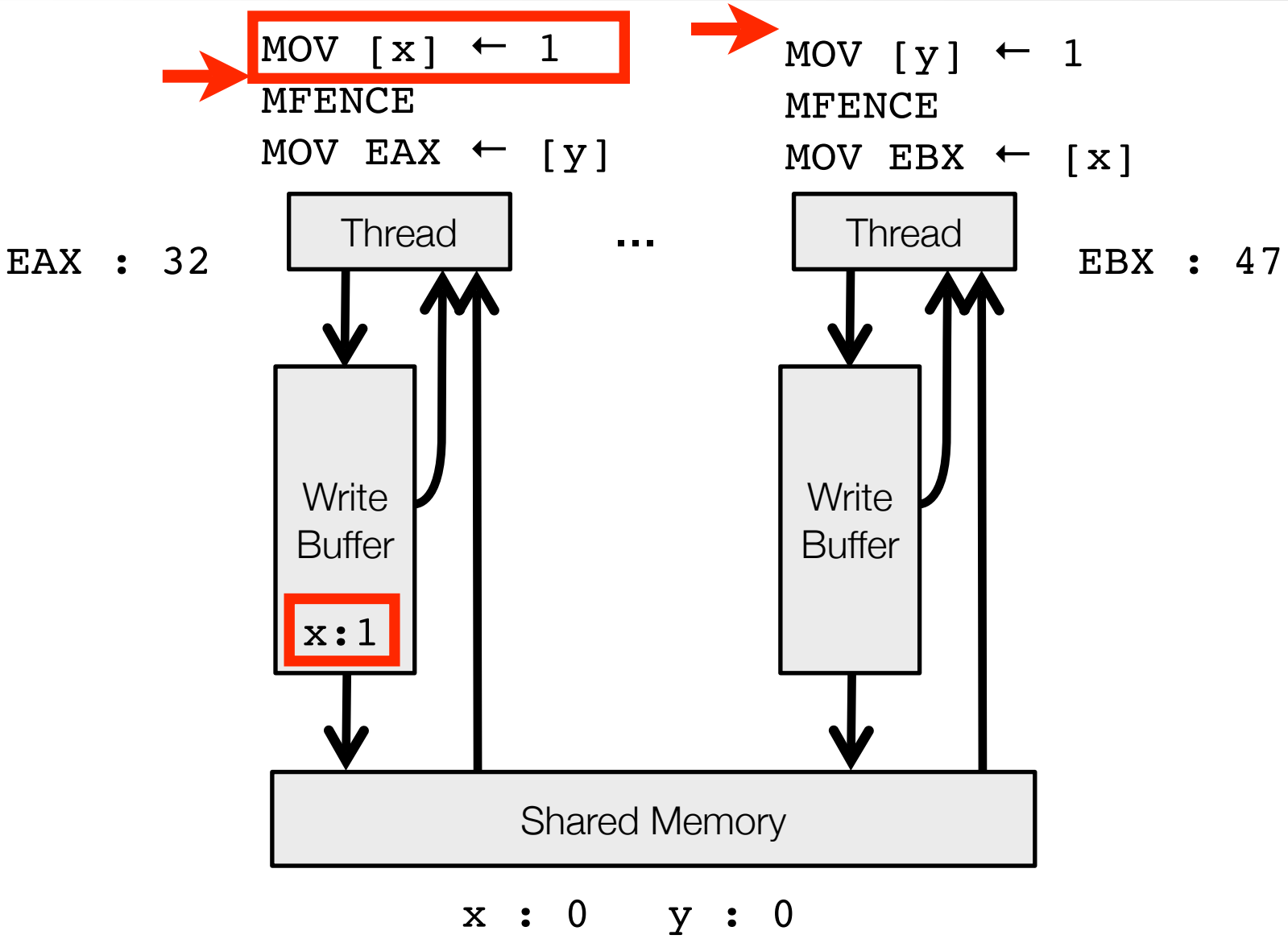
Store buffering



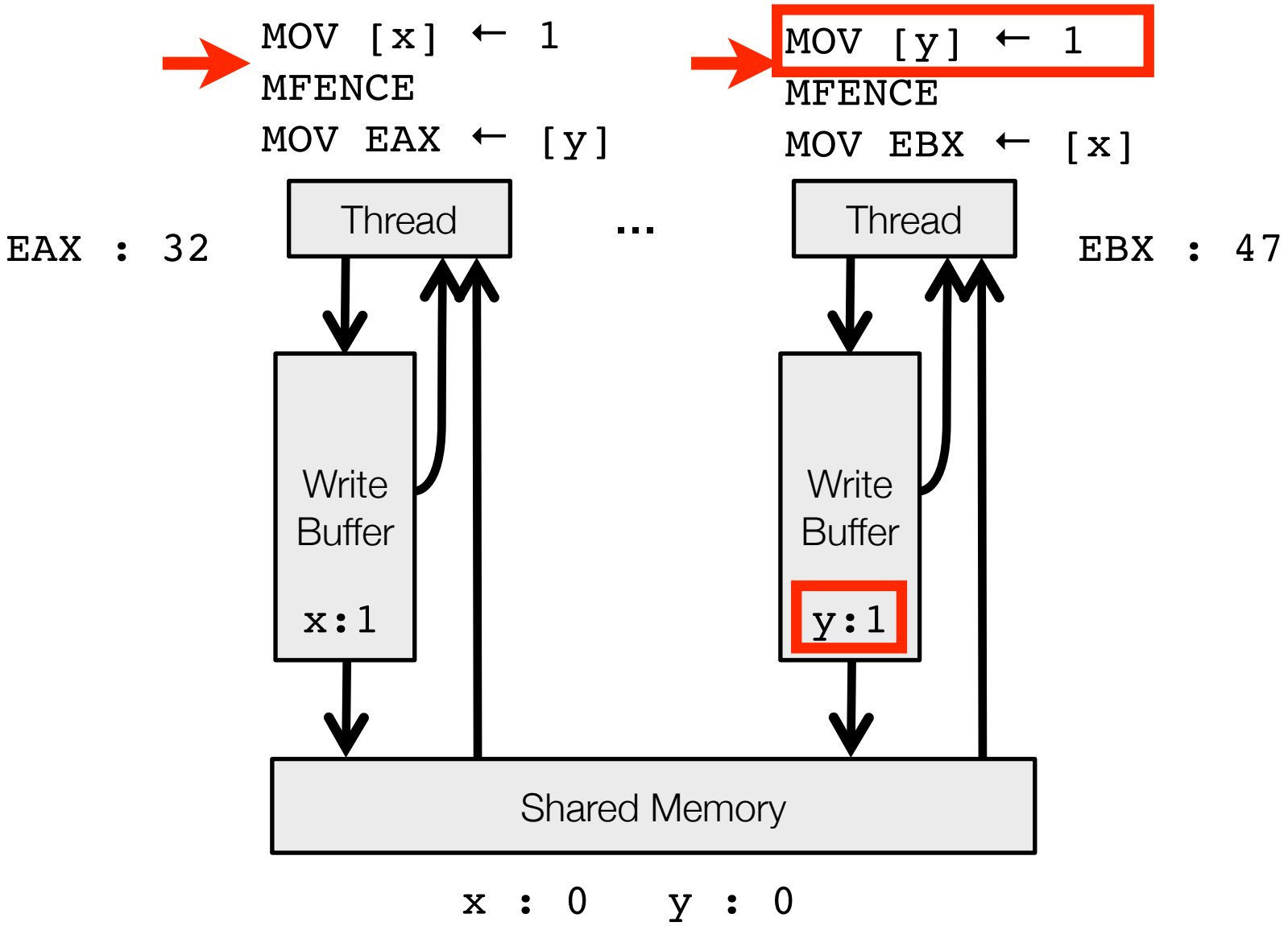
Store buffering + fences



Store buffering + fences

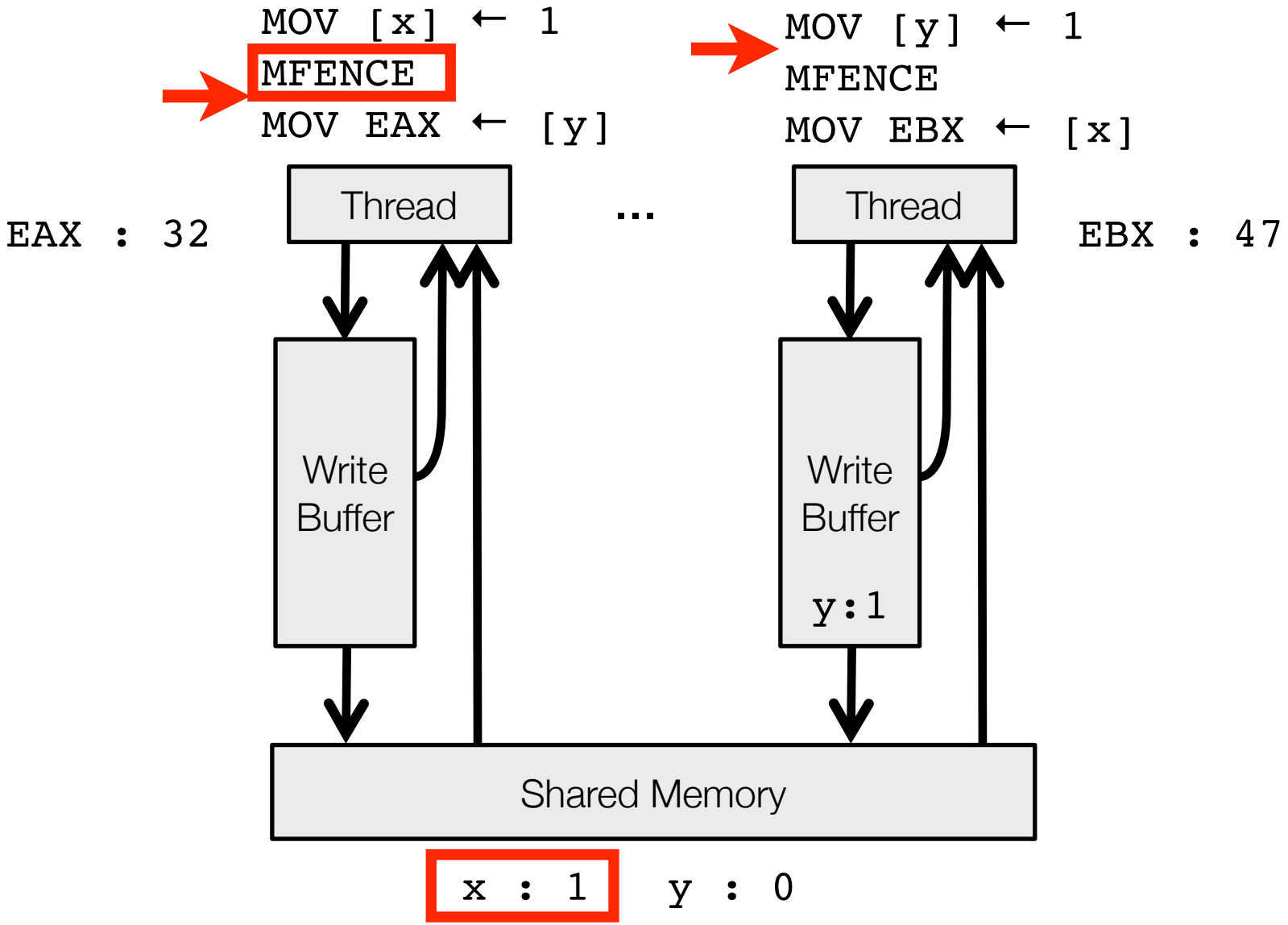


Store buffering + fences



Store buffering + fences

`MFENCE` blocks until the thread buffer is empty



Who inserts fences?

1. The *programmer*, explicitly. Example: Fraser's lockfree-lib:

```
/*
 * II. Memory barriers.
 *  MB():  All preceding memory accesses must commit before any later accesses.
 *
 *  If the compiler does not observe these barriers (but any sane compiler
 *  will!), then VOLATILE should be defined as 'volatile'.
 */
#define MB()  __asm__ __volatile__ ("lock; addl $0,0(%%esp)" : : : "memory")
```

2. The *compiler*, to implement a high-level memory model, e.g. **`SEQ_CST`** C++0x low-level atomics on x86:

Load SEQ_CST: `MFENCE; MOV`

Store SEQ_CST: `MOV; MFENCE`

Fence instructions

1. *Fences are necessary*

   to implement locks & not fully-commutative linearizable objects (e.g., stacks, queues, sets, maps).

   [Attiya et al., POPL 2011]

2. *Fences can be expensive*

Redundant fences (1)

If we have two consecutive fence instructions, we can remove the *latter*:

```
MFENCE
MFENCE
```

```
MFENCE
NOP
```

The *buffer is already empty* when the second fence is executed.

Generalisation:

```
MFENCE
NON-WRITE INSTR
…
NON-WRITE INSTR
MFENCE
```

```
MFENCE
NON-WRITE INSTR
…
NON-WRITE INSTR
NOP
```

FE1

A fence is redundant if it always follows a previous fence or locked instruction in program order, and no memory store instructions are in between.

A *forward* data-flow problem over the boolean domain $\{\bot, \top\}$

Associate to each program point:

$\bot$: along all execution paths there is an atomic instruction *before* the current program point, with no intervening writes;

$\top$: otherwise.

$$
\begin{aligned}
T_1(\texttt{nop}, \mathcal{E}) &= \mathcal{E} \\
T_1(\texttt{op}(op, \vec{r}, r), \mathcal{E}) &= \mathcal{E} \\
T_1(\texttt{load}(\kappa, addr, \vec{r}, r), \mathcal{E}) &= \mathcal{E} \\
T_1(\texttt{store}(\kappa, addr, \vec{r}, src), \mathcal{E}) &= \top \\
T_1(\texttt{call}(sig, ros, args, res), \mathcal{E}) &= \top \\
T_1(\texttt{cond}(cond, args), \mathcal{E}) &= \mathcal{E} \\
T_1(\texttt{return}(optarg), \mathcal{E}) &= \top \\
T_1(\texttt{threadcreate}(optarg), \mathcal{E}) &= \top \\
T_1(\texttt{atomic}(aop, \vec{r}, r), \mathcal{E}) &= \bot \\
T_1(\texttt{fence}, \mathcal{E}) &= \bot
\end{aligned}
$$

$$
\mathcal{FE}_1(n) = \begin{cases} \top & \text{if predecessors}(n) = \emptyset \\ \bigsqcup_{p \in \text{predecessors}(n)} T_1(instr(p), \mathcal{FE}_1(p)) & \text{otherwise} \end{cases}
$$

FE1

A fence is redundant if it always follows a previous fence or locked instruction in program order, and no memory store instructions are in between.

A *forward* data-flow problem over the bo

Assoc

⊥ : alo
is a
cur
no

⊤ : oth

$$T_1(\mathbf{nop}, \mathcal{E}) = \mathcal{E}$$
$$= \mathcal{E}$$
$$= \mathcal{E}$$
$$= \top$$
$$) = \top$$
$$= \mathcal{E}$$
$$= \top$$
$$= \top$$
$$= \bot$$
$$= \bot$$

Implementation:

1. Use CompCert implementation of Kildall algorithm to solve the data-flow equations.
2. Replace **`MFENCE`**s for which the analysis returns ⊥ with **`NOP`** instructions.

$$\mathcal{FE}_1(n) = \begin{cases} \emptyset \\ \bigsqcup_{p \in \text{predecessors}(n)} T_1(instr(p), \mathcal{FE}_1(p)) \quad \text{otherwise} \end{cases}$$

Redundant fences (2)

If we have two consecutive fence instructions, we can remove the *former*:

```
MFENCE
MFENCE
```

```
NOP
MFENCE
```

Intuition: the visible effects initially published by the former fence, are now published by the latter, and nobody can tell the difference.

Generalisation:

```
MFENCE
INSTRUCTION 1
…
INSTRUCTION n
MFENCE
```


```
NOP
INSTRUCTION 1
…
INSTRUCTION n
MFENCE
```

Redundant fences (2)

If there are reads in between the fences…

`[x]=[y]=0`

Thread 0	Thread 1

`MOV [x] ← 1`
`MFENCE`
`MOV EAX ← [y]`
`MFENCE`
	`MOV [y] ← 1`
	`MFENCE`
	`MOV EBX ← [x]`

`EAX = EBX = 0`
forbidden

but

`[x]=[y]=0`

Thread 0	Thread 1

`MOV [x] ← 1`
`NOP`
`MOV EAX ← [y]`
`MFENCE`
	`MOV [y] ← 1`
	`MFENCE`
	`MOV EBX ← [x]`

`EAX = EBX = 0`
allowed

Redundant fences (2)

If there are reads in between the fences…

[x]=[y]=0

Thread 0	Thread 1
MOV [x] ← 1 **MFENCE** MOV EAX ← [y]	MOV [y] ← 1 MFENCE

EAX = EBX = 0
forbidden

but

If there are reads in between, the optimisation is unsound.

[x]=[y]=0

Thread 0	Thread 1
MOV [x] ← 1 **NOP** MOV EAX ← [y] MFENCE	MOV [y] ← 1 MFENCE MOV EBX ← [x]

EAX = EBX = 0
allowed

Redundant fences (2)

Swapping a **`STORE`** and a **`MFENCE`** is sound:

1. transformed program's behaviours ⊆ source program's behaviours

(source program might leave pending write in its buffer)

2. There is the new intermediate state if the buffer was initially non-empty, but this intermediate state *is not observable.*

(a local read is needed to access the local buffer)

Intuition: Iterate this swapping...

FE2

A fence is redundant if it always precedes a later fence or locked instruction in program order, and no memory read instructions are in between.

A *backward* data-flow problem over the boolean domain $\{\bot, \top\}$

Associate to each program point:

$\bot$: along all execution paths there is an atomic instruction *after* the current program point, with no intervening reads;

$\top$: otherwise.

$$
\begin{aligned}
T_2(\texttt{nop}, \mathcal{E}) &= \mathcal{E} \\
T_2(\texttt{op}(op, \vec{r}, r), \mathcal{E}) &= \mathcal{E} \\
T_2(\texttt{load}(\kappa, addr, \vec{r}, r), \mathcal{E}) &= \top \\
T_2(\texttt{store}(\kappa, addr, \vec{r}, src), \mathcal{E}) &= \mathcal{E} \\
T_2(\texttt{call}(sig, ros, args, res), \mathcal{E}) &= \top \\
T_2(\texttt{cond}(cond, args), \mathcal{E}) &= \mathcal{E} \\
T_2(\texttt{return}(optarg), \mathcal{E}) &= \top \\
T_2(\texttt{threadcreate}(optarg), \mathcal{E}) &= \top \\
T_2(\texttt{atomic}(aop, \vec{r}, r), \mathcal{E}) &= \bot \\
T_2(\texttt{fence}, \mathcal{E}) &= \bot
\end{aligned}
$$

$$
\mathcal{FE}_2(n) = \begin{cases} \top & \text{if successors}(n) = \emptyset \\ \bigsqcup_{s \in \text{successors}(n)} T_2(instr(s), \mathcal{FE}_2(s)) & \text{otherwise} \end{cases}
$$

Informal correctness argument

Intuition: FE2 can be thought as iterating

```
MFENCE; STORE      →    STORE; MFENCE
MFENCE; non-mem    →    non-mem; MFENCE
```

and then applying

```
MFENCE; MFENCE     →    NOP; MFENCE
```

This argument works for *finite traces*, but not for *infinite traces* as the later fence might never be executed:

```
MFENCE;            →    NOP;
STORE;                  STORE;
WHILE(1);               WHILE(1);
MFENCE                  MFENCE
```

Basic simulations

A pair of relations

$$\sim \in \mathbb{P}(src.states \times tgt.states) \qquad > \in \mathbb{P}(tgt.states \times tgt.states)$$

is a *basic simulation* for $\mathsf{compile} : src.prg \to tgt.prg$ if:

$$sim\_init : \forall p\, p'.\ \mathsf{compile}(p) = p' \implies \forall t \in \mathsf{init}(p').\ \exists s \in \mathsf{init}(p).\ s \sim t$$

$$sim\_end : \forall s\, t.\ s \sim t \wedge t \not\to \_ \implies s \not\to \_$$

$$sim\_step : \forall s\, t\, t'\, ev.\ s \sim t \wedge t \xrightarrow{ev} t' \wedge ev \neq \mathsf{oom} \implies$$

$$\begin{array}{ll} (s \xrightarrow{\tau}{}^{*} \xrightarrow{\mathtt{fail}} \_) & \text{— } s \textit{ reaches a failure} \\ \vee\ (\exists s'.\ s \xrightarrow{\tau}{}^{*} \xrightarrow{ev} s' \wedge s' \sim t') & \text{— } s \textit{ does matching step sequence} \\ \vee\ (ev = \tau \wedge t > t' \wedge s \sim t')\,. & \text{— } s \textit{ stutters (only allowed if } t > t') \end{array}$$

Exhibiting a basic simulation implies:

$$\mathsf{traces}(\mathsf{compile}(p)) \setminus \{t \cdot \mathsf{inftau} \mid t \text{ trace}\} \subseteq \mathsf{traces}(p)$$

“simulation can stutter forever”

Usual approach: measured simulations

Definition 2 (Measured sim.). *A* measured simulation *is any basic simulation* $(\sim, >)$ *such that* $>$ *is well-founded.*

Theorem 1. *If there exists a measured simulation for the compilation function* compile, *then for all programs* p, $\mathsf{traces}(\mathsf{compile}(p)) \subseteq \mathsf{traces}(p)$.

Simulation for FE2

$s \equiv_i t$ iff thread i of s and t have identical pc, local states and buffers

$s \rightsquigarrow_i s'$ iff thread i of s can execute zero or more `NOP`, `OP`, `STORE` and `MFENCE` instructions and end in the state s'

$s \sim t$ iff

- t's CFG is the optimised version of s's CFG; and
- s and t have identical memories; and
- $\forall$ thread i, either $s \equiv_i t$ or

  the analysis for i's pc returned $\bot$ and $\exists s'$, $s \rightsquigarrow_i s'$ and $s' \equiv_i t$

  "s is some instructions behind and can catch up"

Stutter condition:

$t > t'$ iff $t \rightarrow t'$ by a thread executing a `NOP`, `OP`, `STORE` or `MFENCE` (and t's buffer being non-empty)

Simulation for FE2

$s \equiv_i t$ iff thread i of s and t have identical pc, local states and buffers

$s \rightsquigarrow_i s'$ iff th… d

`MFE`…

$s \sim t$ iff

– t's CFG…

– s and t…

– ∀ thread…

But if (1) all threads have non-empty buffers, and
(2) are stuck executing infinite loops, and
(3) no writes are ever propagated to memory,
then we can stutter forever.

(i.e., $>$ is not well-founded.)

the analysis for i's pc returned $\bot$ and $\exists s'$, $s \rightsquigarrow_i s'$ and $s' \equiv_i t$

"s is some instructions behind and can catch up"

Stutter condition:

$t > t'$ iff $t \rightarrow t'$ by a thread executing a `NOP`, `OP`, `STORE` or `MFENCE`
(and t's buffer being non-empty)

Simulation for FE2

$s \equiv_i t$ iff thread i of s and t have identical pc, local states and buffers

$s \rightsquigarrow_i s'$ iff th… d

MFE…

$s \sim t$ iff

– t's CFG…

– s and t…

– ∀ thread…

(i…

t … $\equiv_i t$

Stutter conditio…

$t > t'$ iff $t \rightarrow$…

(and t's buffer being non-empty)

But if (1) all threads have non-empty buffers, and
(2) are stuck executing infinite loops, and
(3) no writes are ever propagated to memory,
then we can stutter forever.

Solution 1: Assume this case never arises (*fairness*)

Solution 2: Do a case split.
— If this case does not arise, we are done.
— If it does, use a different (weaker) simulation to construct an infinite trace for the source

Weaktau simulation

Definition 3 (Weaktau sim.). *A* weaktau simulation *consists of a basic simulation* $(\sim, >)$ *with and an additional relation between source and target states,* $\simeq \in \mathbb{P}(src.states \times tgt.states)$ *satisfying the following properties:*

$$sim\_weaken : \forall s, t.\ s \sim t \implies s \simeq t$$

$$sim\_wstep : \forall s\, t\, t'.\ s \simeq t \wedge t \xrightarrow{\tau} t' \wedge t > t' \implies$$

$$(s \xrightarrow{\tau}{}^{*} \xrightarrow{\texttt{fail}} \_) \qquad \text{— } s \textit{ reaches a failure}$$

$$\vee\ (\exists s'.\ s \xrightarrow{\tau}{}^{*} \xrightarrow{\tau} s' \wedge s' \simeq t') \qquad \text{— } s \textit{ does a matching step sequence.}$$

Theorem 2. *If there exists a weaktau-simulation* $(\sim, >, \simeq)$ *for the compilation function* compile, *then for all programs* p, $\mathsf{traces}(\mathsf{compile}(p)) \subseteq \mathsf{traces}(p)$.

Remarks:

— Once the simulation game moves from ~ to ≃, stuttering is forbidden;

— Can view difference between ~ and ≃ as a boolean prophecy variable.

Weaktau simulation for FE2

$s \sim t$, $t > t'$ as before.

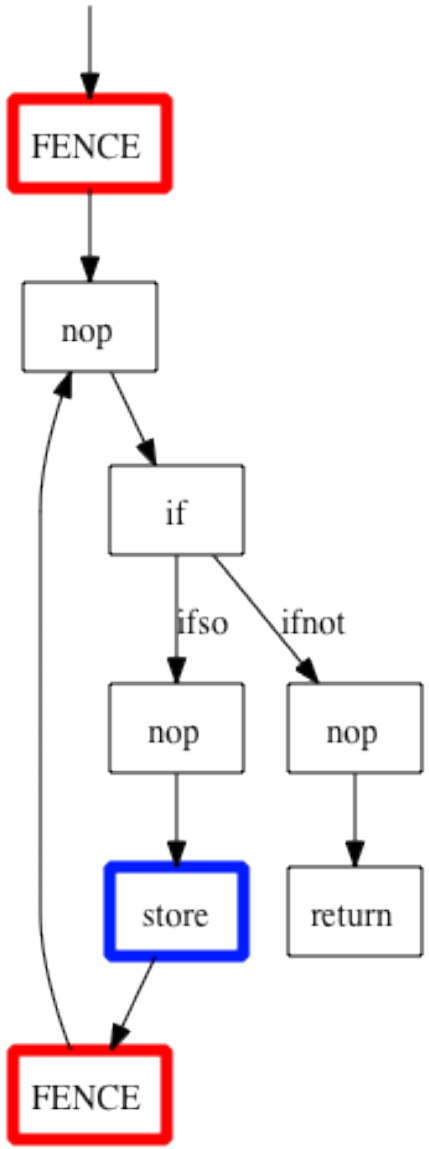
$s \simeq t$ iff

– t's CFG is the optimised version of s's CFG; and

– $\forall i, \exists s'$ s.t. $s \rightsquigarrow_i s' \equiv_i t$.

(i.e., same as $s \sim t$ except that the memories memories are unrelated.)

A closer look at the RTL

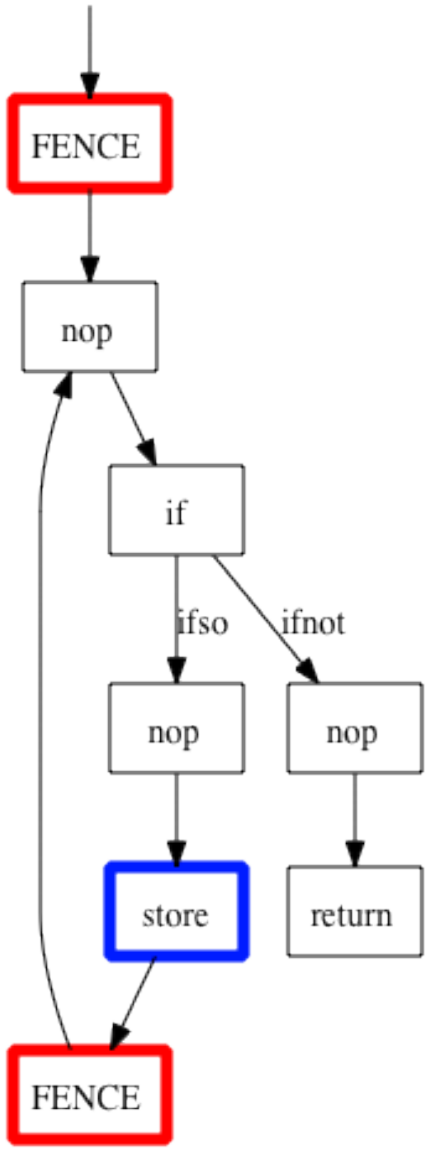


Patterns like that on the left are common.

FE1 and FE2 do not optimise these patterns.

It would be nice to hoist those fences out of the loop.

A closer look at the RTL



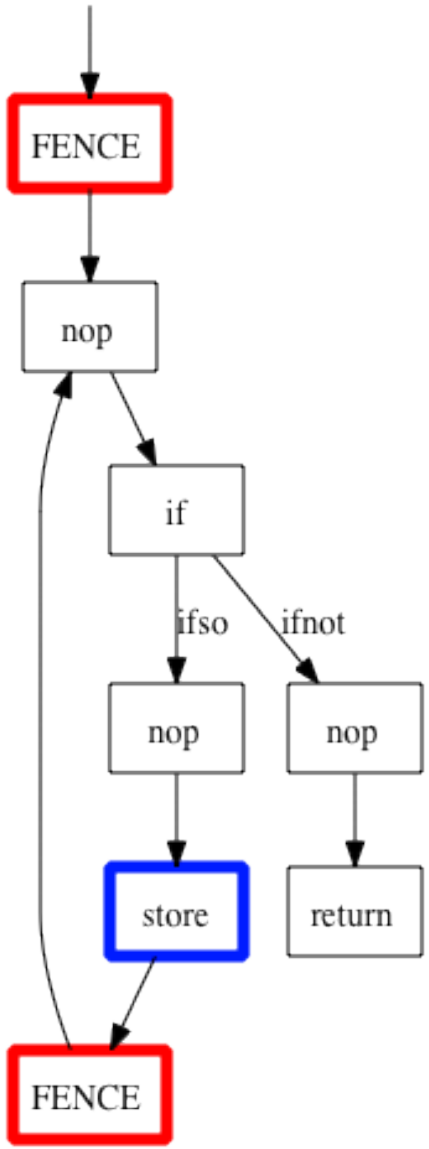
Patterns like that on the left are common.

FE1 and FE2 do not optimise these patterns.

It would be nice to [illegible] out of the loop.

Do you perform PRE?

A closer look at the RTL



Patterns like that on the left are common.

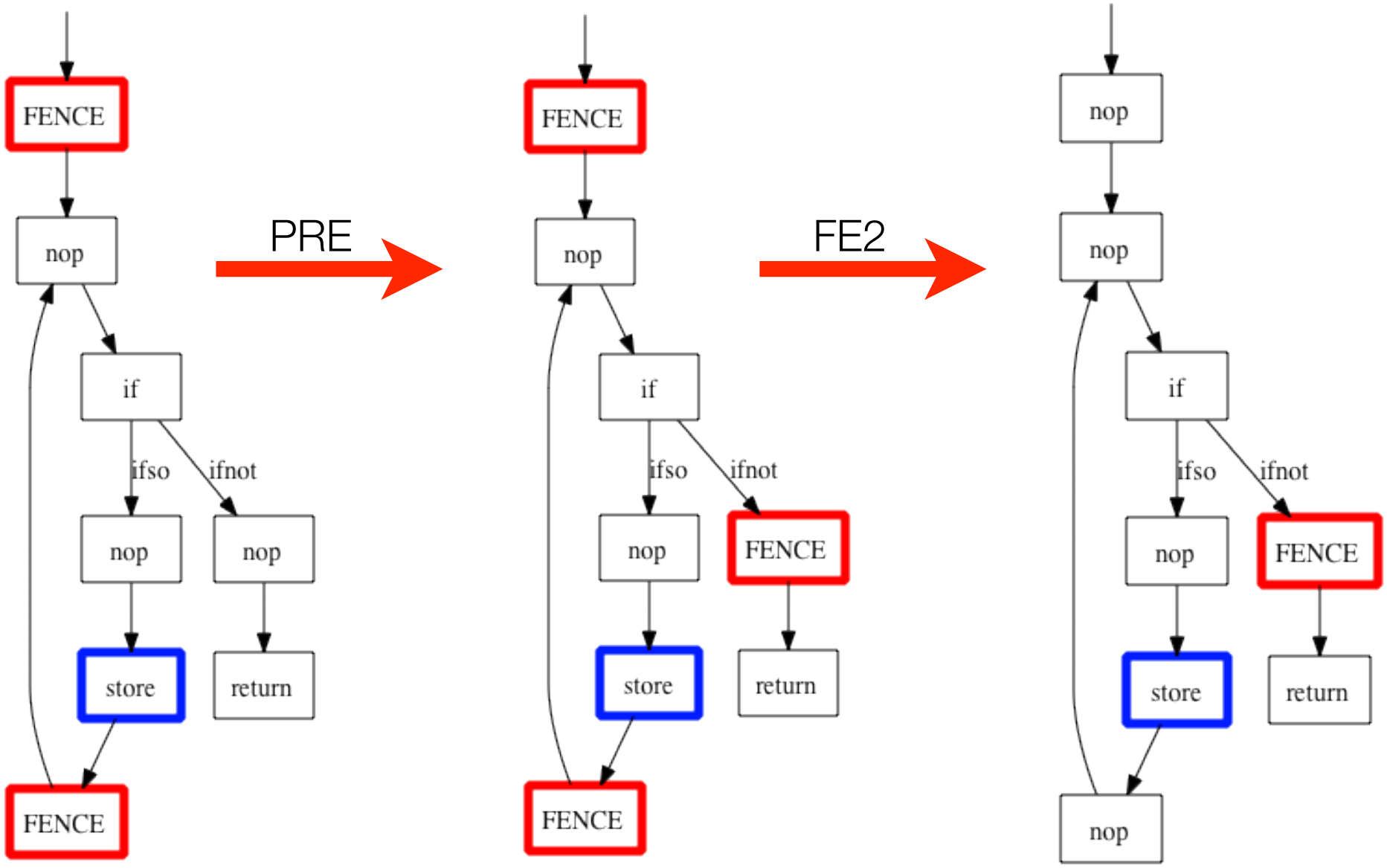
FE1 and FE2 do not optimise these patterns.

It would be nice to [illegible] out of the loop.

Do you perform PRE?

...adding a fence is always safe...

Partial redundancy elimination



Conclusion

Summary

— Two simple fence elimination optimisations under TSO

— Integrated with CompCertTSO

— New proof technique: weaktau simulation

Possible future directions:

— Perform inter-procedural analysis

— More advanced optimisations (e.g., fence placement optimisations)

— More relaxed memory models (e.g., Power or C++)