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    r0=0    r0=1    r0=0
rl=1    rl=1    rl=0    rl=0
```

SC

x86
WEAK MEMORY!

```
x86
```

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

\[
\begin{align*}
\text{MOV} [x] &\ 1 \\
\text{MOV} [y] &\ 1 \\
\text{MOV} r0 [y] &\ \\
\text{MOV} r1 [x] &\
\end{align*}
\]

\[
\begin{array}{cccc}
\text{W x 1} & \text{W y 1} \\
\text{R y 1} & \text{R x 1} \\
\text{r0=1} & \text{r1=1} \\
x86: & \checkmark \\
\text{SC:} & \checkmark \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{W x 1} & \text{W y 1} \\
\text{R y 1} & \text{R x 1} \\
\text{r0=0} & \text{r1=1} \\
x86: & \checkmark \\
\text{SC:} & \checkmark \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{W x 1} & \text{W y 1} \\
\text{R y 1} & \text{R x 0} \\
\text{r0=1} & \text{r1=0} \\
x86: & \checkmark \\
\text{SC:} & \checkmark \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{W x 1} & \text{W y 1} \\
\text{R y 0} & \text{R x 0} \\
\text{r0=0} & \text{r1=0} \\
x86: & \checkmark \\
\text{SC:} & \times \\
\end{array}
\]
Transaction Memory: Architectural Support for Lock-Free Data Structures

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

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, convoys, and difficulty of avoiding deadlocks. 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 exclusivity. Transactional memory allows programmers to define and execute operations that apply to a collection of memory locations. It avoids deadlock by making it easy to correctly implement conflicting operations. The implementation is of general interest and remains in advance.

Transaction Memory:

• X86: xbegin
  ...
  xend

• Power: tbegin
  ...
  tend

• ARM: tstart
  ...
  tcommit

• C++: atomic {
  ...
}
WEAK MEMORY + TM = ?

\[
\begin{array}{c|c}
\text{XBEGIN} & \text{XBEGIN} \\
\text{MOV} [x] 1 & \text{MOV} [y] 1 \\
\text{MOV} r0 [y] & \text{MOV} r1 [x] \\
\text{XEND} & \text{XEND}
\end{array}
\]

\[
\begin{array}{lrrr}
r0=1 & r0=0 & r0=1 & r0=0 \\
r1=1 & r1=1 & r1=0 & r1=0
\end{array}
\]

SC

x86
WEAK MEMORY + TM = ?

**SC**

**x86**

**transactional SC**

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

**XBEGIN**

MOV [x] 1
MOV r0 [y]
XEND

**XBEGIN**

MOV [y] 1
MOV r1 [x]
XEND
BUILDING OUR MODELS

**x86:**

acyclic \(po_{loc} \cup com\) \hspace{1cm} (Coherence)
empty \((rmw \cap (fr_c \cup co_e))\) \hspace{1cm} (RMWIsol)
acyclic \((hb)\) \hspace{1cm} (Order)

where \(ppo = ((W \times W) \cup (R \times W) \cup (R \times R)) \cap po\)
\[tfence = po \cap ((\neg stxn; stxn) \cup (stxn; \neg stxn))\]
\[L = \text{domain}(rmw) \cup \text{range}(rmw)\]
\[implied = [L]; po \cup po; [L] \cup tfence\]
\[hb = mfence \cup ppo \cup implied \cup rf_e \cup fr \cup co\]
acyclic(stronglift \((com, stxn)\)) \hspace{1cm} (STGIsol)
acyclic(stronglift \((hb, stxn)\)) \hspace{1cm} (TXNOder)

**Power:**

acyclic \((po_{loc} \cup com)\) \hspace{1cm} (Coherence)
empty \((rmw \cap (fr_c \cup co_e))\) \hspace{1cm} (RMWIsol)
acyclic \((hb)\) \hspace{1cm} (Order)

where \(ppo = (\text{preserved program order, elided})\)
\[tfence = po \cap ((\neg stxn; stxn) \cup (stxn; \neg stxn))\]
fence = sync \cup tfence \cup (\neg sync \setminus (W \times R))
\[ihb = ppo \cup fence\]
\[ihb = (rf_e \cup ((fr_e \cup co_e)^* ; ihb^*); (fr_e \cup co_e)^* ; rf_e^2)\]
\[hb = (rf_e^2 ; ihb ; rf_e^2) \cup weaklift(ihb, stxn)\]
acyclic(\((co \cup prop)\)) \hspace{1cm} (PROPAGATION)

where \(efence = rf_e^2 \cup fence \cup rf_e^2\)
\[prop_1 = [W]; efence ; hb^* ; [W]\]
\[prop_2 = com_e^* ; efence^* ; hb^* ; (sync \cup tfence) ; hb^*\]
\[tprop_1 = rf_e ; stxn ; [W]\]
\[tprop_2 = stxn ; rf_e\]
\[prop = prop_1 \cup prop_2 \cup tprop_1 \cup tprop_2\]
irreflexive\((fr_c ; prop ; hb^*)\) \hspace{1cm} (HbCom)
acyclic\((com, stxn)\) \hspace{1cm} (STGIsol)
acyclic\((hb, stxn)\) \hspace{1cm} (TXNOder)
empty\((rmw \cap tfence^*\)) \hspace{1cm} (TXNCANCELARMW)

**ARM:**

acyclic \((po_{loc} \cup com)\) \hspace{1cm} (Coherence)
acyclic \((ob)\) \hspace{1cm} (Order)

where \(dob = (\text{order imposed by dependencies, elided})\)
\[aob = (\text{order imposed by atomic RMWs, elided})\]
\[bob = (\text{order imposed by barriers, elided})\]
\[tfence = po \cap ((\neg stxn; stxn) \cup (stxn; \neg stxn))\]
\[ob = com_e \cup do_b \cup aob \cup bob \cup tfence\]
empty \((rmw \cap (fr_c \cup co_e))\) \hspace{1cm} (RMWIsol)
acyclic\((stronglift(com, stxn))\) \hspace{1cm} (STGIsol)
acyclic\((stronglift(ob, stxn))\) \hspace{1cm} (TXNOder)
empty\((rmw \cap tfence^*\)) \hspace{1cm} (TXNCANCELARMW)

**C++:**

irreflexive\((hb ; com^*)\) \hspace{1cm} (HbCom)

where \(sw = (\text{synchronises-with, elided})\)
\[ecom = com \cup (co; rf)\]
\[tsw = weaklift(ecom, stxn)\]
\[hb = (sw \cup tsw \cup po)^+\]
empty \((rmw \cap (fr_c \cup co_e))\) \hspace{1cm} (RMWIsol)
acyclic\((po \cup rf)\) \hspace{1cm} (NoThinAir)
acyclic\((pse)\) \hspace{1cm} (SeqCst)

where \(psc = (\text{constraints on SC events, elided})\)
empty \(cnf \setminus (Ato^2 \setminus (hb \cup hb^{-1}))\) \hspace{1cm} (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 forbidden

**x86**

<table>
<thead>
<tr>
<th>Test size (instructions)</th>
<th>Number of tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>22</td>
</tr>
<tr>
<td>7</td>
<td>300</td>
</tr>
</tbody>
</table>

**Power**

<table>
<thead>
<tr>
<th>Test size (instructions)</th>
<th>Number of tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>250</td>
</tr>
<tr>
<td>6</td>
<td>1000</td>
</tr>
</tbody>
</table>

![Graphs showing the number of tests for different test sizes for x86 and Power architectures.](image-url)
VALIDATING OUR MODELS

Behaviours that should be allowed

x86

Test size (instructions)

Number of tests

Power

Test size (instructions)

Number of tests
USING OUR MODELS
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:
\texttt{ldaxr W2,[M]}
\texttt{cbnz W2,Loop}
\texttt{mov W3,#1}
\texttt{stxr W4,W3,[M]}
\texttt{cbnz W4,Loop}
\texttt{ldr W5,[X]}
\texttt{add W5,W5,#2}
\texttt{str W5,[X]}
\texttt{stlr WZR,[M]}

\texttt{lock()}
\texttt{mov W7,#1}
\texttt{str W7,[X]}
\texttt{unlock()}
LOCK ELISION

Loop:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldaxr W2,[M]</td>
<td>Load with acquire for W2 from memory location [M]</td>
</tr>
<tr>
<td>cbnz W2,Loop</td>
<td>Conditional branch on zero, if W2 is zero, branch to Loop</td>
</tr>
<tr>
<td>mov W3,#1</td>
<td>Move constant #1 into W3</td>
</tr>
<tr>
<td>stxr W4,W3,[M]</td>
<td>Store with release for W3 from memory location [M]</td>
</tr>
<tr>
<td>cbnz W4,Loop</td>
<td>Conditional branch on zero, if W4 is zero, branch to Loop</td>
</tr>
<tr>
<td>ldr W5,[X]</td>
<td>Load from memory location [X] into W5</td>
</tr>
<tr>
<td>add W5,W5,#2</td>
<td>Add #2 to W5, store result in W5</td>
</tr>
<tr>
<td>str W5,[X]</td>
<td>Store W5 into memory location [X]</td>
</tr>
<tr>
<td>stlr WZR,[M]</td>
<td>Store with release for WZR from memory location [M]</td>
</tr>
</tbody>
</table>

Start:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tstart</td>
<td>Start of the transaction</td>
</tr>
<tr>
<td>ldr W6,[M]</td>
<td>Load from memory location [M] into W6</td>
</tr>
<tr>
<td>cbz W6,L1</td>
<td>Conditional branch on zero, if W6 is zero, branch to L1</td>
</tr>
<tr>
<td>tcancel</td>
<td>Transaction cancel instruction</td>
</tr>
<tr>
<td>L1:</td>
<td>Branch target</td>
</tr>
<tr>
<td>mov W7,#1</td>
<td>Move constant #1 into W7</td>
</tr>
<tr>
<td>str W7,[X]</td>
<td>Store W7 into memory location [X]</td>
</tr>
<tr>
<td>tcommit</td>
<td>Transaction commit instruction</td>
</tr>
</tbody>
</table>
LOCK ELISION
• 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