



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**

The screenshot shows a GitHub search interface for the keyword "MOVNTI". On the left sidebar, the "Code" category is selected, showing 308K results. Below it, the "Languages" section lists various programming languages with their respective result counts: CSV (1,900), C (102,715), Lua (5,530), Makefile (9,001), Unix Assembly (46,848), JavaScript (11,240), Lex (490), JSON (7,353), LLVM (483), HTML (585), Vim Script (6,691), Text (54,804), Assembly (420), and C++ (27,361). The main area displays "308,196 code results" with a "Sort: Best match" dropdown. Three code snippets are shown as search results, all from the repository "stubdom/newlib-1.16.0/newlib/libc/machine/x86_64/memset.S" (or ".bak" for backups). Each snippet shows assembly code with the MOVNTI instruction used in a loop to store 128 bytes at a time with minimum cache pollution. The code is as follows:

```
39  shrq  $7, rcx          /* Store 128 bytes at a time with minimum cache
40  polution */
41  .p2align 4
42  loop:
43  movnti rax, (rdi)
44  movnti rax, 8 (rdi)
45  movnti rax, 16 (rdi)
46  movnti rax, 24 (rdi)
47  movnti rax, 32 (rdi)
```

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

Ex86 (Extended x86):

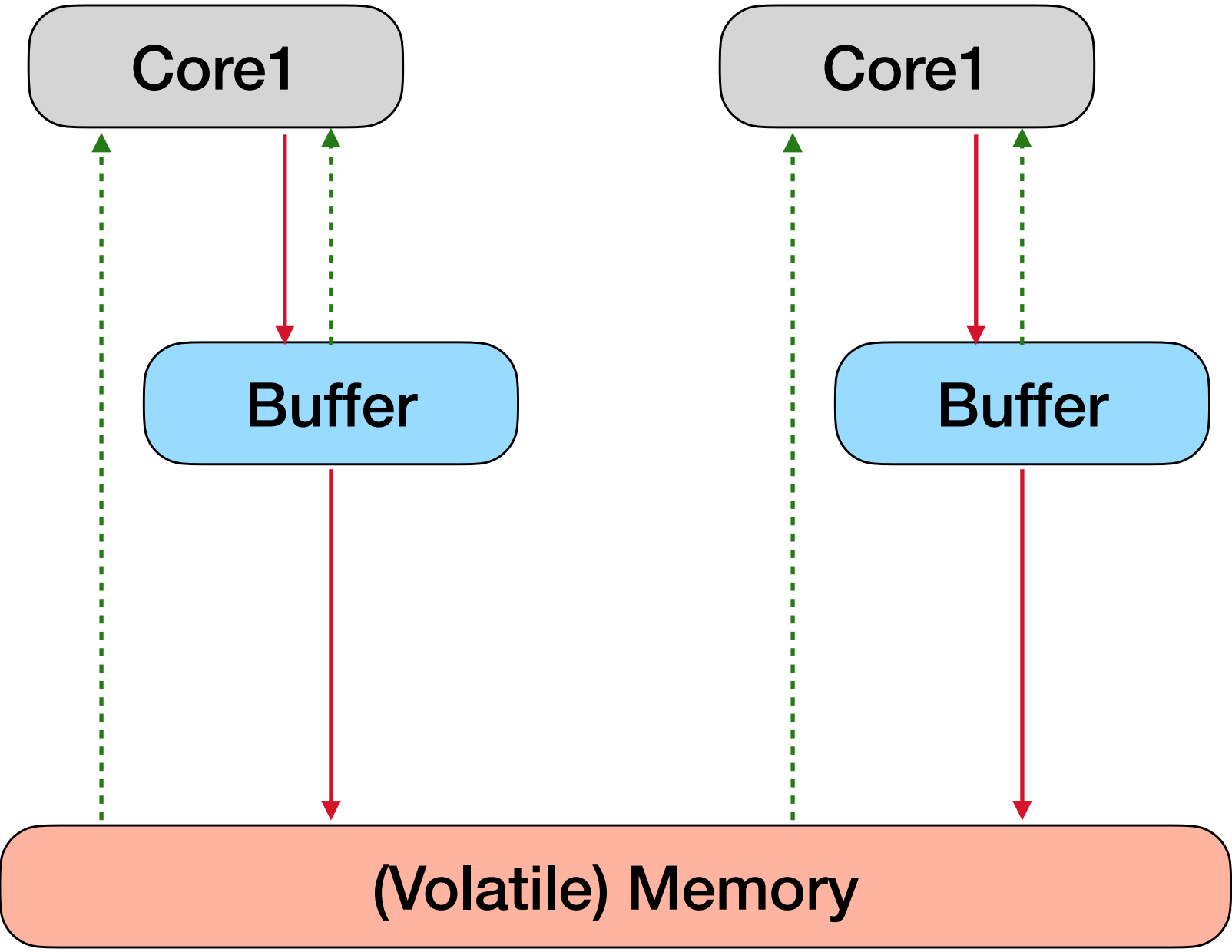
Formal **consistency** semantics of Intel-x86 architectures
including
non-temporal stores & memory types

→ Interaction with non-cache-coherent DMA device drivers

^{*} There are two other memory types: WP and UC⁻

Ex86: Extended Intel-x86 Consistency Semantics

Isn't it just **TSO**?



A Better x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors

Scott Owens
University of Cambridge

Peter Sewell
University of Cambridge

Francesco Zappa Nardelli
INRIA

Susmit Sarkar
University of Cambridge

Magnus O. Myreen
University of Cambridge

<http://www.cl.cam.ac.uk/users/pes20/weakmemory>

ABSTRACT. Exploiting the multiprocessors that have recently become ubiquitous requires high-performance and reliable concurrent systems code, for concurrent programming, compilers, and operating system kernels, for example. However, concurrent multiprocessors typically do not provide the sequentially consistent memory that is assumed by most work on semantics and verification. Instead, they have relaxed memory models, varying in subtle ways between processor families, consistent views of a shared memory. Second, the public vendor architectures, supposedly specifying what programmers can rely on, are often in ambiguous informal prose (a particularly poor medium for loose specifications), leading to widespread confusion.

In this paper we focus on x86 processors. We review several recent Intel and AMD specifications, showing that all contain serious ambiguities, some are arguably too weak to model that, to the best of our knowledge, suffers from none of these problems. It is mathematically precise (rigorous abstract machine which should be widely accessible to working programmers. We illustrate how this can be used to reason about the correctness of a Linux spinlock implementation and describe a general theory of data-race-freedom for x86-TSO. This should put x86 multiprocessor system building on a more solid foundation; it should also provide a basis for future work on verification of such systems.

1 Introduction

Most previous research on assumes sequential consistency occur in a global memory model for concurrent programs. In this paper we consider the behaviour of concurrent programs on two processors, given two memory locations, `proc:0` and `proc:1` respectively, as in the program below,

<code>iwp2.3.a/amd4</code>	<code>pr</code>
<code>poi:0</code>	<code>MOV [y]</code>
<code>poi:1</code>	<code>MOV EAX ← [y]</code>
Allow: <code>0:EAX=0 ∧ 1:EBX=0</code>	<code>MOV EBX ← [x]</code>

One can view this as a visible consistency model where each processor effectively has a FIFO buffer of writes.

1. INTRODUCTION

Multiprocessor machines, with many processors acting on a shared memory, have been developed since the 1960s; they are now ubiquitous. Meanwhile, the difficulty of programming concurrent systems has motivated extensive research on programming language design, semantics, and verification, from semaphores and monitors to program logics, software model checking, and so forth. This work has almost always assumed that concurrent threads share a single sequentially consistent memory [21], with their reads and writes interleaved in some order. In fact, however, real multiprocessors use sophisticated techniques to achieve high performance: store buffers, hierarchies of local caches, etc.

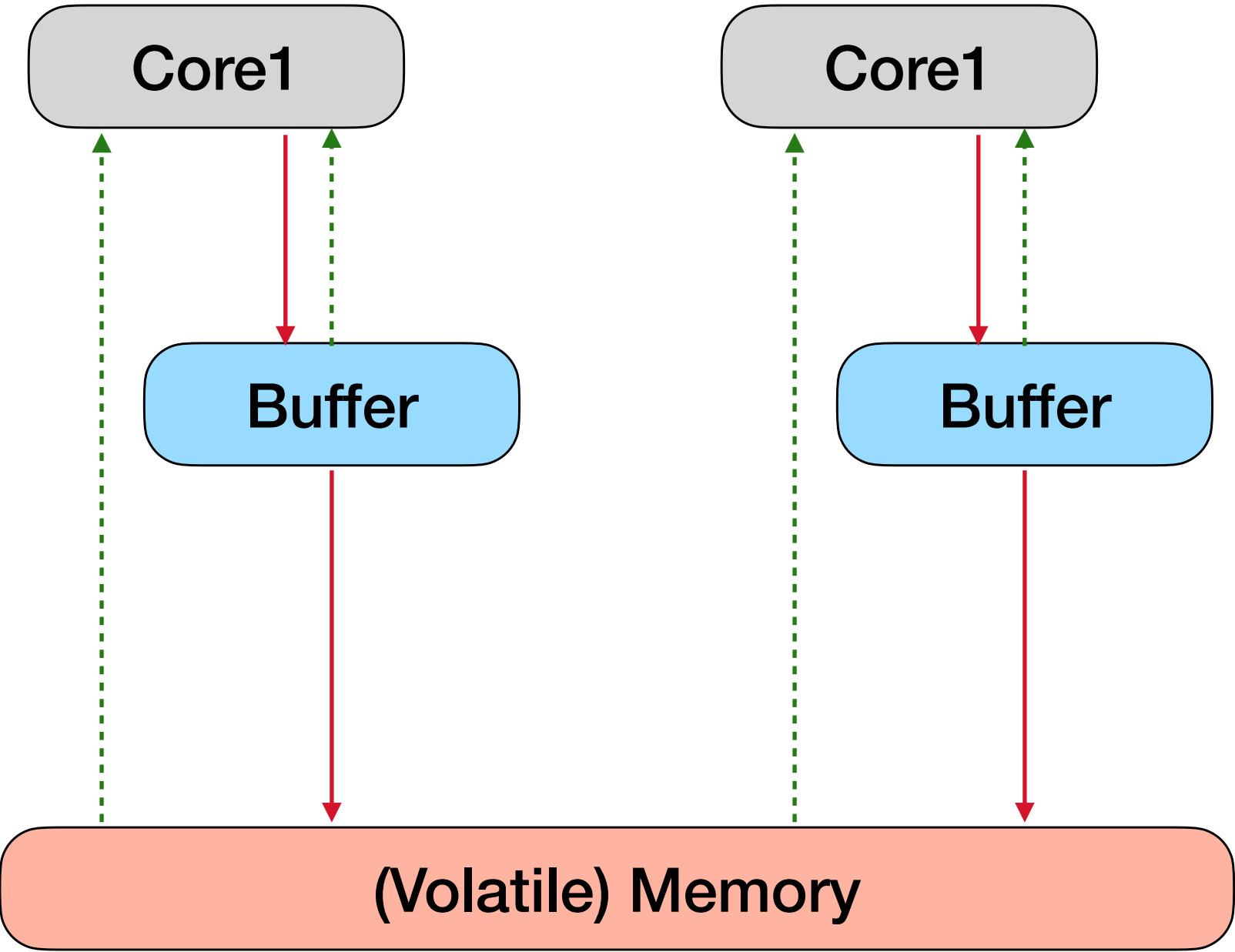
Proc 0	Proc 1
<code>MOV [x] ← 1</code>	<code>MOV [y] ← 1</code>
<code>MOV EAX ← [y]</code>	<code>MOV EBX ← [x]</code>
Allowed Final State: <code>Proc 0:EAX=0 ∧ Proc 1:EBX=0</code>	

Microarchitecturally, one can view this particular example as a visible consequence of store buffering: if each processor effectively has a FIFO buffer of pending memory writes (to avoid the need to block while a write completes), then the reads from `y` and `x` could occur before the writes have propagated from the buffers to main memory. Other families of multiprocessors, dating back at least to the IBM 370, and including ARM, Itanium, POWER, and SPARC, also exhibit relaxed-memory behaviour. Moreover, there are major and subtle differences between different processor families (arising from their different internal design choices): in the details of exactly what non-sequentially-consistent executions they permit, and of what memory programmer synchronisation instructions they provide. For any of these processors, relaxed-memory semantics exacerbates the difficulties of writing correct concurrent systems programmers cannot rely on the sequential concept of global memory order. Still, work on multiprocessor semantics has made significant progress in the last few years, and it is now possible to give a rigorous and usable model for x86 multiprocessors.

Ex86: Extended Intel-x86 Consistency Semantics

Isn't it just **TSO**?

TSO confirmed for **WB** memory only



A Better x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors
Scott Owens
University of Cambridge
<http://www.cl.cam.ac.uk/users/pes20/weakmemory>

x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors
Peter Sewell
University of Cambridge
Francesco Zappa Nardelli
INRIA
<http://www.cl.cam.ac.uk/users/pes20/weakmemory>

x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors
Susmit Sarkar
University of Cambridge
Magnus O. Myreen
University of Cambridge

Abstract. Real multiprocessor machines that have recently become ubiquitous require high-performance and reliable concurrent systems code, for concurrent programming, compilers, operating system kernels, synchronisation libraries, and so on. However, concurrent programming typically does not provide the sequential consistency of a shared memory. Instead, they have relaxed memory models, varying in subtle ways between processor families, consistent memory that is assumed by most work on semantics and verification. In this paper we focus on x86 processors. We review several recent Intel and AMD specifications, showing that all contain serious ambiguities, some are arguably too weak to be used as a basis for formal reasoning about the correctness of a Linux spinlock implementation, and some are simply unsound with respect to actual hardware. We present a new x86-TSO programmatic model that, to the best of our knowledge, suffers from none of these problems. It is mathematically precise (rigorous abstract machine which should be widely accessible to working programmers. We illustrate how this can be used to reason about the correctness of a Linux spinlock implementation and describe a general theory of data-race-freedom for x86-TSO. This should put x86 multiprocessor system building on a more solid foundation; it should also provide a basis for future work on verification of such systems.

1. INTRODUCTION
Multiprocessor machines, with many processors acting on a shared memory, have been developed since the 1960s; they are now ubiquitous. Meanwhile, the difficulty of programming concurrent systems has motivated extensive research on programming language design, semantics, and verification, from semaphores and monitors to program logics, software model checking, and so forth. This work has almost always assumed that concurrent threads share a single sequentially consistent memory [21], with their reads and writes interleaved in some order. In fact, however, real multiprocessors use sophisticated techniques to achieve high performance: store buffers, hierarchies of local caches, and so on. These techniques are not observable by sequential code, but in multithreaded programs different threads may see subtly different views of memory; such machines exhibit *relaxed*, or *weak*, memory models [6, 17, 19, 7].

For a simple example, consider the following assembly language program (SB) for modern Intel or AMD x86 multiprocessors: given two distinct memory locations x and y (initially holding 0), if two processors respectively write 1 to x and y and then read from y and x (into register EAX to read 0 in the same execution. It is easy to check that this result cannot arise from any interleaving of the reads and writes of the two processors; modern x86 multiprocessors do not have a sequentially consistent semantics.

Proc 0	Proc 1
MOV $x \leftarrow 1$	MOV $y \leftarrow 1$
MOV $EAX \leftarrow y$	MOV $EBX \leftarrow x$
Allowed Final State: Proc 0: $EAX=0 \wedge$ Proc 1: $EBX=0$	

Microarchitecturally, one can view this particular example as a visible consequence of store buffering: if each processor effectively has a FIFO buffer of pending memory writes (to avoid the need to block while a write completes), then the reads from y and x could occur before the writes have propagated from the buffers to main memory. Other families of multiprocessors, dating back at least to the IBM 370, and including ARM, Itanium, POWER, and SPARC, also exhibit relaxed-memory behaviour. Moreover, there are major and subtle differences between different processor families (arising from their different internal design choices): in the details of exactly what non-sequentially consistent executions they permit, and of what memory programmer synchronisation instructions they provide. For any of these processors, relaxed-memory semantics exacerbates the difficulties of writing correct concurrent systems programmers cannot rely on the sequential concept of memory reads and writes. Still, work on verification of such systems.

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

Store buffering (SB)

Initially, $x = y = 0$

$$\begin{array}{l} x := 1 \\ a := y \text{ //0} \end{array} \parallel \begin{array}{l} y := 1 \\ b := x \text{ //0} \end{array}$$

Message passing (MP)

Initially, $x = y = 0$

$$\begin{array}{l} x := 1 \\ y := 1 \end{array} \parallel \begin{array}{l} a := y \text{ //1} \\ b := x \text{ //0} \end{array}$$

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

Store buffering (SB)

Initially, $x = y = 0$

$$\begin{array}{l} x := 1 \\ a := y \text{ //0} \end{array} \parallel \begin{array}{l} y := 1 \\ b := x \text{ //0} \end{array}$$

SC

X

Message passing (MP)

Initially, $x = y = 0$

$$\begin{array}{l} x := 1 \\ y := 1 \end{array} \parallel \begin{array}{l} a := y \text{ //1} \\ b := x \text{ //0} \end{array}$$

X

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



Message passing (MP)

Initially, $x = y = 0$

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





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

Store buffering (SB)

Initially, $x = y = 0$

$$\begin{array}{l} x := 1 \\ a := y \text{ // } 0 \end{array} \parallel \begin{array}{l} y := 1 \\ b := x \text{ // } 0 \end{array}$$

SC

X

TSO/ WB, WT

✓

Message passing (MP)

Initially, $x = y = 0$

$$\begin{array}{l} x := 1 \\ y := 1 \end{array} \parallel \begin{array}{l} a := y \text{ // } 1 \\ b := x \text{ // } 0 \end{array}$$

X

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

Memory Type and Mnemonic	Cacheable	Writeback Cacheable	Allows Speculative Reads	Memory Ordering Model
Write Through (WT)	Yes	No	Yes	Speculative Processor Ordering.
Write Back (WB)	Yes	Yes	Yes	Speculative Processor Ordering.

TSO



WB and WT Memory Types

Table 11-2. Memory Types and Their Properties

Memory Type and Mnemonic	Cacheable	Writeback Cacheable	Allows Speculative Reads	Memory Ordering Model
Write Through (WT)	Yes	No	Yes	Speculative Processor Ordering.
Write Back (WB)	Yes	Yes	Yes	Speculative Processor Ordering.

TSO

applies **only** to **all-WB/ all-WT** accesses, **not mixed** accesses

WB and WT Memory Types

Table 11-2. Memory Types and Their Properties

Memory Type and Mnemonic	Cacheable	Writeback Cacheable	Allows Speculative Reads	Memory Ordering Model
Write Through (WT)	Yes	No	Yes	Speculative Processor Ordering.
Write Back (WB)	Yes	Yes	Yes	Speculative Processor Ordering.

TSO

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.

The extent of
WB/WT Specification
in the Intel manual

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



UC



Message passing (MP)

Initially, $x = y = 0$

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







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

Store buffering (SB)

Initially, $x = y = 0$

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

Message passing (MP)

Initially, $x = y = 0$

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

SC

X

X

TSO/ WB, WT

✓

X

UC

X

X

UC memory is subject to SC consistency semantics:

no reordering

UC Memory Type

Table 1 1-2. Memory Types and Their Properties

Memory Type and Mnemonic	Cacheable	Writeback Cacheable	Allows Speculative Reads	Memory Ordering Model
Strong Uncacheable (UC)	No	No	No	Strong Ordering ← SC

UC Memory Type

Table 1 1-2. Memory Types and Their Properties

Memory Type and Mnemonic	Cacheable	Writeback Cacheable	Allows Speculative Reads	Memory Ordering Model
Strong Uncacheable (UC)	No	No	No	Strong Ordering ← SC

applies **only** to **all-UC** accesses, **not mixed** accesses

UC Memory Type

Table 11-2. Memory Types and Their Properties

Memory Type and Mnemonic	Cacheable	Writeback Cacheable	Allows Speculative Reads	Memory Ordering Model
Strong Uncacheable (UC)	No	No	No	Strong Ordering ← SC

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.

The extent of UC Specification in the Intel manual

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

Store buffering (SB)

Initially, $x = y = 0$

$$\begin{array}{l} x := 1 \\ a := y \text{ // } 0 \end{array} \parallel \begin{array}{l} y := 1 \\ b := x \text{ // } 0 \end{array}$$

SC



TSO/ WB, WT



UC



WC



Message passing (MP)

Initially, $x = y = 0$

$$\begin{array}{l} x := 1 \\ y := 1 \end{array} \parallel \begin{array}{l} a := y \text{ // } 1 \\ b := x \text{ // } 0 \end{array}$$









WC memory: **write-write reordering** on different locations

WC Memory Type

Table 11-2. Memory Types and Their Properties

Memory Type and Mnemonic	Cacheable	Writeback Cacheable	Allows Speculative Reads	Memory Ordering Model
Write Combining (WC)	No	No	Yes	Weak Ordering. Available by programming MTRRs or by selecting it through the PAT.

write-write reordering on different locations

WC Memory Type

Table 1 1-2. Memory Types and Their Properties

Memory Type and Mnemonic	Cacheable	Writeback Cacheable	Allows Speculative Reads	Memory Ordering Model
Write Combining (WC)	No	No	Yes	Weak Ordering. Available by programming MTRRs or by selecting it through the PAT.

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

Memory Type and Mnemonic	Cacheable	Writeback Cacheable	Allows Speculative Reads	Memory Ordering Model
Write Combining (WC)	No	No	Yes	Weak Ordering. Available by programming MTRRs or by selecting it through the PAT.

write-write reordering on different locations

applies only to all-WC accesses, not mixed accesses

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, CUID 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.

The extent of WC Specification in the Intel manual

What about ***Non-temporal Stores***?

Intel Manual: 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,

Intel Manual: 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***

Intel Manual: 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***

But...

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/ WB, WT

✓

UC

X

WC

X

MOVNT

✓

Message passing (MP)

Initially, $x = y = 0$

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

X

X

X

✓

✓

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

Store buffering (SB)

Initially, $x = y = 0$

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

Message passing (MP)

Initially, $x = y = 0$

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

SC



TSO/ WB, WT



UC



WC



MOVNT



← WC & NT stores
have different semantics



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

Store buffering (SB)

Message passing (MP)

Initially $x = y = 0$

Initially $x = y = 0$

//1

//0

Solution

Validate the Ex86 Consistency Semantics!

TSO/ WB,

WC

MOVNT



← 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. corel5, corel6 and Xeon
- ❖ Ran each test **at least 6×10^8 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. corel5, corel6 and Xeon
- ❖ Ran each test **at least 6×10^8 times**; ran critical tests up to a few billion times
- ❖ For more details see: <http://diy.inria.fr/x86-memtype>

Ex86 Semantics: Preserved Ordering

Earlier in Program Order	Later in Program Order							
	$R_{wb,wt}$	$R_{uc,wc}$	W_{wb}	$W_{uc,wt}$	$W_{wc,nt}$	U,MF,SF	FL	FO
	R	✓	✓	✓	✓	✓	✓	✓
	W_{wb}	✗	✓	✓	sloc	✓	✓	scl
	$W_{wt,uc}$	✗	✓	✓	✓	✓	✓	✓
	$W_{wc,nt}$	✗	✓	sloc	sloc	✓	✓	scl
	U,MF	✓	✓	✓	✓	✓	✓	✓
	SF	✗	✓†	✓	✓	✓	✓	✓
	FL	✗	✓	✓	✓	✓	✓	✗
	FO	✗	✓	✗	✓	✗	✗	✗

✓ 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

Earlier in Program Order	Later in Program Order							
	$R_{wb,wt}$	$R_{uc,wc}$	W_{wb}	$W_{uc,wt}$	$W_{wc,nt}$	U,MF,SF	FL	FO
	R	✓	✓	✓	✓	✓	✓	✓
	W_{wb}	✗	✓	✓	sloc	✓	✓	scl
	$W_{wt,uc}$	✗	✓	✓	✓	✓	✓	✓
	$W_{wc,nt}$	✗	✓	sloc	✓	sloc	✓	scl
	U,MF	✓	✓	✓	✓	✓	✓	✓
	SF	✗	✓†	✓	✓	✓	✓	✓
	FL	✗	✓	✓	✓	✓	✓	✗
	FO	✗	✓	✗	✓	✗	✗	✗

✓ 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

Earlier in Program Order	Later in Program Order							
	$R_{wb,wt}$	$R_{uc,wc}$	W_{wb}	$W_{uc,wt}$	$W_{wc,nt}$	U,MF,SF	FL	FO
	R	✓	✓	✓	✓	✓	✓	✓
	W_{wb}	✗	✓	✓	sloc	✓	✓	scl
	$W_{wt,uc}$	✗	✓	✓	✓	✓	✓	✓
	$W_{wc,nt}$	✗	✓	sloc	✓	sloc	✓	scl
	U,MF	✓	✓	✓	✓	✓	✓	✓
	SF	✗	✓†	✓	✓	✓	✓	✓
	FL	✗	✓	✓	✓	✓	✓	✗
	FO	✗	✓	✗	✓	✗	✗	✗

✓ 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

		Later in Program Order							
Earlier in Program Order		$R_{wb,wt}$	$R_{uc,wc}$	W_{wb}	$W_{uc,wt}$	$W_{wc,nt}$	U,MF,SF	FL	FO
	R	✓	✓	✓	✓	✓	✓	✓	✓
	W_{wb}	✗	✓	✓	✓	sloc	✓	✓	scl
	$W_{wt,uc}$	✗	✓	✓	✓	✓	✓	✓	✓
	$W_{wc,nt}$	✗	✓	sloc	✓	sloc	✓	✓	scl
	U,MF	✓	✓	✓	✓	✓	✓	✓	✓
	SF	✗	✓†	✓	✓	✓	✓	✓	✓
	FL	✗	✓	✓	✓	✓	✓	✓	✗
	FO	✗	✓	✗	✓	✗	✓	✗	✗

✓ 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

Earlier in Program Order	Later in Program Order							
	$R_{wb,wt}$	$R_{uc,wc}$	W_{wb}	$W_{uc,wt}$	$W_{wc,nt}$	U,MF,SF	FL	FO
	R	✓	✓	✓	✓	✓	✓	✓
	W_{wb}	✗	✓	✓	sloc	✓	✓	scl
	$W_{wt,uc}$	✗	✓	✓	✓	✓	✓	✓
	$W_{wc,nt}$	✗	✓	sloc	sloc	✓	✓	scl
	U,MF	✓	✓	✓	✓	✓	✓	✓
	SF	✗	✓†	✓	✓	✓	✓	✓
	FL	✗	✓	✓	✓	✓	✓	✗
	FO	✗	✓	✗	✗	✓	✗	✗

✓ 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

Earlier in Program Order	Later in Program Order								
	$R_{wb,wt}$	$R_{uc,wc}$	W_{wb}	$W_{uc,wt}$	$W_{wc,nt}$	U,MF,SF	FL	FO	
	R	✓	✓	✓	✓	✓	✓	✓	
	W_{wb}	✗	✓	✓	✓	sloc	✓	scl	
	$W_{wt,uc}$	✗	✓	✓	✓	✓	✓	✓	
	$W_{wc,nt}$	✗	✓	sloc	✓	sloc	✓	scl	
	U,MF	✓	✓	✓	✓	✓	✓	✓	
	SF	✗	✓†	✓	✓	✓	✓	✓	
	FL	✗	✓	✓	✓	✓	✓	✗	
	FO	✗	✓	✗	✓	✗	✓	✗	

✓ Order preserved;
may not be reordered

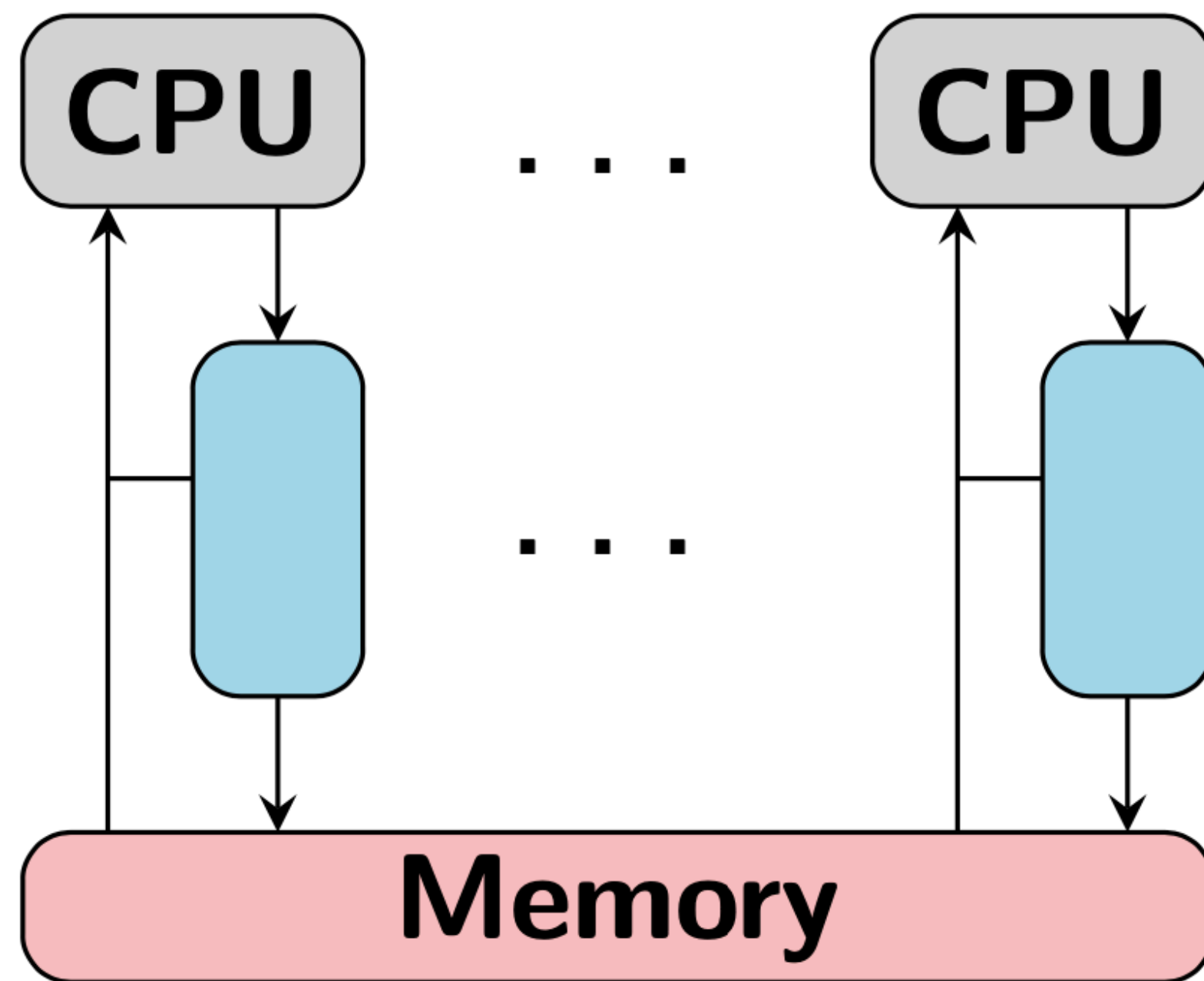
sloc: Order preserved iff
on the same location

scl: Order preserved iff
on the same cache line

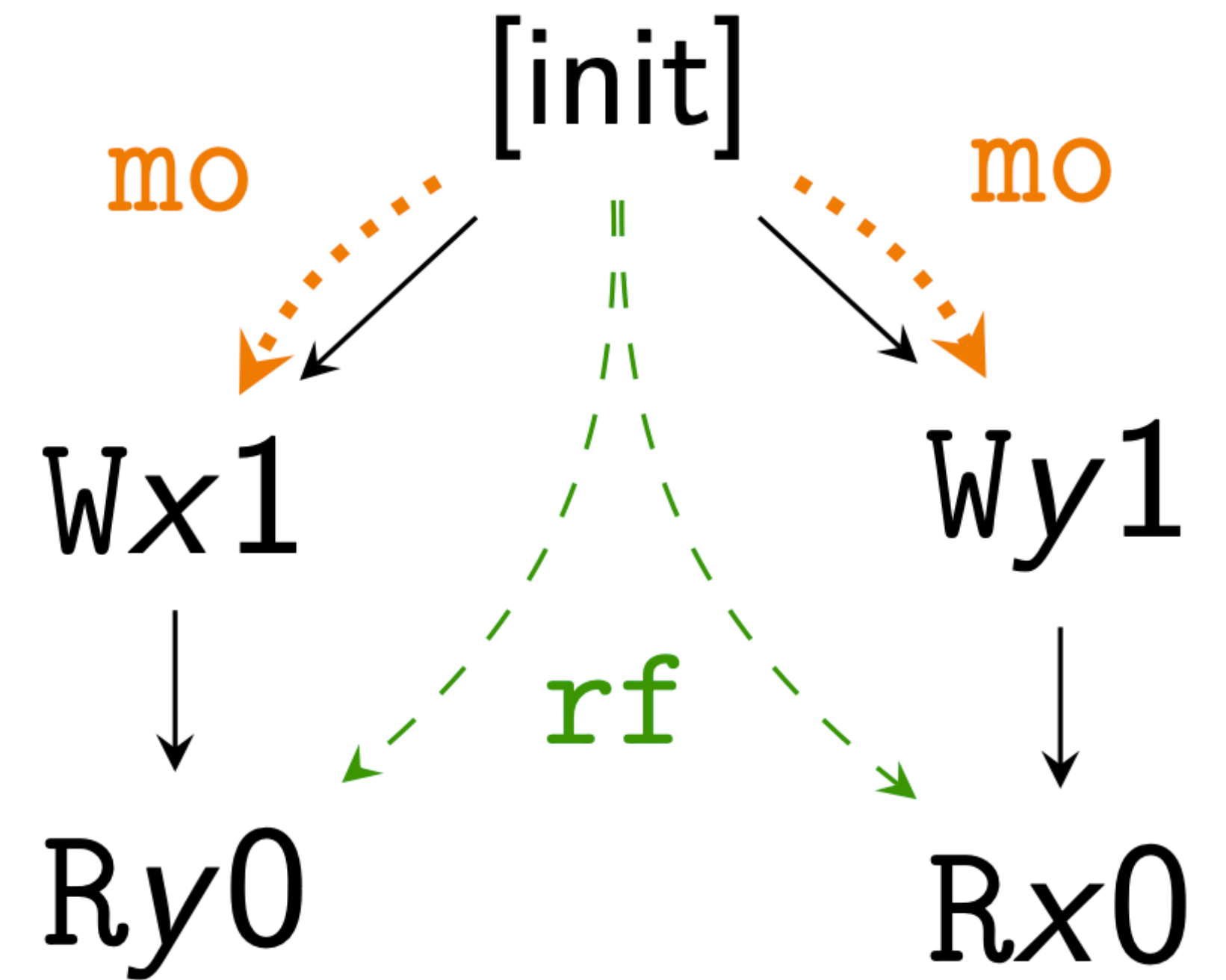
✗ Order not preserved
may be reordered

Ex86 Semantics: Two ***Equivalent*** Models

Operational Ex86



Declarative Ex86



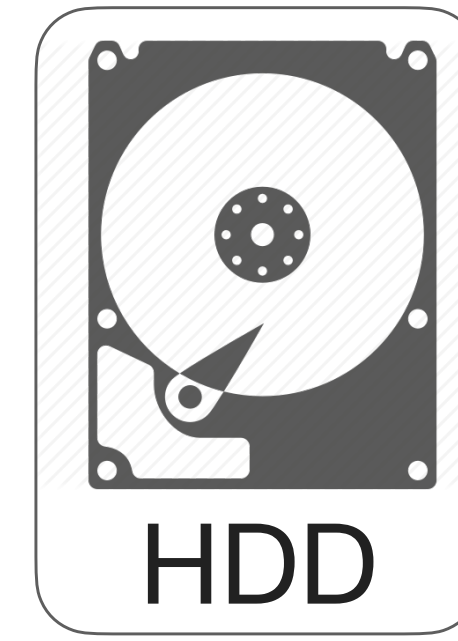
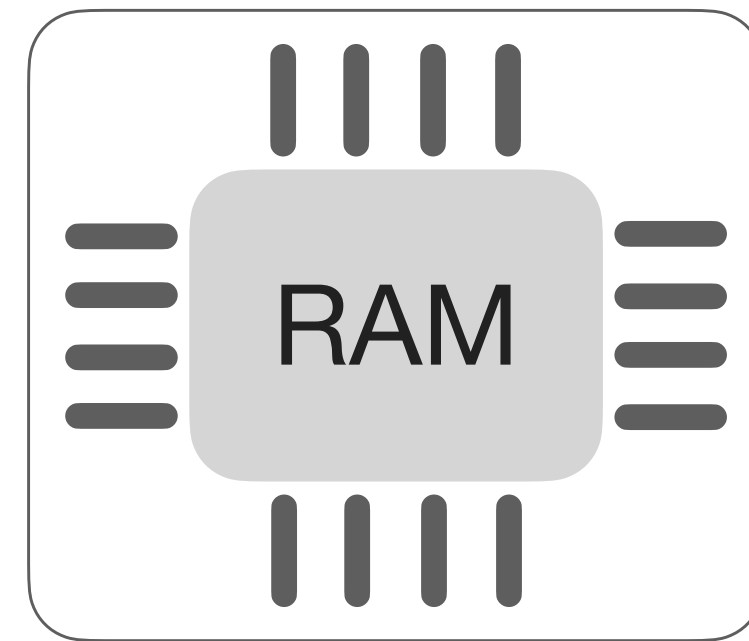
Proved the ***equivalence*** of the two models

What about Intel-x86 ***Persistency*** Semantics?

Computer Storage

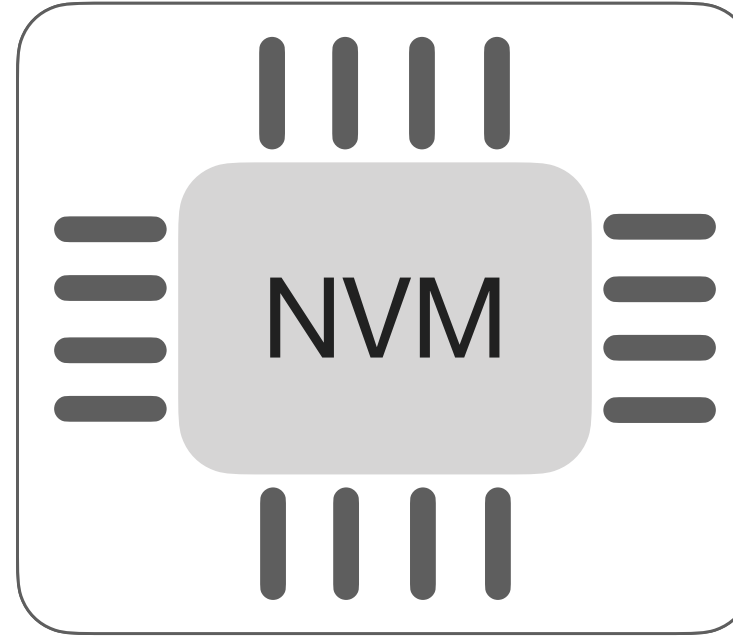


✓ *fast*
✗ *volatile*



✗ *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?

```
// 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 ;
```



```
// 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

Persistency Model

the **order** in which writes
are **persisted** to NVM

Full Semantics

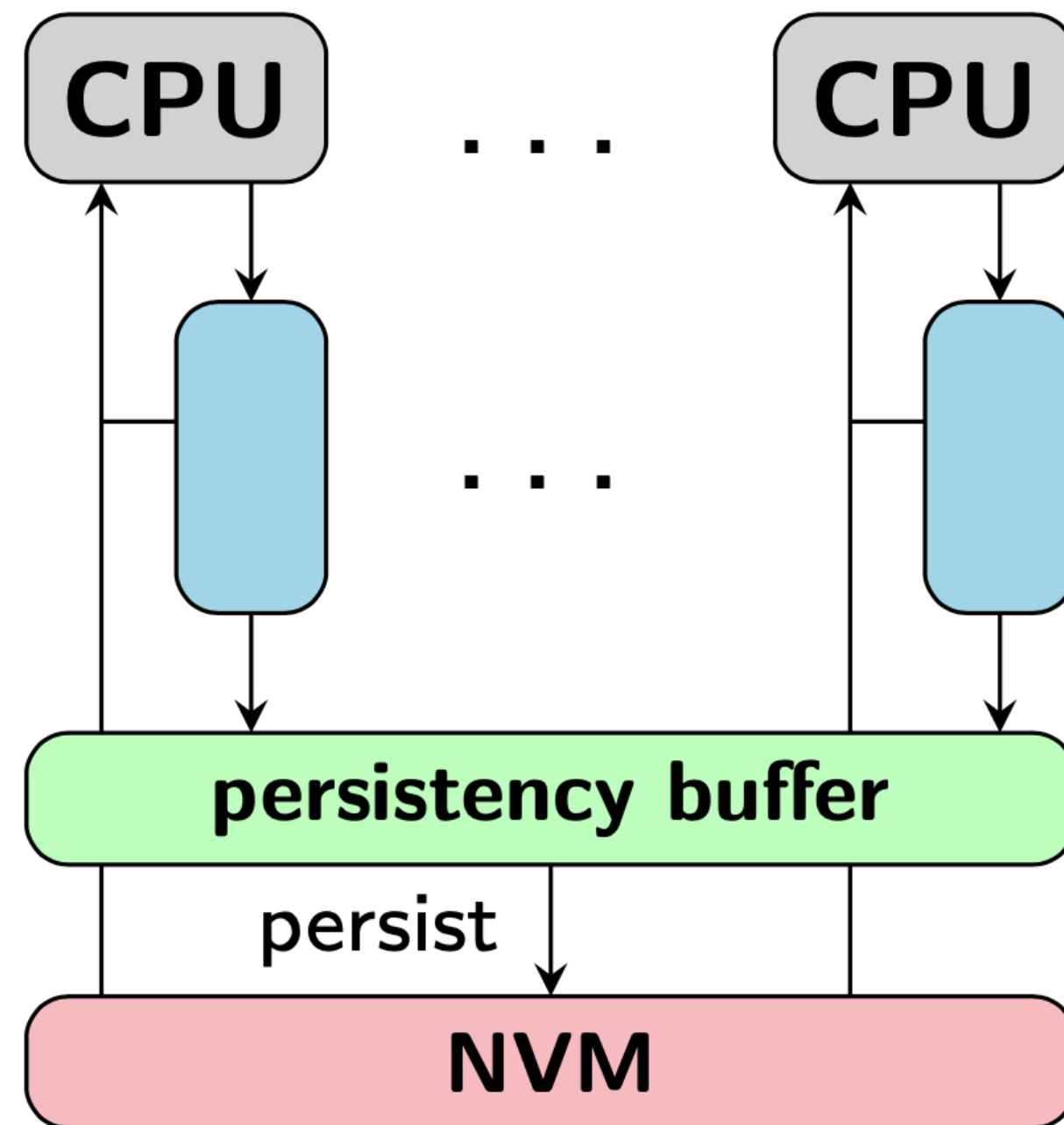
Consistency + Persistency Model

PEx86 (Persistent Extended x86):

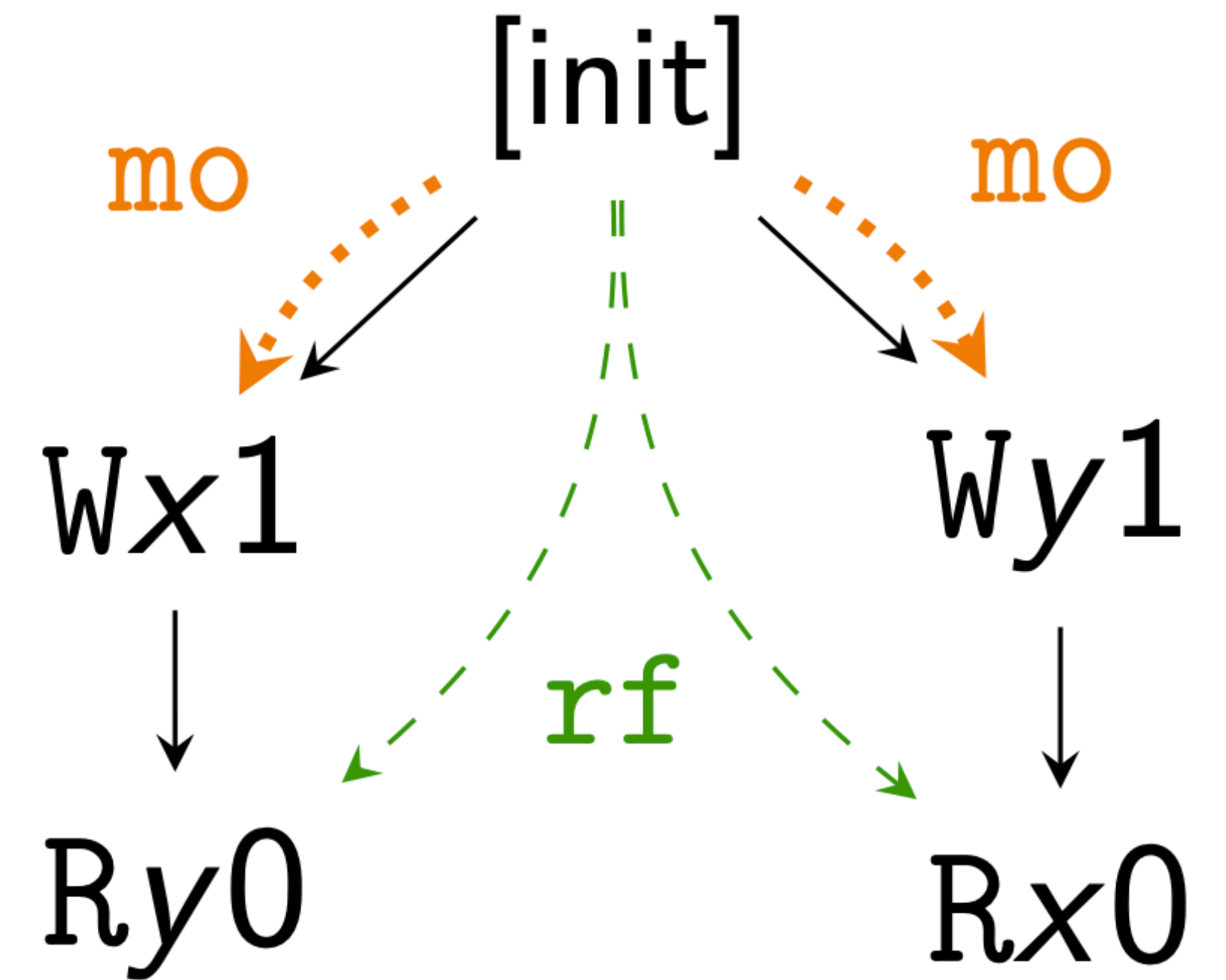
Formal **consistency + Persistency** semantics of
Intel-x86 architectures
including
non-temporal stores & memory types

PEx86 Semantics: Two ***Equivalent*** Models

Operational PEx86



Declarative PEx86



Proved the ***equivalence*** of the two models

PEx86: Persistent Extended Intel-x86 Semantics

$x, y \in \text{Loc}_{\text{wb}}$	$x, x', y \in \text{Loc}_{\text{wb}}$	$x, x', y \in \text{Loc}_{\text{wb}}$	$x, x', y \in \text{Loc}_{\text{wb}}$	$x, x', y \in \text{Loc}_{\text{wb}}$
$x := 1$ $y := 1$	$x := 1$ clflush x' $y := 1$	$x := 1$ clflushopt x' $y := 1$	$x := 1$ clflushopt x' xchg ($y, 1$)	$x := 1$ clflushopt x' ; sfence $y := 1$
rec: $x, y \in \{0, 1\}$	rec: $y = 1 \Rightarrow x = 1$	rec: $x, y \in \{0, 1\}$	rec: $y = 1 \Rightarrow x = 1$	rec: $y = 1 \Rightarrow x = 1$

$x \in \text{Loc}_{\text{uc}} \cup \text{wt}$ $y \in \text{Loc}$	$x \in \text{Loc}_{\text{wc}},$ $y \in \text{Loc}_{\text{wc}} \cup \text{wb}$	$x \in \text{Loc}_{\text{wc}},$ $y \in \text{Loc}_{\text{uc}} \cup \text{wt}$	$x \in \text{Loc}_{\text{wb}} \cup \text{wt} \cup \text{wc}$ $y \in \text{Loc}_{\text{uc}} \cup \text{wt}$	$x \in \text{Loc}_{\text{wb}} \cup \text{wt} \cup \text{wc}$ $y \in \text{Loc}_{\text{wc}} \cup \text{wb}$
$x := 1$ $y := 1$	$x := 1$ $y := 1$	$x := 1$ $y := 1$	$x := 1$ $x :=_{\text{NT}} 2$ $y := 1$	$x := 1$ $x :=_{\text{NT}} 2$ 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: $y = 1 \Rightarrow x = 2$

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 **persistence** 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 **persistence** 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 **persistence** 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!