Extending Intel-x86 Consistency and Persistency:
Formalising the Semantics of Intel-x86 Memory Types & Non-temporal Stores

Azalea Raad
Imperial College London

Luc Maranget
Inria Paris

Viktor Vafeiadis
MPI-SWS

POPL, 2022
Intel-x86 Non-temporal Stores

- Write *directly to memory*, bypassing cache
- Avoids *cache pollution*
- *Ubiquitous* (application-level use)
Intel-x86 Non-temporal Stores

- Write **directly to memory**, bypassing cache
- Avoids **cache pollution**
- **Ubiquitous** (application-level use)
  - 308K instances of MOVNTI on GitHub including in C, C++ & Assembly
Intel-x86 Non-temporal Stores

❖ Write *directly to memory*, bypassing cache
❖ Avoids *cache pollution*
❖ *Ubiquitous* (application-level use)
  ➡ 308K instances of MOVNTI on GitHub
     including in **C**, **C++** & **Assembly**
  ➡ `memset` function in the **C runtime**
  ➡ `memcpy` in **glibc**
Intel-x86 Non-temporal Stores

- Write **directly to memory**, bypassing cache
- Avoids **cache pollution**
- **Ubiquitous** (application-level use)
  - 308K instances of MOVNTI on GitHub including in C, C++ & Assembly
  - `memset` function in the C runtime
  - `memcpy` in glibc
  - Large-scale projects: PMDK and SPDK to interface with NVM
  - Large-scale projects: DPDK and DML to communicate with accelerators
Intel-x86 Memory Types

- Also known as *memory cacheability*: UC, WC, WT, WB

- **Non-cacheable** types: bypass memory, access (read/write) memory directly
  - UC: Strong Uncacheable
  - WC: Write Combining

- **Cacheable** types: memory accesses go through the cache hierarchy
  - WB: Write Back
  - WT: Write Through

* There are two other memory types: WP and UC
Intel-x86 Memory Types

- Also known as **memory cacheability**: UC, WC, WT, WB

- **Non-cacheable** types: bypass memory, access (read/write) memory directly
  - UC: Strong Uncacheable
  - WC: Write Combining

- **Cacheable** types: memory accesses go through the cache hierarchy
  - WB: Write Back
  - WT: Write Through

- Use within **system-level** code
  - Linux Kernel: WC for frame buffer optimisation
  - Linux Kernel: UC for memory-mapped I/O
  - Interaction with non-cache-coherent DMA device drivers

* There are two other memory types: WP and UC*
Intel-x86 Memory Types

- Also known as memory cacheability*: UC, WC, WT, WB

- Non-cacheable types: bypass memory, access (read/write) memory directly

- Cacheable types: memory accesses go through the cache hierarchy

- UC: Strong Uncacheable
- WC: Write Combining
- WB: Write Back
- WT: Write Through

- Use within system-level code
- Linux Kernel: WC for frame buffer optimisation
- Linux Kernel: UC for memory-mapped I/O

- Interaction with non-cache-coherent DMA device drivers

* There are two other memory types: WP and UC

Ex86 (Extended x86):

Formal consistency semantics of Intel-x86 architectures including non-temporal stores & memory types
Ex86: Extended Intel-x86 Consistency Semantics

Isn’t it just TSO?
Isn’t it just TSO?

TSO confirmed for **WB memory only**
Ex86: Extended Intel-x86 Consistency Semantics

### Store buffering (SB)
- Initially, $x = y = 0$
- $x := 1$  
- $a := y$  
- $b := x$

### Message passing (MP)
- Initially, $x = y = 0$
- $x := 1$
- $a := y$
- $y := 1$
- $b := x$
Ex86: Extended Intel-x86 Consistency Semantics

Store buffering (SB)

Initially, $x = y = 0$

$x := 1 \parallel y := 1$

$a := y \parallel b := x$ // 0

Message passing (MP)

Initially, $x = y = 0$

$x := 1 \parallel a := y$ // 1

$y := 1 \parallel b := x$ // 0
**Ex86: Extended Intel-x86 Consistency Semantics**

### Store buffering (SB)

Initially, \(x = y = 0\)

\[
\begin{align*}
x & := 1 & y & := 1 \\
 a & := y & b & := x
\end{align*}
\]

<table>
<thead>
<tr>
<th>SC</th>
<th>TSO</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✔</td>
</tr>
</tbody>
</table>

### Message passing (MP)

Initially, \(x = y = 0\)

\[
\begin{align*}
x & := 1 & a & := y \\
 y & := 1 & b & := x
\end{align*}
\]

<table>
<thead>
<tr>
<th>SC</th>
<th>TSO</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
**Ex86: Extended Intel-x86 Consistency Semantics**

<table>
<thead>
<tr>
<th>Store buffering (SB)</th>
<th>Message passing (MP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially, $x = y = 0$</td>
<td>Initially, $x = y = 0$</td>
</tr>
<tr>
<td>$x := 1$</td>
<td>$x := 1$</td>
</tr>
<tr>
<td>$a := y$ // 0</td>
<td>$a := y$ // 1</td>
</tr>
<tr>
<td>$b := x$ // 0</td>
<td>$b := x$ // 0</td>
</tr>
</tbody>
</table>

**SC**

- X
- ✔

**TSO/ WB, WT**

- X
- ✔

**WB, WT** memory are subject to **TSO** consistency:

write-read reordering
### WB and WT Memory Types

#### Table 11-2. Memory Types and Their Properties

<table>
<thead>
<tr>
<th>Memory Type and Mnemonic</th>
<th>Cacheable</th>
<th>Writeback Cacheable</th>
<th>Allows Speculative Reads</th>
<th>Memory Ordering Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Through (WT)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Speculative Processor Ordering.</td>
</tr>
<tr>
<td>Write Back (WB)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Speculative Processor Ordering.</td>
</tr>
</tbody>
</table>
## WB and WT Memory Types

**Table 11-2. Memory Types and Their Properties**

<table>
<thead>
<tr>
<th>Memory Type and Mnemonic</th>
<th>Cacheable</th>
<th>Writeback Cacheable</th>
<th>Allows Speculative Reads</th>
<th>Memory Ordering Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Through (WT)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Speculative Processor Ordering.</td>
</tr>
<tr>
<td>Write Back (WB)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Speculative Processor Ordering.</td>
</tr>
</tbody>
</table>

---

 applies only to **all-WB/ all-WT** accesses, **not mixed** accesses.
### WB and WT Memory Types

#### Table 11-2. Memory Types and Their Properties

<table>
<thead>
<tr>
<th>Memory Type and Mnemonic</th>
<th>Cacheable</th>
<th>Writeback Cacheable</th>
<th>Allows Speculative Reads</th>
<th>Memory Ordering Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Through (WT)</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Speculative Processor Ordering.</td>
</tr>
<tr>
<td>Write Back (WB)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Speculative Processor Ordering.</td>
</tr>
</tbody>
</table>

---

**Write-through (WT)** — Writes and reads to and from system memory are cached. Reads come from cache lines on cache hits; read misses cause cache fills. Speculative reads are allowed. All writes are written to a cache line (when possible) and through to system memory. When writing through to memory, invalid cache lines are never filled, and valid cache lines are either filled or invalidated. Write combining is allowed. This type of cache-control is appropriate for frame buffers or when there are devices on the system bus that access system memory, but do not perform snooping of memory accesses. It enforces coherency between caches in the processors and system memory.

**Write-back (WB)** — Writes and reads to and from system memory are cached. Reads come from cache lines on cache hits; read misses cause cache fills. Speculative reads are allowed. Write misses cause cache line fills (in processor families starting with the P6 family processors), and writes are performed entirely in the cache, when possible. Write combining is allowed. The write-back memory type reduces bus traffic by eliminating many unnecessary writes to system memory. Writes to a cache line are not immediately forwarded to system memory; instead, they are accumulated in the cache. The modified cache lines are written to system memory later, when a write-back operation is performed. Write-back operations are triggered when cache lines need to be deallocated, such as when new cache lines are being allocated in a cache that is already full. They also are triggered by the mechanisms used to maintain cache consistency. This type of cache-control provides the best performance, but it requires that all devices that access system memory on the system bus be able to snoop memory accesses to ensure system memory and cache coherency.

---

The extent of WB/WT Specification in the Intel manual

Note: The extent of WB/WT Specification in the Intel manual applies only to all-WB/ all-WT accesses, not mixed accesses.
**Ex86: Extended Intel-x86 Consistency Semantics**

### Store buffering (SB)
- Initially, $x = y = 0$
- $x := 1$
- $a := y$ // 0
- $b := x$ // 0

<table>
<thead>
<tr>
<th>SC</th>
<th>✔</th>
<th>✔</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSO/ WB, WT</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>UC</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

### Message passing (MP)
- Initially, $x = y = 0$
- $x := 1$
- $a := y$ // 1
- $y := 1$
- $b := x$ // 0

<table>
<thead>
<tr>
<th>SC</th>
<th>✗</th>
<th>✗</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSO/ WB, WT</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>UC</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
### Ex86: Extended Intel-x86 Consistency Semantics

#### Store buffering (SB)

Initially, \( x = y = 0 \)

- \( x := 1 \)
- \( a := y \)  // 0

\[
\begin{align*}
\text{SC} & \quad \times \quad \times \\
\text{TSO/WB, WT} & \quad \checkmark \\
\text{UC} & \quad \times \quad \times
\end{align*}
\]

#### Message passing (MP)

Initially, \( x = y = 0 \)

- \( x := 1 \)
- \( y := 1 \)
- \( a := y \)  // 1
- \( b := x \)  // 0

\[
\begin{align*}
\text{SC} & \quad \times \\
\text{TSO/WB, WT} & \quad \times \\
\text{UC} & \quad \times
\end{align*}
\]

**UC** memory is subject to **SC** consistency semantics:

- no reordering
## UC Memory Type

### Table 11-2. Memory Types and Their Properties

<table>
<thead>
<tr>
<th>Memory Type and Mnemonic</th>
<th>Cacheable</th>
<th>Writeback Cacheable</th>
<th>Allows Speculative Reads</th>
<th>Memory Ordering Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Uncacheable (UC)</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Strong Ordering</td>
</tr>
</tbody>
</table>

*SC*
**UC Memory Type**

**Table 11-2. Memory Types and Their Properties**

<table>
<thead>
<tr>
<th>Memory Type and Mnemonic</th>
<th>Cacheable</th>
<th>Writeback Cacheable</th>
<th>Allows Speculative Reads</th>
<th>Memory Ordering Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Uncacheable (UC)</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Strong Ordering</td>
</tr>
</tbody>
</table>

**applies only to all-UC accesses, not mixed accesses**
## UC Memory Type

### Table 11-2. Memory Types and Their Properties

<table>
<thead>
<tr>
<th>Memory Type and Mnemonic</th>
<th>Cacheable</th>
<th>Writeback Cacheable</th>
<th>Allows Speculative Reads</th>
<th>Memory Ordering Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Uncacheable (UC)</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Strong Ordering</td>
</tr>
</tbody>
</table>

This applies only to all-UC accesses, not mixed accesses.

### Strong Uncacheable (UC)

System memory locations are not cached. All reads and writes appear on the system bus and are executed in program order without reordering. No speculative memory accesses, page-table walks, or prefetches of speculated branch targets are made. This type of cache-control is useful for memory-mapped I/O devices. When used with normal RAM, it greatly reduces processor performance.

**Ex86: Extended Intel-x86 Consistency Semantics**

<table>
<thead>
<tr>
<th>Store buffering (SB)</th>
<th>Message passing (MP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially, $x = y = 0$</td>
<td>Initially, $x = y = 0$</td>
</tr>
<tr>
<td>$x := 1$</td>
<td>$a := y$ //1</td>
</tr>
<tr>
<td>$a := y$ //0</td>
<td>$y := 1$</td>
</tr>
<tr>
<td>$b := x$ //0</td>
<td>$b := x$ //0</td>
</tr>
</tbody>
</table>

| SC | ✔ | ✔ |
| TSO/ WB, WT | ✔ | ✗ |
| UC | ✗ | ✗ |
| WC | ✗ | ✔ |

**WC** memory: *write-write reordering* on different locations.
## WC Memory Type

### Table 11-2. Memory Types and Their Properties

<table>
<thead>
<tr>
<th>Memory Type and Mnemonic</th>
<th>Cacheable</th>
<th>Writeback Cacheable</th>
<th>Allows Speculative Reads</th>
<th>Memory Ordering Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Combining (WC)</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td><strong>Weak Ordering.</strong> Available by programming MTRRs or by selecting it through the PAT.</td>
</tr>
</tbody>
</table>

write-write reordering on different locations
## WC Memory Type

**Table 11-2. Memory Types and Their Properties**

<table>
<thead>
<tr>
<th>Memory Type and Mnemonic</th>
<th>Cacheable</th>
<th>Writeback Cacheable</th>
<th>Allows Speculative Reads</th>
<th>Memory Ordering Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Combining (WC)</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Weak Ordering. Available by programming MTRRs or by selecting it through the PAT.</td>
</tr>
</tbody>
</table>

*write-write reordering on different locations*

*applies only to all-WC accesses, not mixed accesses*
## WC Memory Type

### Table 11-2. Memory Types and Their Properties

<table>
<thead>
<tr>
<th>Memory Type and Mnemonic</th>
<th>Cacheable</th>
<th>Writeback Cacheable</th>
<th>Allows Speculative Reads</th>
<th>Memory Ordering Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Combining (WC)</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Weak Ordering. Available by programming MTRRs or by selecting it through the PAT.</td>
</tr>
</tbody>
</table>

**Write Combining (WC)** — System memory locations are not cached (as with uncachable memory) and coherency is not enforced by the processor's bus coherency protocol. Speculative reads are allowed. Writes may be delayed and combined in the write combining buffer (WC buffer) to reduce memory accesses. If the WC buffer is partially filled, the writes may be delayed until the next occurrence of a serializing event; such as an SFENCE or MFENCE instruction, CPUID or other serializing instruction, a read or write to uncached memory, an interrupt occurrence, or an execution of a LOCK instruction (including one with an XACQUIRE or XRELEASE prefix). In addition, an execution of the XEND instruction (to end a transactional region) evicts any writes that were buffered before the corresponding execution of the XBEGIN instruction (to begin the transactional region) before evicting any writes that were performed inside the transactional region.

This type of cache-control is appropriate for video frame buffers, where the order of writes is unimportant as long as the writes update memory so they can be seen on the graphics display. See Section 11.3.1, “Buffering of Write Combining Memory Locations,” for more information about caching the WC memory type. This memory type is available in the Pentium Pro and Pentium II processors by programming the MTRRs; or in processor families starting from the Pentium III processors by programming the MTRRs or by selecting it through the PAT.

**Write-write reordering on different locations**

**The extent of WC Specification in the Intel manual**

**applies only to all-WC accesses, not mixed accesses**
What about *Non-temporal Stores*?
These SSE and SSE2 non-temporal store instructions minimize cache pollutions by treating the memory being accessed as the write combining (WC) type.

Using the WC semantics, the store transaction will be weakly ordered, meaning that the data may not be written to memory in program order,
According to the Intel manual:

**Non-temporal stores** have the **same semantics** as **WC memory**
These SSE and SSE2 non-temporal store instructions minimize cache pollutions by treating the memory being accessed as the write combining (WC) type.

Using the WC semantics, the store transaction will be weakly ordered, meaning that the data may not be written to memory in program order.

According to the Intel manual: **Non-temporal stores** have the **same semantics** as **WC memory**

*But...*
**Ex86**: Extended Intel-x86 *Consistency* Semantics

**Store buffering (SB)**

Initially, $x = y = 0$

<table>
<thead>
<tr>
<th>Operation</th>
<th>SC</th>
<th>TSO/ WB, WT</th>
<th>UC</th>
<th>WC</th>
<th>MOVNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := 1$</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>$a := y$</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>$y := 1$</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>$b := x$</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
</tr>
</tbody>
</table>

**Message passing (MP)**

Initially, $x = y = 0$

<table>
<thead>
<tr>
<th>Operation</th>
<th>SC</th>
<th>TSO/ WB, WT</th>
<th>UC</th>
<th>WC</th>
<th>MOVNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := 1$</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>$a := y$</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>$y := 1$</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>$b := x$</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
### Ex86: Extended Intel-x86 Consistency Semantics

#### Store buffering (SB)

Initially, $x = y = 0$

- $x := 1$
- $a := y$  
- $y := 1$
- $b := x$

<table>
<thead>
<tr>
<th>SC</th>
<th>TSO/ WB, WT</th>
<th>UC</th>
<th>WC</th>
<th>MOVNNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✔</td>
</tr>
</tbody>
</table>

#### Message passing (MP)

Initially, $x = y = 0$

- $x := 1$
- $a := y$
- $y := 1$
- $b := x$

<table>
<thead>
<tr>
<th>SC</th>
<th>TSO/ WB, WT</th>
<th>UC</th>
<th>WC</th>
<th>MOVNNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✔</td>
</tr>
</tbody>
</table>

WC & NT stores have **different** semantics

- MOVNNT has ✔️
- WC has ✗ (red)

Note:
- WC & NT stores have different semantics.
**Ex86**: Extended Intel-x86 *Consistency* Semantics

**Solution**

**Validate** the Ex86 Consistency Semantics!

WC & NT stores have **different** semantics.
Ex86 Validation

- **Validated** Ex86 using the **diy** tool suite
- Extended the **klitmus** tool to allow for specifying memory types
- Built a test base of **over 2200 tests**
- Ran tests on **various Intel-x86 CPU implementations**
  - e.g. corei5, corei6 and Xeon
- Ran each test **at least 6 x 10^8 times**; ran critical tests up to a few billion times
Ex86 Validation

- **Validated** Ex86 using the **diy** tool suite
- Extended the **klitmus** tool to allow for specifying memory types
- Built a test base of **over 2200 tests**
- Ran tests on **various Intel-x86 CPU implementations**
  - e.g. corei5, corei6 and Xeon
- Ran each test **at least 6 x 10^8 times**; ran critical tests up to a few billion times
- For more details see: [http://diy.inria.fr/x86-memtype](http://diy.inria.fr/x86-memtype)
### Ex86 Semantics: Preserved Ordering

<table>
<thead>
<tr>
<th>Earlier in Program Order</th>
<th>Later in Program Order</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(R_{wb,wt})</td>
</tr>
<tr>
<td>(R)</td>
<td>✓</td>
</tr>
<tr>
<td>(W_{wb})</td>
<td>✗</td>
</tr>
<tr>
<td>(W_{wt,uc})</td>
<td>✗</td>
</tr>
<tr>
<td>(W_{wc,nt})</td>
<td>✗</td>
</tr>
<tr>
<td>(U, MF)</td>
<td>✓</td>
</tr>
<tr>
<td>(SF)</td>
<td>✗</td>
</tr>
<tr>
<td>(FL)</td>
<td>✗</td>
</tr>
<tr>
<td>(FO)</td>
<td>✗</td>
</tr>
</tbody>
</table>

- ✓ Order preserved; may not be reordered
- ✗ Order not preserved; may be reordered
- \(sloc\): Order preserved iff on the same location
- \(scl\): Order preserved iff on the same cache line
# Ex86 Semantics: Preserved Ordering

## Later in Program Order

<table>
<thead>
<tr>
<th>Earlier in Program Order</th>
<th>$R_{wb,wt}$</th>
<th>$R_{uc,wc}$</th>
<th>$W_{wb}$</th>
<th>$W_{uc,wt}$</th>
<th>$W_{wc,nt}$</th>
<th>$U, MF, SF$</th>
<th>$FL$</th>
<th>$FO$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$W_{wb}$</td>
<td>❌</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>sloc</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$W_{wt,uc}$</td>
<td>❌</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$W_{wc,nt}$</td>
<td>❌</td>
<td>✓</td>
<td>sloc</td>
<td>✓</td>
<td>sloc</td>
<td>✓</td>
<td>✓</td>
<td>scl</td>
</tr>
<tr>
<td>$U, MF$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$SF$</td>
<td>❌</td>
<td>✓†</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$FL$</td>
<td>❌</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>$FO$</td>
<td>❌</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

- **Order preserved; may not be reordered**
- **sloc:** Order preserved iff on the same location
- **scl:** Order preserved iff on the same cache line
- **Order not preserved may be reordered**
# Ex86 Semantics: Preserved Ordering

<table>
<thead>
<tr>
<th>Earlier in Program Order</th>
<th>Later in Program Order</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$R_{wb,wt}$</td>
</tr>
<tr>
<td>$R$</td>
<td>✓</td>
</tr>
<tr>
<td>$W_{wb}$</td>
<td>✗</td>
</tr>
<tr>
<td>$W_{wt,uc}$</td>
<td>✗</td>
</tr>
<tr>
<td>$W_{wc,nt}$</td>
<td>✗</td>
</tr>
<tr>
<td>$U, MF$</td>
<td>✓</td>
</tr>
<tr>
<td>$SF$</td>
<td>✗</td>
</tr>
<tr>
<td>$FL$</td>
<td>✗</td>
</tr>
<tr>
<td>$FO$</td>
<td>✗</td>
</tr>
</tbody>
</table>

- ✓ Order preserved; may not be reordered
- ✗ Order not preserved; may be reordered

**sloc:** Order preserved iff on the same location

**scl:** Order preserved iff on the same cache line
# Ex86 Semantics: Preserved Ordering

## Later in Program Order

<table>
<thead>
<tr>
<th>Earlier in Program Order</th>
<th>Later in Program Order</th>
<th>R&lt;sub&gt;wb,wt&lt;/sub&gt;</th>
<th>R&lt;sub&gt;uc,wc&lt;/sub&gt;</th>
<th>W&lt;sub&gt;wb&lt;/sub&gt;</th>
<th>W&lt;sub&gt;uc,wt&lt;/sub&gt;</th>
<th>W&lt;sub&gt;wc,nt&lt;/sub&gt;</th>
<th>U,MF,SF</th>
<th>FL</th>
<th>FO</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>W&lt;sub&gt;wb&lt;/sub&gt;</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>W&lt;sub&gt;wt,uc&lt;/sub&gt;</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>W&lt;sub&gt;wc,nt&lt;/sub&gt;</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>U,MF</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SF</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>FO</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
</tbody>
</table>

- ✓ Order preserved; may not be reordered
- X Order not preserved; may be reordered
- sloc: Order preserved iff on the same location
- scl: Order preserved iff on the same cache line
# Ex86 Semantics: Preserved Ordering

## Later in Program Order

<table>
<thead>
<tr>
<th>Earlier in Program Order</th>
<th>$R_{wb,wt}$</th>
<th>$R_{uc,wc}$</th>
<th>$W_{wb}$</th>
<th>$W_{uc,wt}$</th>
<th>$W_{wc,nt}$</th>
<th>$U, MF, SF$</th>
<th>$FL$</th>
<th>$FO$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R$</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>$W_{wb}$</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>sloc</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>$W_{wt,uc}$</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>$W_{wc,nt}$</td>
<td>✗</td>
<td>✔</td>
<td>sloc</td>
<td>✔</td>
<td>✔</td>
<td>sloc</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>$U, MF$</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>$SF$</td>
<td>✗</td>
<td>✔ (†)</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>$FL$</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>X</td>
</tr>
<tr>
<td>$FO$</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>✗</td>
<td>✔</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

- ✔ Order preserved; may not be reordered
- ✗ Order not preserved; may be reordered
- sloc: Order preserved iff on the same location
- scl: Order preserved iff on the same cache line
# Ex86 Semantics: Preserved Ordering

<table>
<thead>
<tr>
<th>Earlier in Program Order</th>
<th>Later in Program Order</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( R_{wb,wt} )</td>
</tr>
<tr>
<td>( R )</td>
<td>✓</td>
</tr>
<tr>
<td>( W_{wb} )</td>
<td>✓</td>
</tr>
<tr>
<td>( W_{wt,uc} )</td>
<td>✓</td>
</tr>
<tr>
<td>( W_{wc,nt} )</td>
<td>✓</td>
</tr>
<tr>
<td>( U,M,F )</td>
<td>✓</td>
</tr>
<tr>
<td>( SF )</td>
<td>✓</td>
</tr>
<tr>
<td>( FL )</td>
<td>✓</td>
</tr>
<tr>
<td>( FO )</td>
<td>✓</td>
</tr>
</tbody>
</table>

- **✓** Order preserved; may not be reordered
- **sloc**: Order preserved iff on the same location
- **scl**: Order preserved iff on the same cache line
- **✗** Order not preserved may be reordered
Ex86 Semantics: Two *Equivalent* Models

**Operational** Ex86

**Declarative** Ex86

Proved the *equivalence* of the two models
What about Intel-x86 *Persistency* Semantics?
Computer Storage

- **✓ fast**
- **✗ volatile**
- **✗ slow**
- **✓ persistent**
What is Non-Volatile Memory (NVM)?

**NVM: Hybrid Storage + Memory**

Best of both worlds:

- ✓ **persistent** (like HDD)
- ✓ **fast, random access** (like RAM)
What Can Go Wrong?

```haskell
// x=0; y=0

x := 1;

y := 1;
```
What Can Go Wrong?

// x=0; y=0
x := 1;
y := 1;

// x=1; y=1
What Can Go Wrong?

// x=0; y=0
x := 1;
y := 1;

// x=1; y=1 OR x=0; y=0

!! Execution continues *ahead of persistence*
   — *asynchronous* persists
What Can Go Wrong?

```
// x=0; y=0
x := 1;
y := 1;
```

!! Execution continues *ahead of persistence* — *asynchronous* persists
What Can Go Wrong?

// x=0; y=0
x := 1;
y := 1;

// x=1; y=1 OR x=0; y=0 OR x=1; y=0 OR x=0; y=1

!! Execution continues *ahead of persistence*
  – *asynchronous* persists

!! Writes may persist *out of order*
What Can Go Wrong?

Consistency Model

the order in which writes are made visible to other threads

Persistency Model

the order in which writes are persisted to NVM

Full Semantics

Consistency + Persistency Model
**PEx86 (Persistent Extended x86):**

Formal **consistency + Persistency** semantics of Intel-x86 architectures including non-temporal stores & memory types
PEx86 Semantics: Two Equivalent Models

**Operational** PEx86

**Declarative** PEx86

Proved the *equivalence* of the two models
## PEx86: Persistent Extended Intel-x86 Semantics

<table>
<thead>
<tr>
<th>x, y ∈ Loc(_{wb})</th>
<th>x, x', y ∈ Loc(_{wb})</th>
<th>x, x', y ∈ Loc(_{wb})</th>
<th>x, x', y ∈ Loc(_{wb})</th>
<th>x, x', y ∈ Loc(_{wb})</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := 1</td>
<td>x := 1</td>
<td>x := 1</td>
<td>x := 1</td>
<td>x := 1</td>
</tr>
<tr>
<td>y := 1</td>
<td>clflush x'</td>
<td>clflushopt x'</td>
<td>clflushopt x'</td>
<td>clflushopt x'</td>
</tr>
<tr>
<td></td>
<td>y := 1</td>
<td>y := 1</td>
<td>xchg(y, 1)</td>
<td>sfence y := 1</td>
</tr>
<tr>
<td>rec: x,y ∈ {0,1}</td>
<td>rec: y=1 ⇒ x=1</td>
<td>rec: x,y ∈ {0,1}</td>
<td>rec: y=1 ⇒ x=1</td>
<td>rec: y=1 ⇒ x=1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x ∈ Loc(_{uc∪wt})</th>
<th>y ∈ Loc(_{wc∪wb})</th>
<th>x ∈ Loc(<em>{wc}), y ∈ Loc(</em>{uc∪wt})</th>
<th>x ∈ Loc(<em>{wc}), y ∈ Loc(</em>{uc∪wt})</th>
<th>x ∈ Loc(<em>{wb∪wt∪wc}), y ∈ Loc(</em>{wc∪wb})</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := 1</td>
<td>x := 1</td>
<td>x := 1</td>
<td>x := 1</td>
<td>x := 1</td>
</tr>
<tr>
<td>y := 1</td>
<td>y := 1</td>
<td>y := 1</td>
<td>x := NT 2</td>
<td>x := NT 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rec: y=1 ⇒ x=1</td>
<td>y := 1</td>
<td>y := 1</td>
</tr>
<tr>
<td>rec: y=1 ⇒ x=1</td>
<td>rec: x,y ∈ {0,1}</td>
<td>rec: y=1 ⇒ x=1</td>
<td>rec: y=1 ⇒ x=2</td>
<td>rec: y=1 ⇒ x=2</td>
</tr>
</tbody>
</table>
Conclusions

❖ Developed **Ex86**: an extensive Intel-x86 *consistency* model
  ➪ Memory types (WB, WT, WC, UC)
  ➪ Non-temporal stores

❖ Formalised Ex86 both *operationally* & *declaratively*, and proved them *equivalent*

❖ **Empirically validated** Ex86 through extensive testing

❖ Developed **PEx86**: an extensive Intel-x86 *consistency* and *persistency* model
  ➪ Memory types (WB, WT, WC, UC)
  ➪ Non-temporal stores

❖ Formalised PEx86 both *operationally* & *declaratively*, and proved them *equivalent*
Conclusions

- Developed **Ex86**: an extensive Intel-x86 *consistency* model
  - Memory types (WB, WT, WC, UC)
  - Non-temporal stores

- Formalised Ex86 both *operationally* & *declaratively*, and proved them *equivalent*

- **Empirically validated** Ex86 through extensive testing

- Developed **PEx86**: an extensive Intel-x86 *consistency* and *persistency* model
  - Memory types (WB, WT, WC, UC)
  - Non-temporal stores

- Formalised PEx86 both *operationally* & *declaratively*, and proved them *equivalent*

- Future work
  - Program logics
  - Model checking algorithms
Conclusions

- Developed **Ex86**: an extensive Intel-x86 **consistency** model
  - Memory types (WB, WT, WC, UC)
  - Non-temporal stores

- Formalised Ex86 both **operationally & declaratively**, and proved them **equivalent**

- **Empirically validated** Ex86 through extensive testing

- Developed **PEx86**: an extensive Intel-x86 **consistency** and **persistency** model
  - Memory types (WB, WT, WC, UC)
  - Non-temporal stores

- Formalised PEx86 both **operationally & declaratively**, and proved them **equivalent**

- Future work
  - Program logics
  - Model checking algorithms

Thank You for Listening!

azalea@imperial.ac.uk
SoundAndComplete.org