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

Azalea Raad
Imperial College London

NANDA Workshop, 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)
  - 332K 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)
  - 332K 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)
  - 332K 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

* 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

* 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

**Ex86 (Extended x86):**

Formal **consistency** semantics of Intel-x86 architectures including

non-temporal stores & memory types

* There are two other memory types: WP and UC

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
Isn’t it just **TSO** (write-read reordering)?
Isn’t it just **TSO** (write-read reordering)?

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

**Store buffering (SB)**

Initially, $x = y = 0$

$x := 1$  \quad || \quad y := 1$

$a := y \quad // 0$  \quad || \quad b := x \quad // 0$

**Message passing (MP)**

Initially, $x = y = 0$

$x := 1$  \quad || \quad a := y \quad // 1$

$y := 1$  \quad || \quad b := x \quad // 0$
**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>$a := y$</td>
<td>$a := y$</td>
</tr>
<tr>
<td>$//0$</td>
<td>$//1$</td>
</tr>
<tr>
<td>$b := x$</td>
<td>$b := x$</td>
</tr>
<tr>
<td>$//0$</td>
<td>$//0$</td>
</tr>
</tbody>
</table>

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

**Store buffering (SB)**

Initially, $x = y = 0$

$x := 1$ // $y := 1$

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

- SC: X
- TSO: ✓

**Message passing (MP)**

Initially, $x = y = 0$

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

$y := 1$ $b := x$ // 0

- SC: X
- TSO: X
**Ex86: Extended Intel-x86 Consistency Semantics**

**Store buffering (SB)**

Initially, \( x = y = 0 \)

\[
\begin{align*}
x &:= 1 \\
a &:= y \quad \text{// 0} \\
y &:= 1 \\
b &:= x \quad \text{// 0}
\end{align*}
\]

- SC: ✗
- TSO/ WB, WT: ✔

**Message passing (MP)**

Initially, \( x = y = 0 \)

\[
\begin{align*}
x &:= 1 \\
a &:= y \quad \text{// 1} \\
y &:= 1 \\
b &:= x \quad \text{// 0}
\end{align*}
\]

- SC: ✗
- TSO/ WB, WT: ✗

**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>
<tr>
<td>Memory Type and Mnemonic</td>
<td>Cacheable</td>
<td>Writeback Cacheable</td>
<td>Allows Speculative Reads</td>
<td>Memory Ordering Model</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------</td>
<td>---------------------</td>
<td>-------------------------</td>
<td>-------------------------------------------</td>
</tr>
<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

[TSO]
### 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

**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.
Ex86: Extended Intel-x86 Consistency Semantics

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

SC ✔
TSO/ WB, WT ✓

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

✗
✗
**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>$a := y$</td>
<td>$a := y$</td>
</tr>
<tr>
<td>$b := x$</td>
<td>$b := x$</td>
</tr>
<tr>
<td>$// 0$</td>
<td>$// 1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Consistency Model</th>
<th>SB</th>
<th>MP</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC</td>
<td>✗</td>
<td>✗</td>
</tr>
<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

<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 ) ( \parallel ) ( y := 1 )</td>
<td>( x := 1 ) ( \parallel ) ( a := y ) ( \parallel 1 )</td>
</tr>
<tr>
<td>( a := y ) ( \parallel 0 ) ( b := x ) ( \parallel 0 )</td>
<td>( y := 1 ) ( \parallel b := x ) ( \parallel 0 )</td>
</tr>
</tbody>
</table>

- **SC**
  - ✗
  - ✗

- **TSO/ WB, WT**
  - ✓
  - ✗

- **UC**
  - ✗
  - ✗

**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>
## 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 SC</td>
</tr>
</tbody>
</table>
## 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 SC</td>
</tr>
</tbody>
</table>

- **SC** 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>

![](image)

**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.

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

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

- **SC**: ✗
- **TSO/ WB, WT**: ✔
- **UC**: ✗

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

- **SC**: ✗
- **TSO/ WB, WT**: ✗
- **UC**: ✗
**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>

**Consistency Models**

- **SC**: $X$
- **TSO/ WB, WT**: $✓$
- **UC**: $X$
- **WC**: $X$

**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>Weak Ordering. 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><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**
- **applies only to all-WC accesses, not mixed accesses**
The extent of WC Specification in the Intel manual

Write Combining (WC) — System memory locations are not cached (as with uncacheable 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.
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, $x = y = 0$

$x := 1 \quad \| \quad y := 1$

$a := y \quad // 0 \quad \| \quad b := x \quad // 0$

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

### Message passing (MP)

Initially, $x = y = 0$

$x := 1 \quad \| \quad a := y \quad // 1$

$y := 1 \quad \| \quad b := x \quad // 0$

<table>
<thead>
<tr>
<th>SC</th>
<th>✔</th>
</tr>
</thead>
<tbody>
<tr>
<td>TSO/ WB, WT</td>
<td>✔</td>
</tr>
<tr>
<td>UC</td>
<td>X</td>
</tr>
<tr>
<td>WC</td>
<td>✔</td>
</tr>
<tr>
<td>MOVNT</td>
<td>✔</td>
</tr>
</tbody>
</table>
Ex86: Extended Intel-x86 **Consistency** Semantics

Store buffering (SB)

Initially, $x = y = 0$

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

Message passing (MP)

Initially, $x = y = 0$

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

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Store buffering (SB)</th>
<th>Message passing (MP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC</td>
<td>$\times$</td>
<td>$\times$</td>
</tr>
<tr>
<td>TSO/ WB, WT</td>
<td>$\checkmark$</td>
<td>$\times$</td>
</tr>
<tr>
<td>UC</td>
<td>$\times$</td>
<td>$\times$</td>
</tr>
<tr>
<td>WC</td>
<td>$\times$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>MOVNT</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
</tbody>
</table>

WC & NT stores have **different** semantics.
**Solution**

*Validate* the Ex86 Consistency Semantics!
Ex86 Validation

- Validated Ex86 using the diy tool suite
- Extended the klitmus tool to allow for specifying memory types
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><strong>R</strong></td>
<td>✓</td>
</tr>
<tr>
<td><strong>$W_{wb}$</strong></td>
<td>×</td>
</tr>
<tr>
<td><strong>$W_{wt,uc}$</strong></td>
<td>×</td>
</tr>
<tr>
<td><strong>$W_{wc,nt}$</strong></td>
<td>×</td>
</tr>
<tr>
<td><strong>U,MF</strong></td>
<td>✓</td>
</tr>
<tr>
<td><strong>SF</strong></td>
<td>×</td>
</tr>
<tr>
<td><strong>FL</strong></td>
<td>×</td>
</tr>
<tr>
<td><strong>FO</strong></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

<table>
<thead>
<tr>
<th>Earlier in Program Order</th>
<th>Later 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>
<td>✓</td>
</tr>
<tr>
<td>(W_{wb})</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</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>
<td>✓</td>
</tr>
<tr>
<td>(W_{wc,nt})</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>X</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

\(\text{sloc}\): Order preserved iff on the same location
\(\text{scl}\): 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, 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
What about Intel-x86 *Persistency* Semantics?
Computer Storage

RAM

HDD
Computer Storage

✓ fast
✗ volatile
Computer Storage

✓ fast
✗ volatile

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

// x=0; y=0

x := 1;

y := 1;
What Can Go Wrong?

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

// 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;

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

!! 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
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
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: Persistent Extended Intel-x86 Semantics

<table>
<thead>
<tr>
<th>$x, y \in \text{Loc}_{wb}$</th>
<th>$x, y \in \text{Loc}_{wb}$</th>
<th>$x, x', y \in \text{Loc}_{wb}$</th>
<th>$x, x', y \in \text{Loc}_{wb}$</th>
<th>$x, x', y \in \text{Loc}_{wb}$</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 $x$, $y := 1$</td>
<td>clflushopt $x'$, $y := 1$</td>
<td>clflushopt $x'$, $xchg(y, 1)$</td>
<td>clflushopt $x'$, sfence, $y := 1$</td>
</tr>
<tr>
<td>rec: $x, y \in {0,1}$</td>
<td>rec: $y = 1 \Rightarrow x = 1$</td>
<td>rec: $x, y \in {0,1}$</td>
<td>rec: $y = 1 \Rightarrow x = 1$</td>
<td>rec: $y = 1 \Rightarrow x = 1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x \in \text{Loc}_{uc \cup wt}$, $y \in \text{Loc}$</th>
<th>$x \in \text{Loc}<em>{wc}$, $y \in \text{Loc}</em>{wc \cup wb}$</th>
<th>$x \in \text{Loc}<em>{wc}$, $y \in \text{Loc}</em>{uc \cup wt}$</th>
<th>$x \in \text{Loc}<em>{wb \cup wt \cup wc}$, $y \in \text{Loc}</em>{uc \cup wt}$</th>
<th>$x \in \text{Loc}<em>{wb \cup wt \cup wc}$, $y \in \text{Loc}</em>{wc \cup wb}$</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>$y := 1$</td>
</tr>
<tr>
<td>rec: $y = 1 \Rightarrow x = 1$</td>
<td>rec: $x, y \in {0,1}$</td>
<td>rec: $y = 1 \Rightarrow x = 1$</td>
<td>rec: $y = 1 \Rightarrow x = 2$</td>
<td>rec: $y = 1 \Rightarrow x = 2$</td>
</tr>
</tbody>
</table>
### PEx86: Persistent Extended Intel-x86 Semantics

<table>
<thead>
<tr>
<th>$x, y \in \text{Loc}_{\text{wb}}$</th>
<th>$x, y \in \text{Loc}_{\text{wb}}$</th>
<th>$x, x', y \in \text{Loc}_{\text{wb}}$</th>
<th>$x, x', y \in \text{Loc}_{\text{wb}}$</th>
<th>$x, x', y \in \text{Loc}_{\text{wb}}$</th>
</tr>
</thead>
</table>
| $x := 1$ | $x := 1$ | $x := 1$ | $x := 1$ | $x := 1$
| $y := 1$ | clflush $x$ | clflushopt $x'$ | y := 1 | clflushopt $x'$
| | | | xchg($y, 1$) | sfence
| rec: $x, y \in \{0, 1\}$ | rec: $x = 1 \Rightarrow x = 1$ | rec: $x, y \in \{0, 1\}$ | rec: $y = 1 \Rightarrow x = 1$ | rec: $y = 1 \Rightarrow x = 1$

<table>
<thead>
<tr>
<th>$x \in \text{Loc}_{\text{uc} \cup \text{wt}}$</th>
<th>$x \in \text{Loc}_{\text{uc} \cup \text{wt}}$</th>
<th>$x \in \text{Loc}<em>{\text{wc}}$, $y \in \text{Loc}</em>{\text{wc} \cup \text{wb}}$</th>
<th>$x \in \text{Loc}<em>{\text{wc}}$, $y \in \text{Loc}</em>{\text{uc} \cup \text{wt}}$</th>
<th>$x \in \text{Loc}<em>{\text{wb} \cup \text{wt} \cup \text{wc}}$, $y \in \text{Loc}</em>{\text{wc} \cup \text{wb}}$</th>
</tr>
</thead>
</table>
| $x := 1$ | $x := 1$ | $x := 1$ | $x := 1$ | $x := 1$
| $y := 1$ | $y := 1$ | $y := 1$ | $x := \text{NT} 2$ | $x := \text{NT} 2$
| | | | $y := 1$ | sfence
| | | | | $y := 1$
| rec: $y = 1 \Rightarrow x = 1$ | rec: $x, y \in \{0, 1\}$ | rec: $y = 1 \Rightarrow x = 1$ | rec: $y = 1 \Rightarrow x = 2$ | rec: $y = 1 \Rightarrow x = 2$
# PEx86: Persistent Extended Intel-x86 Semantics

<table>
<thead>
<tr>
<th>$x, y \in \text{Loc}_{wb}$</th>
<th>$x, y \in \text{Loc}_{wb}$</th>
<th>$x, x', y \in \text{Loc}_{wb}$</th>
<th>$x, x', y \in \text{Loc}_{wb}$</th>
<th>$x, x', y \in \text{Loc}_{wb}$</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>$x := 1$</td>
<td>xchg($y, 1$)</td>
<td>sfence</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$y := 1$</td>
</tr>
<tr>
<td>rec: $x,y \in {0,1}$</td>
<td>rec: $y=1$ $\Rightarrow x=1$</td>
<td>rec: $x,y \in {0,1}$</td>
<td>rec: $y=1$ $\Rightarrow x=1$</td>
<td>rec: $y=1$ $\Rightarrow x=1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x \in \text{Loc}_{uc \cup wt}$</th>
<th>$x \in \text{Loc}<em>{wc}$, $y \in \text{Loc}</em>{wc \cup wb}$</th>
<th>$x \in \text{Loc}<em>{wc}$, $y \in \text{Loc}</em>{uc \cup wt}$</th>
<th>$x \in \text{Loc}_{wb \cup wt \cup wc}$</th>
<th>$x \in \text{Loc}_{wb \cup wt \cup wc}$</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>$x := \text{NT}$ 2</td>
<td>$x := \text{NT}$ 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$y := 1$</td>
<td>sfence</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$y := 1$</td>
</tr>
<tr>
<td>rec: $y=1$ $\Rightarrow x=1$</td>
<td>rec: $x,y \in {0,1}$</td>
<td>rec: $y=1$ $\Rightarrow x=1$</td>
<td>rec: $y=1$ $\Rightarrow x=2$</td>
<td>rec: $y=1$ $\Rightarrow x=2$</td>
</tr>
</tbody>
</table>
Persistency Validation?

- How to test for post-crash behaviours?
Persistency Validation?

- How to test for post-crash behaviours?
  1. Contrive crashes (e.g. pull the plug) at crucial times
Persistency Validation?

- How to test for post-crash behaviours?

1. Contrive crashes (e.g. pull the plug) at crucial times

   Do this for **thousands** of tests, each for **hundreds of millions** of times $\geq 10^{11}$ **crashes**

   ![Infeasible emoji]

   **Infeasible!**
Persistency Validation?

- How to test for post-crash behaviours?

  1. Contrive crashes (e.g. pull the plug) at crucial times

     Do this for **thousands** of tests, each for **hundreds of millions** of times $\geq 10^{11}$ crashes

     😯 **Infeasible !**

  2. Directly monitor the memory bus for the order of stores

     😐 **Promising !**
Persistency Validation?

❖ How to test for post-crash behaviours?

1. Contrive crashes (e.g. pull the plug) at crucial times
   
   Do this for **thousands** of tests, each for **hundreds of millions** of times $\geq 10^{11}$ crashes

   🤯 **Infeasible !**

2. Directly monitor the memory bus for the order of stores

   🤔 **Promising !**
Monitoring the Memory Bus with DDR Detective

flush stores
(persistency order)

CPU

bus

NVM
Monitoring the Memory Bus with DDR Detective

- Listen
- Log stores
- Observe order
Monitoring the Memory Bus with DDR Detective

- Listen
- Log stores
- Observe order

flush stores (persistency order)
Monitoring the Memory Bus with DDR Detective

flush stores (persistency order)

CPU

bus

NVM

DDR Detective
Monitoring the Memory Bus with DDR Detective

x := 1;
clflush x;
y := 1

flush stores (persistency order)
Monitoring the Memory Bus with DDR Detective

x := 1; clflush x;
y := 1

Expected Orders in Log
1. x := 1
2. y := 1

DDR Detective
Monitoring the Memory Bus with DDR Detective

x := 1;
clfflush x;
y := 1

Expected Orders in Log
1. x := 1  1. y := 1
2. y := 1  2. x := 1
Monitoring the Memory Bus with DDR Detective

```
x := 1;
clfush x;
y := 1
```

**Expected Orders in Log**

1. x := 1  
2. y := 1

**Actual Orders in Log**

1. y := 1  
2. x := 1

- **Correct**: ✔
- **Incorrect**: ✗

indicates incorrect chip implementation
Monitoring the Memory Bus with DDR Detective

x := 1;
clflush x;
y := 1

Expected Orders in Log

1. x := 1   1. y := 1
2. y := 1   2. x := 1

indicates incorrect chip implementation
Monitoring the Memory Bus with DDR Detective

x := 1;
clflush x;
y := 1

flush stores
(persistency order)

Observed Orders in Log
Monitoring the Memory Bus with DDR Detective

x := 1; clflush x; y := 1

flush stores (persistency order)

Observed Orders in Log
1. x := 1
2. y := 1
Monitoring the Memory Bus with DDR Detective

x := 1;
clflesh x;
y := 1

1. x := 1
2. y := 1
1. y := 1
2. x := 1

Observed Orders in Log
Monitoring the Memory Bus with DDR Detective

flush stores (persistency order)

Oberved Orders in Log

1. x := 1  1. y := 1
2. y := 1  2. x := 1

x := 1;
clfush x;
y := 1
Not so fast…
Not so fast…

forward stores

write-pending queue
battery-backed
i.e. **persistent**
Not so fast…

forward stores

flush stores (possibly reordered)

write-pending queue
battery-backed
i.e. persistent

CPU

WPQ

bus

NVM
Not so fast…

- forward stores
  - persistency order
- flush stores (possibly reordered)

write-pending queue
- battery-backed
- i.e. **persistent**
Not so fast…

- **forward stores**
  - **persistency order**

- **flush stores**
  - (possibly reordered)

**write-pending queue**
- battery-backed
- i.e. **persistent**

**CPU** — **WPQ** — **bus** — **NVM**
Not so fast…

- Forward stores: persistence order
- Flush stores: (possibly reordered)
- Write-pending queue: battery-backed, i.e., persistent

- Must monitor here: (unclear how to do)
- Too late to monitor
Not so fast…

forward stores
persistency order

flush stores
(possibly reordered)

write-pending queue
battery-backed
i.e. persistent

must monitor here
(unclear how to do)

too late to monitor

Inconclusive validation
Conclusions

- Developed **Ex86** and **PEx86**: extensive Intel-x86 **consistency** and **persistence** models
  - Memory types (WB, WT, WC, UC)
  - Non-temporal stores

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

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

- **Empirically validated** Ex86 through extensive testing

- Attempted to **validate** PEx86; inconclusive results
Conclusions

- Developed **Ex86** and **PEx86**: extensive Intel-x86 **consistency** and **persistence** models
  - Memory types (WB, WT, WC, UC)
  - Non-temporal stores
- Formalised **Ex86** both **operationally** & **declaratively**, and proved them **equivalent**
- Formalised **PEx86** both **operationally** & **declaratively**, and proved them **equivalent**
- **Empirically validated** Ex86 through extensive testing
- Attempted to **validate** PEx86; inconclusive results

Thank You for Listening!

azalea@imperial.ac.uk

SoundAndComplete.org

@azalearaad