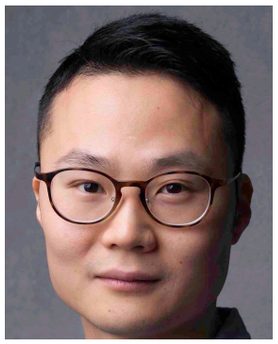
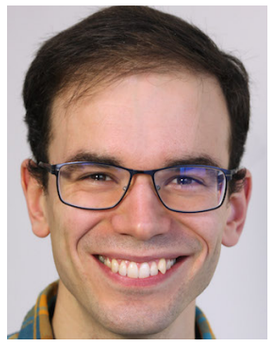
THE SEMANTICS OF TRANSACTIONS AND WEAK MEMORY IN X86, POWER, ARM, AND C++



Nathan Chong
Arm Ltd.

Tyler Sorensen
Princeton University



John Wickerson
Imperial College London

USENIX Annual Technical Conference, 11 July 2019

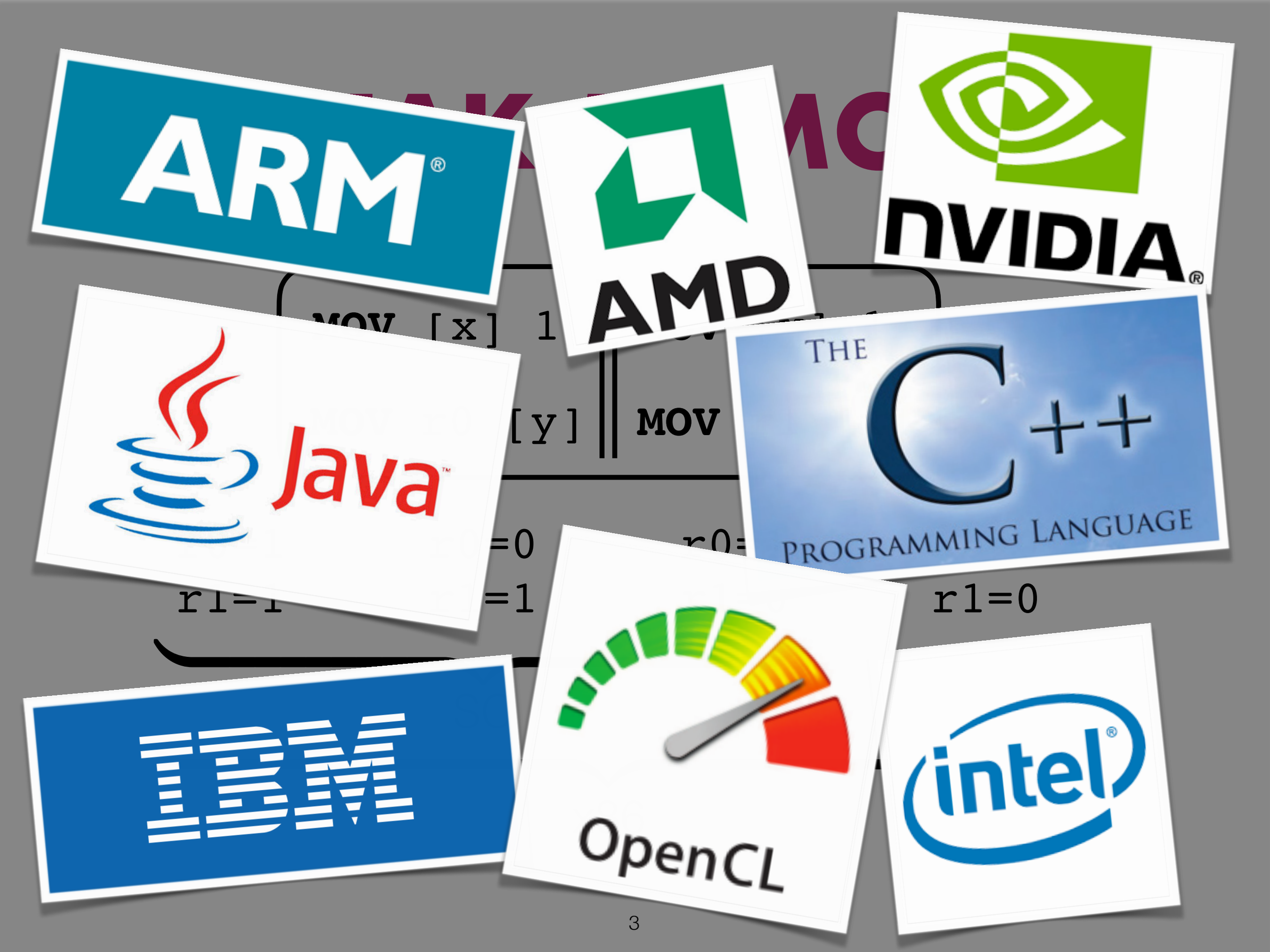
WEAK MEMORY

MOV [x] 1	**MOV** [y] 1
MOV r0 [y]	**MOV** r1 [x]

r0=1 r1=1 | r0=0 r1=1 | r0=1 r1=0 | r0=0 r1=0

SC

x86


ARM
AMD
NVIDIA
Java
THE C++ PROGRAMMING LANGUAGE
IBM
OpenCL
intel
MOV [x] 1
[y] MOV
=0
=1
r1=0

WEAK MEMORY IS HARD!

- x86 proved tricky to formalise correctly *[Sarkar et al., POPL'09; Owens et al., TPHOLs'09]*

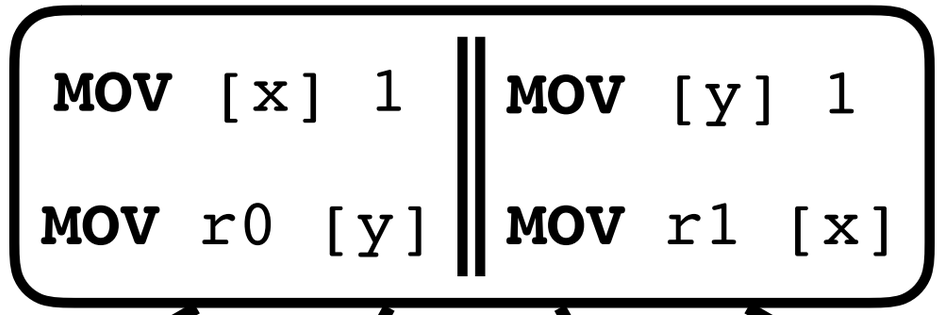
- Bug found in deployed "Power 5" processors *[Alglave et al., CAV'10]*

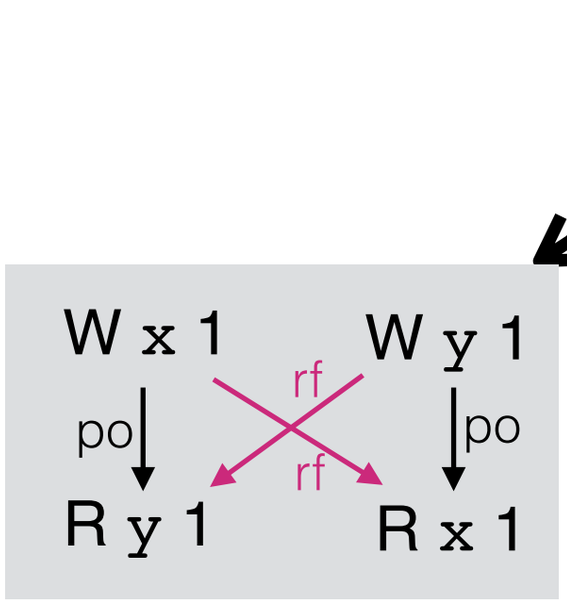
- C++ specification did not guarantee its own key property *[Batty et al., POPL'11]*

- Routine compiler optimisations are invalid under Java and C++ memory models *[Sevcik, PLDI'11; Vafeiadis et al. POPL'15]*

- Behaviour of NVIDIA graphics processors contradicted NVIDIA's programming guide *[Alglave et al., ASPLOS'15]*

MODELLING WEAK MEMORY

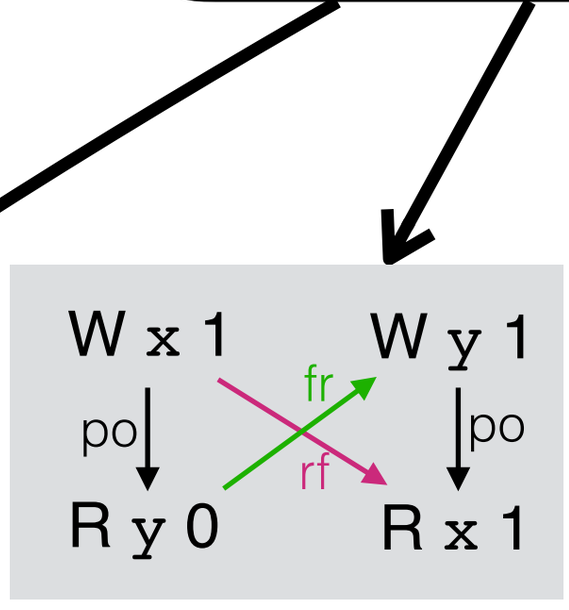




r0=1 r1=1

SC: ✓

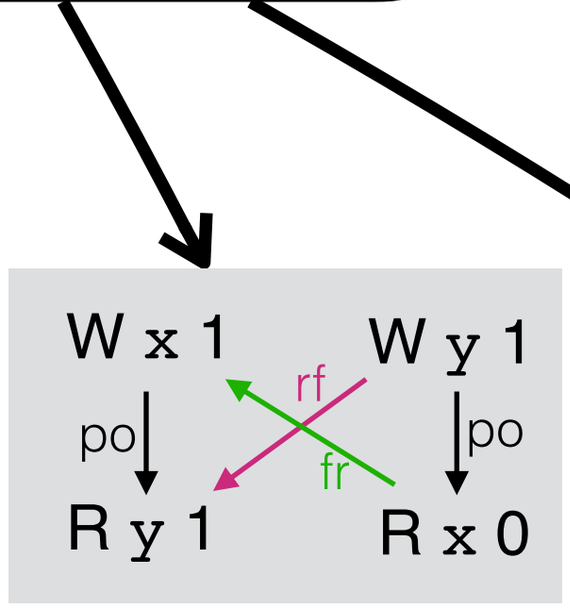
x86: ✓



r0=0 r1=1

SC: ✓

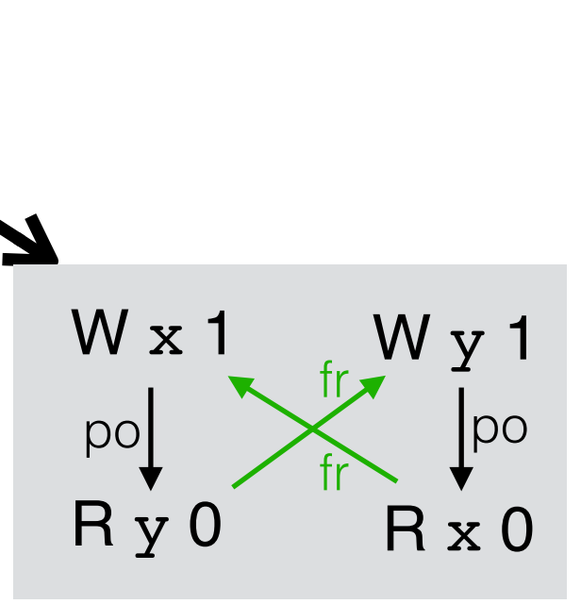
x86: ✓



r0=1 r1=0

SC: ✓

x86: ✓



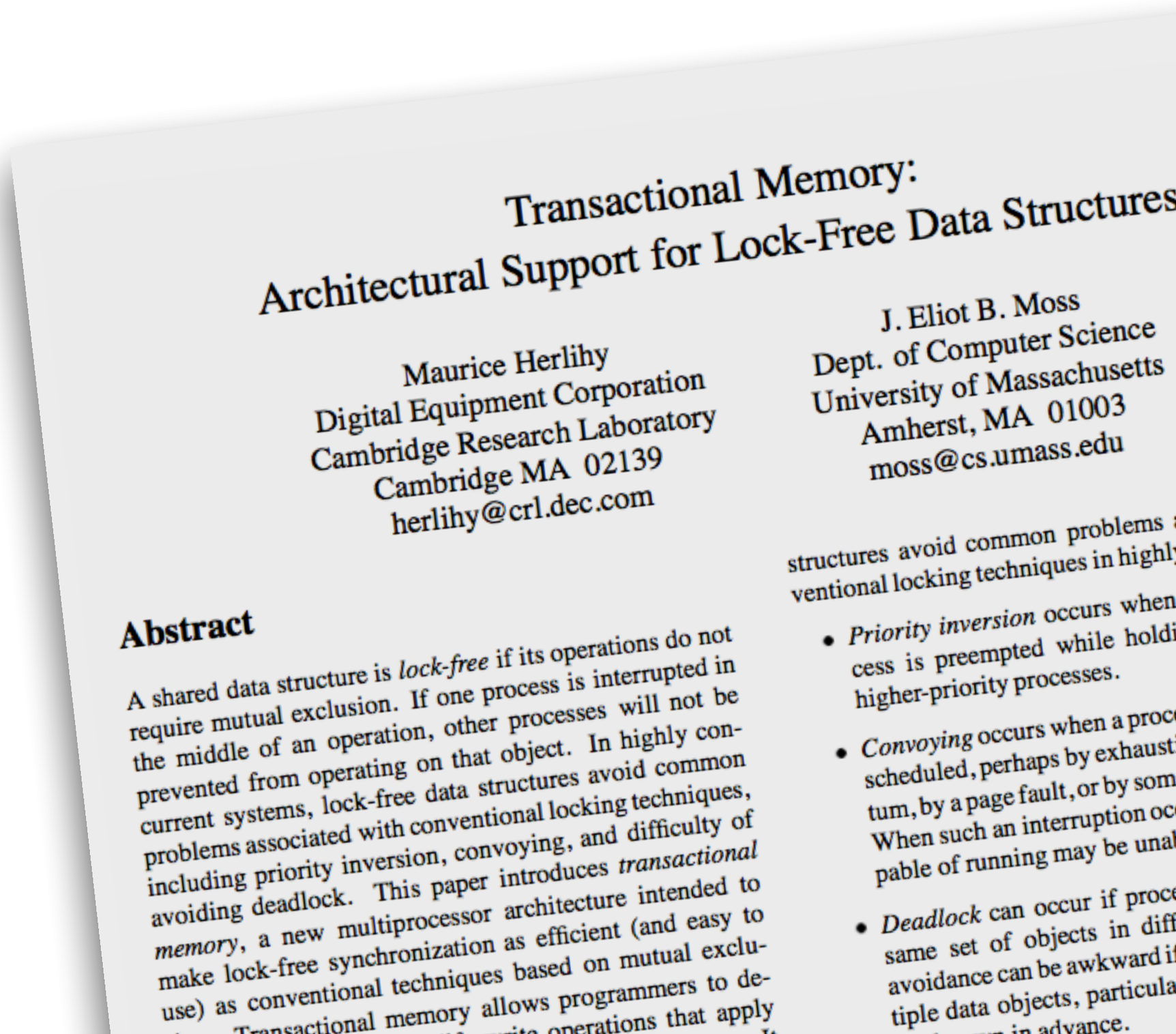
r0=0 r1=0

SC: ✗

x86: ✓

TRANSACTIONAL MEMORY

- X86:
```
XBEGIN
...
XEND
```
- Power:
```
tbegin
...
tend
```
- ARM:
```
tstart
...
tcommit
```
- C++:
```
atomic {
    ...
}
```



Transactional Memory:
Architectural Support for Lock-Free Data Structures

Maurice Herlihy
Digital Equipment Corporation
Cambridge Research Laboratory
Cambridge MA 02139
herlihy@crl.dec.com

J. Eliot B. Moss
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
moss@cs.umass.edu

Abstract

A shared data structure is *lock-free* if its operations do not require mutual exclusion. If one process is interrupted in the middle of an operation, other processes will not be prevented from operating on that object. In highly concurrent systems, lock-free data structures avoid common problems associated with conventional locking techniques, including priority inversion, convoying, and difficulty of avoiding deadlock. This paper introduces *transactional memory*, a new multiprocessor architecture intended to make lock-free synchronization as efficient (and easy to use) as conventional techniques based on mutual exclusion. Transactional memory allows programmers to define customized read-modify-write operations that apply

structures avoid common problems a
ventional locking techniques in highl

- *Priority inversion* occurs when
cess is preempted while holdi
higher-priority processes.
- *Convoying* occurs when a proc
scheduled, perhaps by exhaust
tum, by a page fault, or by som
When such an interruption oc
pable of running may be unab
- *Deadlock* can occur if proce
same set of objects in diff
avoidance can be awkward if
tiple data objects, particula

WEAK MEMORY + TM = ?

XBEGIN	**XBEGIN**
MOV [x] 1	**MOV** [y] 1
MOV r0 [y]	**MOV** r1 [x]
XEND	**XEND**

r0=1	r0=0	r0=1	r0=0
r1=1	r1=1	r1=0	r1=0

SC

x86

WEAK MEMORY + TM = ?

XBEGIN	**XBEGIN**
MOV [x] 1	**MOV** [y] 1
MOV r0 [y]	**MOV** r1 [x]
XEND	**XEND**

`r0=1` `r1=1`

`r0=0` `r1=1`

`r0=1` `r1=0`

`r0=0` `r1=0`

transactional SC

SC

x86

BUILDING OUR MODELS

x86:

acyclic($po_{\text{loc}} \cup com$) (Coherence)
empty($rmw \cap (fr_{\text{e}} ; co_{\text{e}})$) (RMWIsol)
acyclic(hb) (Order)
where $ppo = ((W \times W) \cup (R \times W) \cup (R \times R)) \cap po$
$tfence = po \cap ((\neg stxn ; stxn) \cup (stxn ; \neg stxn))$
$L = \text{domain}(rmw) \cup \text{range}(rmw)$
$implied = [L] ; po \cup po ; [L] \cup tfence$
$hb = mfence \cup ppo \cup implied \cup rf_{\text{e}} \cup fr \cup co$
acyclic(stronglift(com, $stxn$)) (StrongIsol)
acyclic(stronglift(hb, $stxn$)) (TxnOrder)

Power:

acyclic($po_{\text{loc}} \cup com$) (Coherence)
empty($rmw \cap (fr_{\text{e}} ; co_{\text{e}})$) (RMWIsol)
acyclic(hb) (Order)
where ppo = *(preserved program order, elided)*
$tfence = po \cap ((\neg stxn ; stxn) \cup (stxn ; \neg stxn))$
$fence = sync \cup tfence \cup (lwsync \setminus (W \times R))$
$ihb = ppo \cup fence$
$thb = (rf_{\text{e}} \cup ((fr_{\text{e}} \cup co_{\text{e}})^* ; ihb))^* ; (fr_{\text{e}} \cup co_{\text{e}})^* ; rf_{\text{e}}^?$
$hb = (rf_{\text{e}}^? ; ihb ; rf_{\text{e}}^?) \cup \text{weaklift}(thb, stxn)$
acyclic($co \cup prop$) (Propagation)
where $efence = rf_{\text{e}}^? ; fence ; rf_{\text{e}}^?$
$prop_1 = [W] ; efence ; hb^* ; [W]$
$prop_2 = com_{\text{e}}^* ; efence^* ; hb^* ; (sync \cup tfence) ; hb^*$
$tprop_1 = rf_{\text{e}} ; stxn ; [W]$
$tprop_2 = stxn ; rf_{\text{e}}$
$prop = prop_1 \cup prop_2 \cup tprop_1 \cup tprop_2$
irreflexive($fr_{\text{e}} ; prop ; hb^*$) (Observation)
acyclic(stronglift(com, $stxn$)) (StrongIsol)
acyclic(stronglift(hb, $stxn$)) (TxnOrder)
empty($rmw \cap tfence^*$) (TxnCancelsRMW)

ARM:

acyclic($po_{\text{loc}} \cup com$) (Coherence)
acyclic(ob) (Order)
where dob = *(order imposed by dependencies, elided)*
aob = *(order imposed by atomic RMWs, elided)*
bob = *(order imposed by barriers, elided)*
$tfence = po \cap ((\neg stxn ; stxn) \cup (stxn ; \neg stxn))$
$ob = com_{\text{e}} \cup dob \cup aob \cup bob \cup tfence$
empty($rmw \cap (fr_{\text{e}} ; co_{\text{e}})$) (RMWIsol)
acyclic(stronglift(com, $stxn$)) (StrongIsol)
acyclic(stronglift(ob, $stxn$)) (TxnOrder)
empty($rmw \cap tfence^*$) (TxnCancelsRMW)

C++:

irreflexive($hb ; com^*$) (HbCom)
where sw = *(synchronises-with, elided)*
$ecom = com \cup (co ; rf)$
$tsw = \text{weaklift}(ecom, stxn)$
$hb = (sw \cup tsw \cup po)^+$
empty($rmw \cap (fr_{\text{e}} ; co_{\text{e}})$) (RMWIsol)
acyclic($po \cup rf$) (NoThinAir)
acyclic(psc) (SeqCst)
where psc = *(constraints on SC events, elided)*

empty($cnf \setminus Ato^2 \setminus (hb \cup hb^{-1})$) (NoRace)
where $cnf = ((W \times W) \cup (R \times W) \cup (W \times R)) \cap sloc \setminus id$

VALIDATING OUR MODELS

1. Consult architecture manuals.

2. Interview engineers.

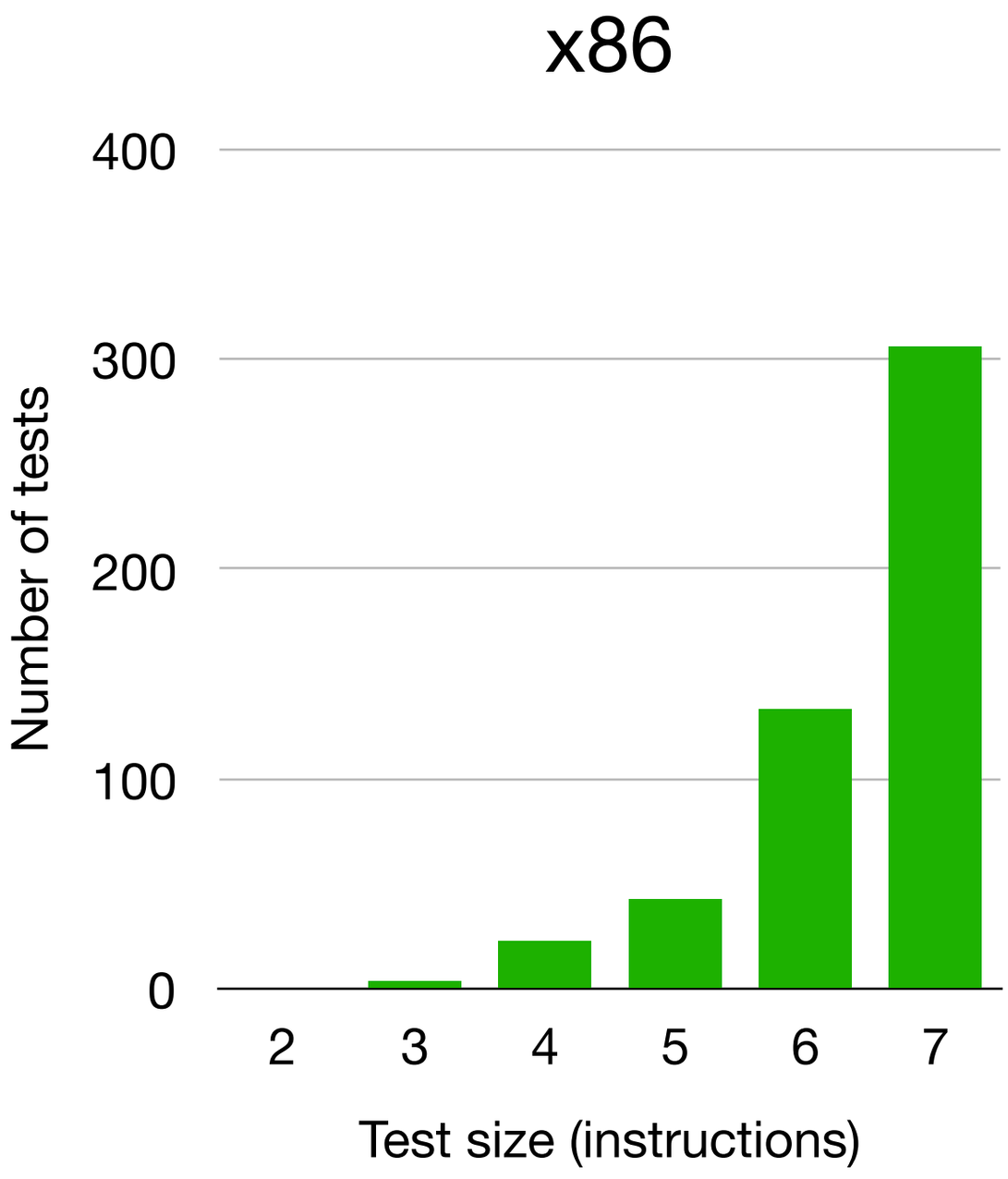
3. Check models have reasonable mathematical properties (e.g. adding/extending/coalescing transactions is safe).

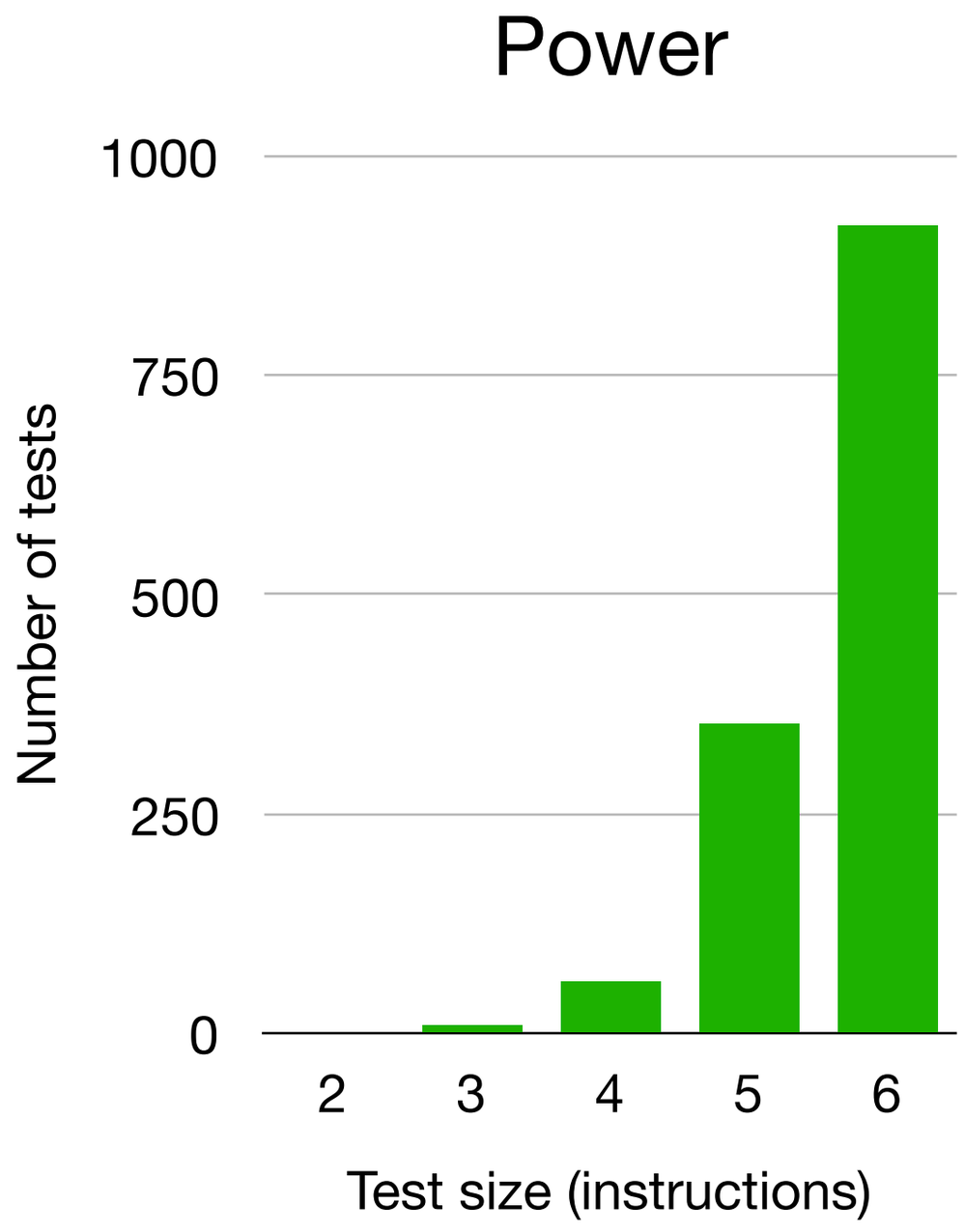
4. Check that models validate existing compiler mappings.

5. Generate conformance tests and run them on hardware.

VALIDATING OUR MODELS

Behaviours that must be <u>forbidden</u>

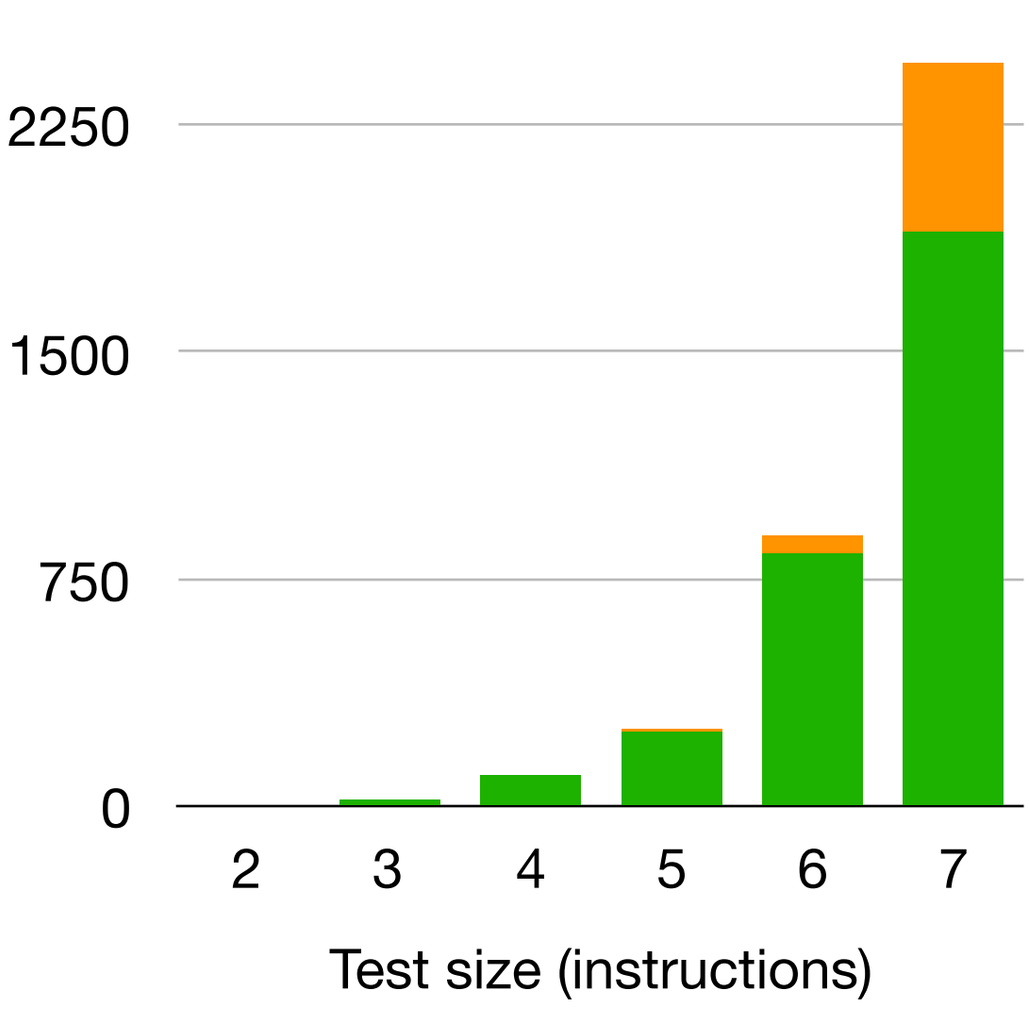




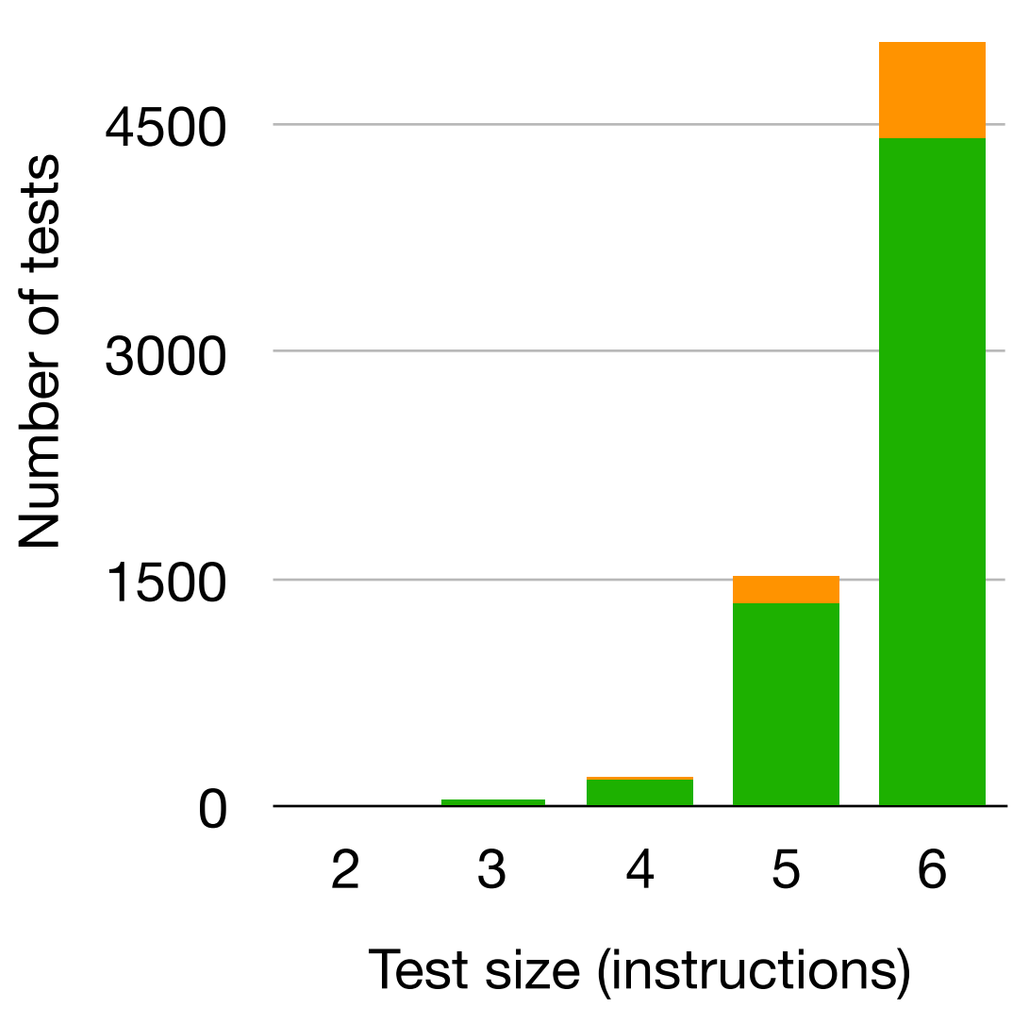
VALIDATING OUR MODELS

Behaviours that should be <u>allowed</u>

x86



Power



USING OUR MODELS

LOCK ELISION

```
lock()
X = X + 2
unlock()
```

```
lock()
X = 1
unlock()
```

LOCK ELISION

```
lock()
ldr  W5,[X]
add  W5,W5,#2
str  W5,[X]
unlock()
```

```
lock()
mov W7,#1
str W7,[X]
unlock()
```

LOCK ELISION

```
Loop:
ldaxr W2,[M]
cbnz  W2,Loop
mov   W3,#1
stxr  W4,W3,[M]
cbnz  W4,Loop
ldr   W5,[X]
add   W5,W5,#2
str   W5,[X]
stlr  WZR,[M]
```

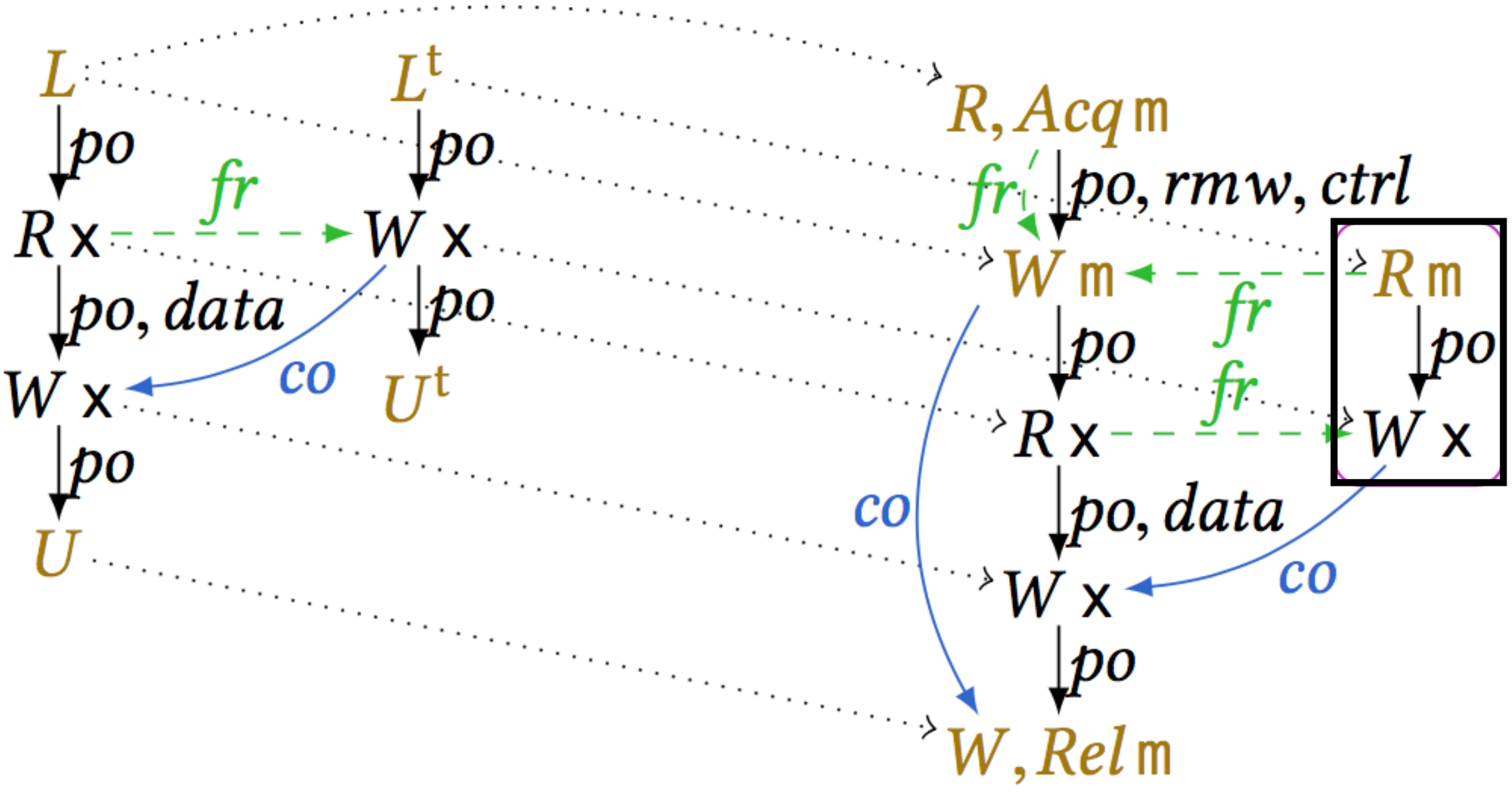
```
lock()
mov W7,#1
str W7,[X]
unlock()
```

LOCK ELISION

```
Loop:
ldaxr W2,[M]
cbnz W2,Loop
mov  W3,#1
stxr W4,W3,[M]
cbnz W4,Loop
ldr  W5,[X]
add  W5,W5,#2
str  W5,[X]
stlr WZR,[M]
```

```
tstart
ldr W6,[M]
cbz W6,L1
tcancel
L1:
mov W7,#1
str W7,[X]
tcommit
```

LOCK ELISION



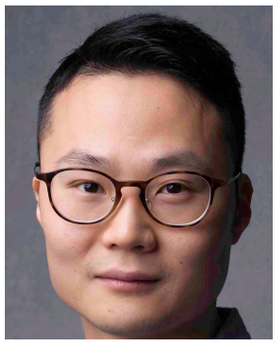
CONCLUSION

- Weak memory is pervasive, and transactional memory is entering the mainstream.

- We have designed and validated formal models of how these features interact in x86, Power, ARM, and C++.

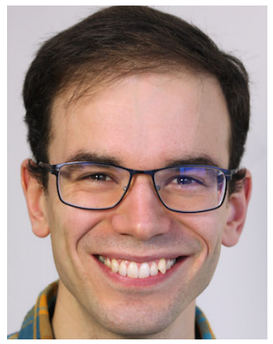
- Weak memory + transactions + lock elision = tricky!

THE SEMANTICS OF TRANSACTIONS AND WEAK MEMORY IN X86, POWER, ARM, AND C++



Nathan Chong
Arm Ltd.

Tyler Sorensen
Princeton University



John Wickerson
Imperial College London

USENIX Annual Technical Conference, 11 July 2019