

Persistency Semantics of the Intel-x86 Architecture

Azalea Raad^{1,2} John Wickerson² Gil Neiger³ Viktor Vafeiadis¹

¹ Max Planck Institute for Software Systems (MPI-SWS)

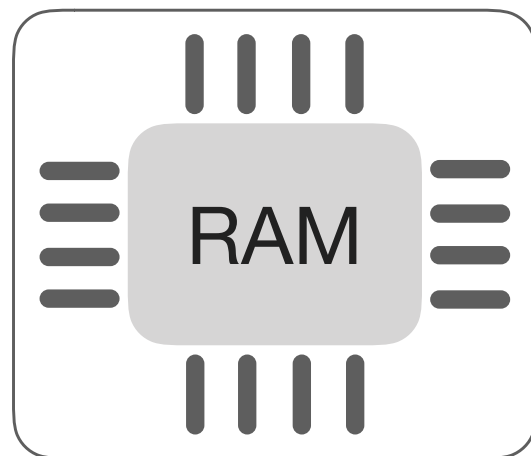
² Imperial College London

³ Intel Corporation

Computer Storage

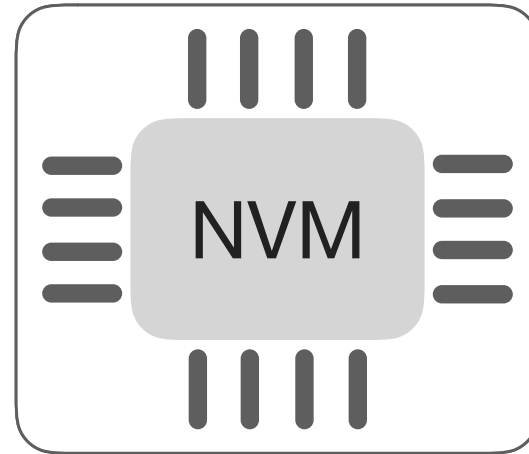


✓ *fast*
✗ *volatile*



✗ *slow*
✓ *persistent*

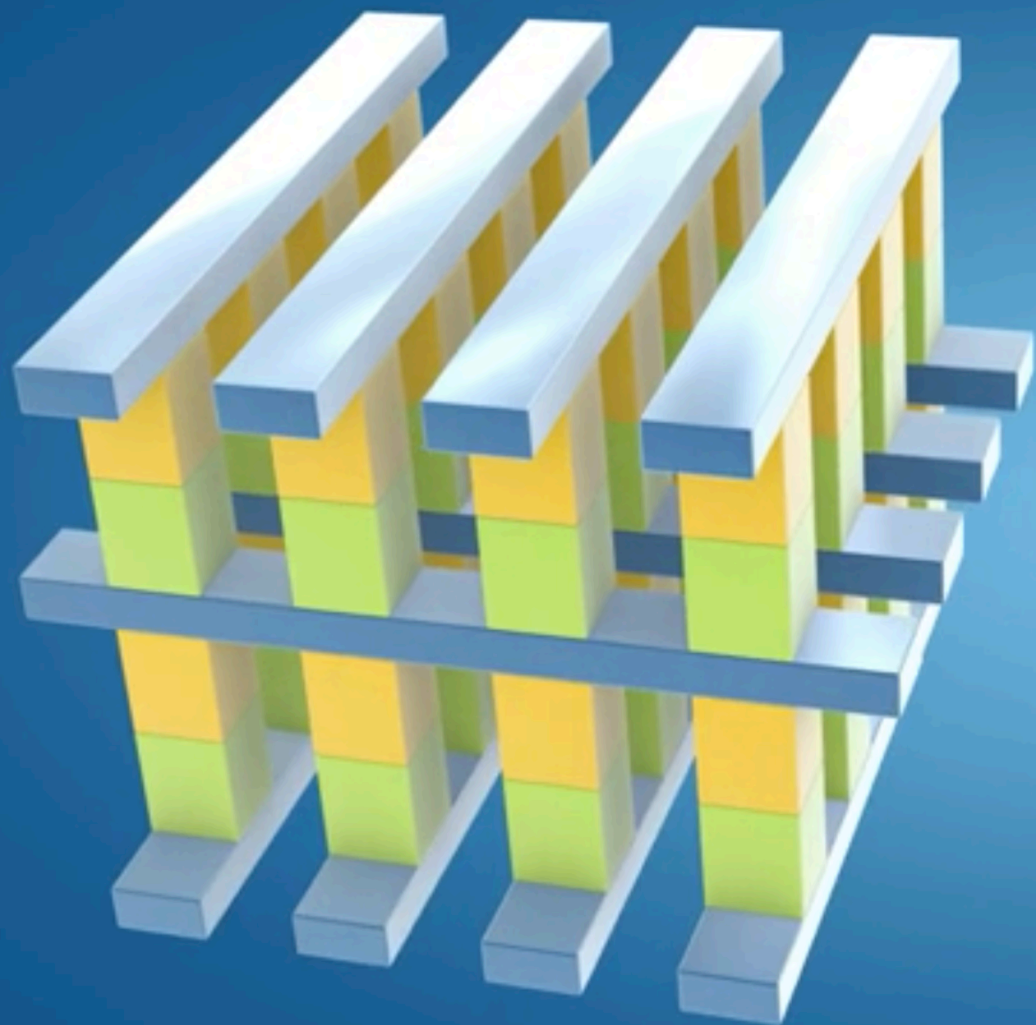
What is Non-Volatile Memory (NVM)?



NVM: Hybrid Storage + Memory

Best of both worlds:

- ✓ ***persistent*** (like HDD)
- ✓ ***fast, random access*** (like RAM)



INTEL® OPTANE™ TECHNOLOGY



FAST




DENSE




NON-VOLATILE

Q: Why *Formal* NVM Semantics?

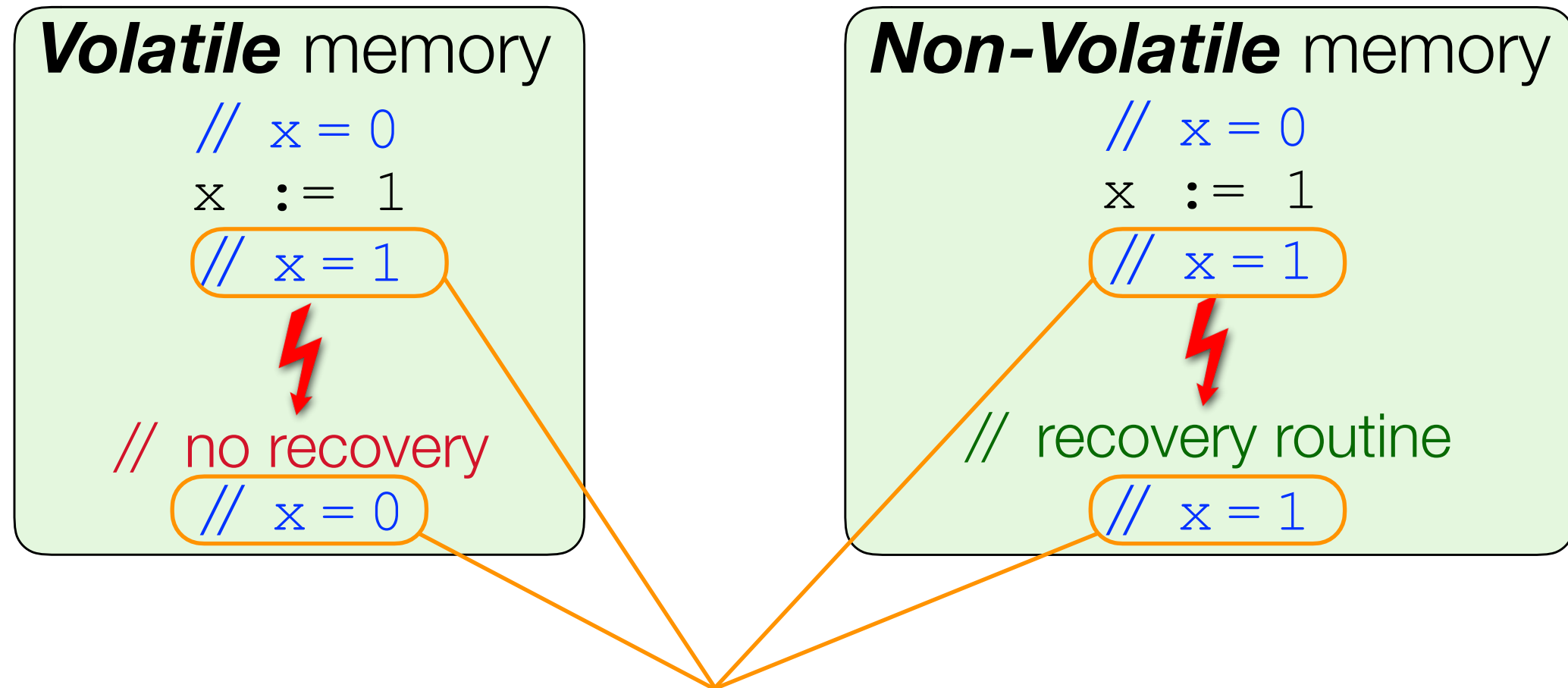
Volatile memory

```
// x = 0  
x := 1  
// x = 1  
  
// no recovery  
// x = 0
```

Non-Volatile memory

```
// x = 0  
x := 1  
// x = 1  
  
// recovery routine  
// x = 1
```

Q: Why *Formal* NVM Semantics?



A: Program *Verification*

Q: Why ***Formal*** NVM Semantics?

What about ***Concurrency***?

// $x = y = \dots = 0$

$C_1 \parallel C_2 \parallel \dots \parallel C_n$

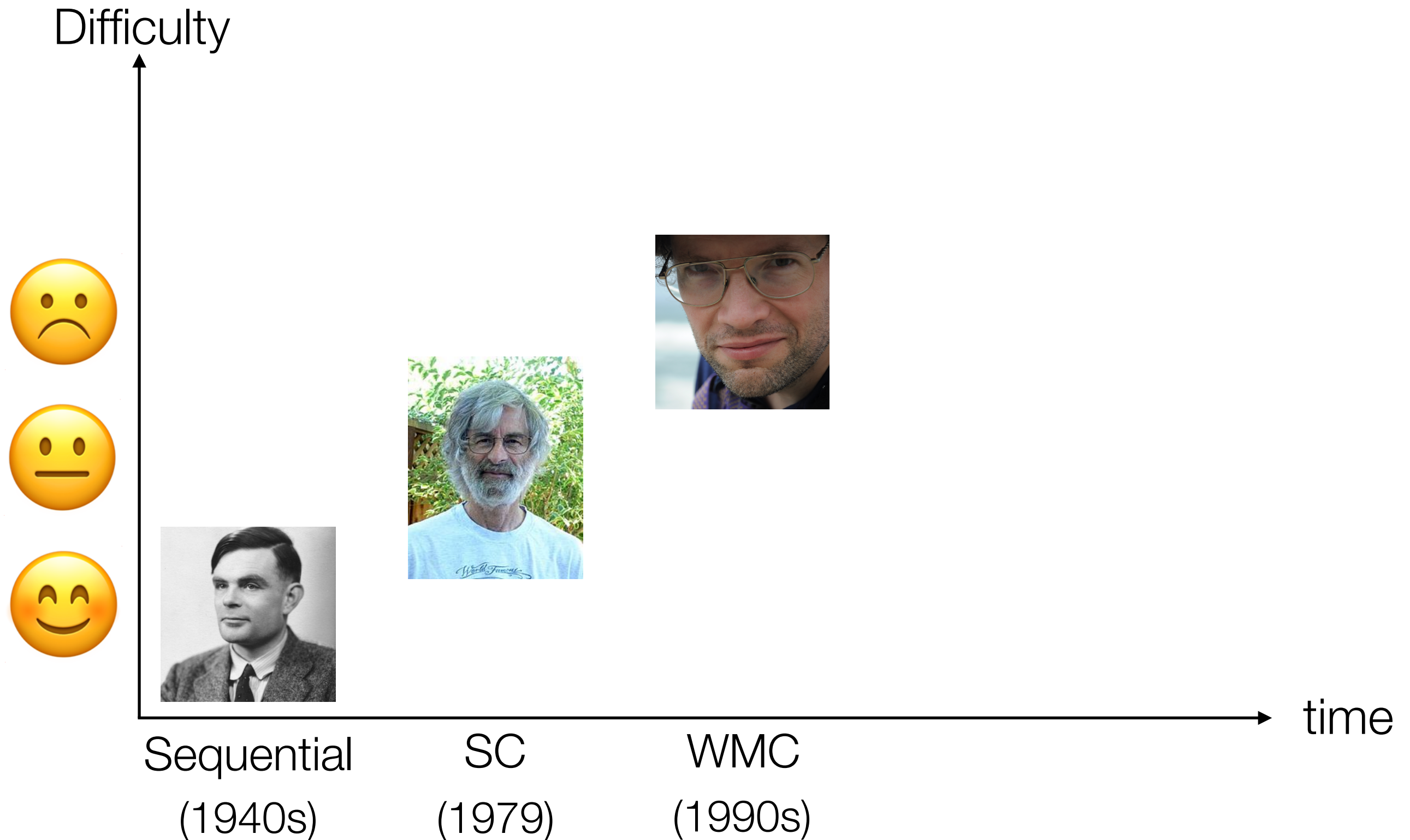
// ???



// recovery routine

// ???

Formal Semantic Models



Weak Memory Consistency (WMC)

No total execution order (*to*) \Rightarrow

weak behaviour absent under SC, caused by:

- instruction **reordering** by compiler
- write propagation across **cache hierarchy**

Weak Memory Consistency (WMC)

No total execution order (*to*) \Rightarrow

weak be

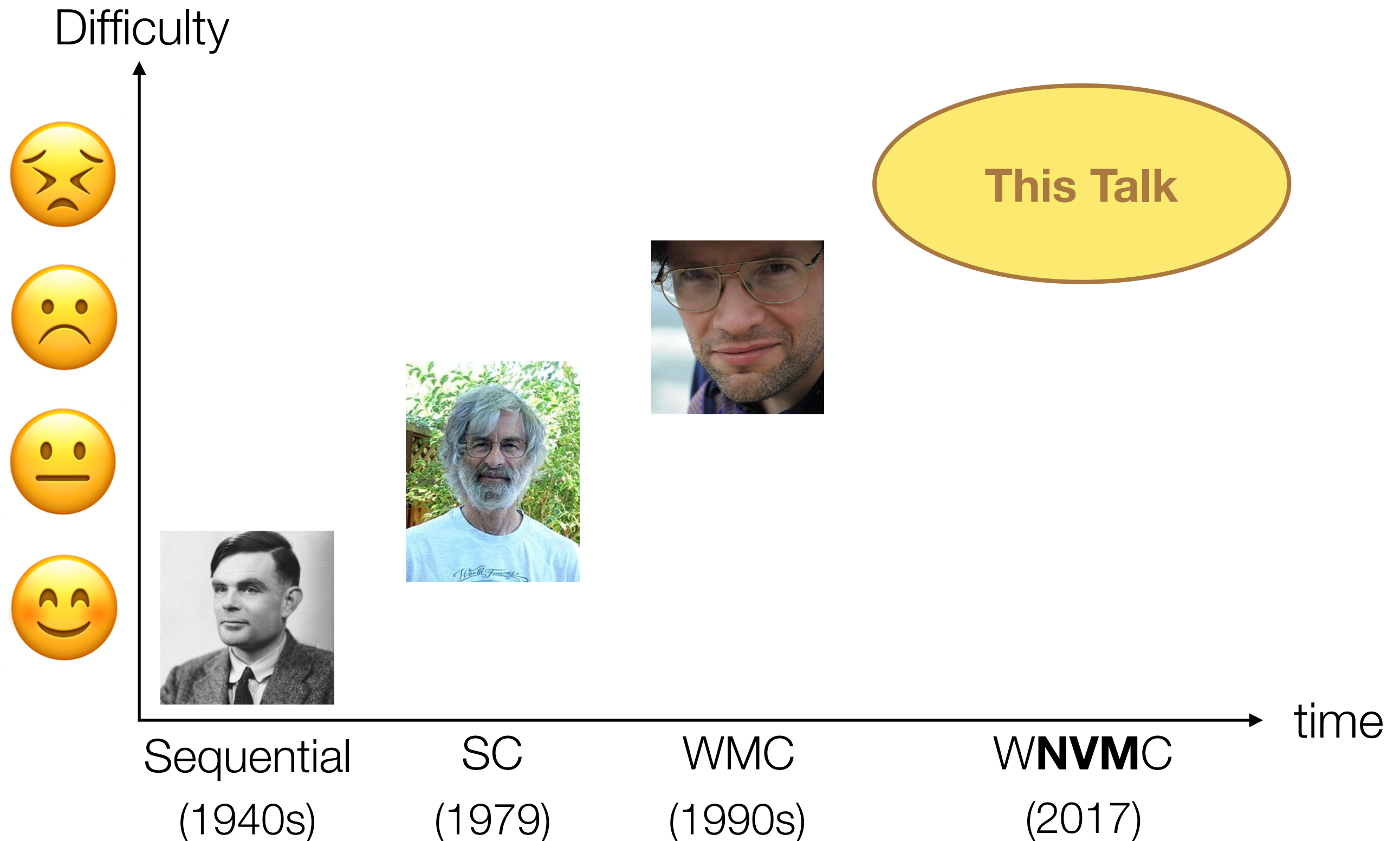
- instr
- write

Consistency Model

the **order** in which
writes are made visible
to other threads

e.g. x86 (TSO), ARMv8, C11, Java

Formal Semantic Models



What Can Go Wrong?

```
// x=y=0
```

```
x := 1;
```

```
y := 1;
```



```
// recovery routine
```

```
// x=y=1 OR x=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***

— ***relaxed*** persists

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

NVM Semantics

Consistency + Persistency Model

// x=

!! Ex

—

!! W

—

y=1

This Talk

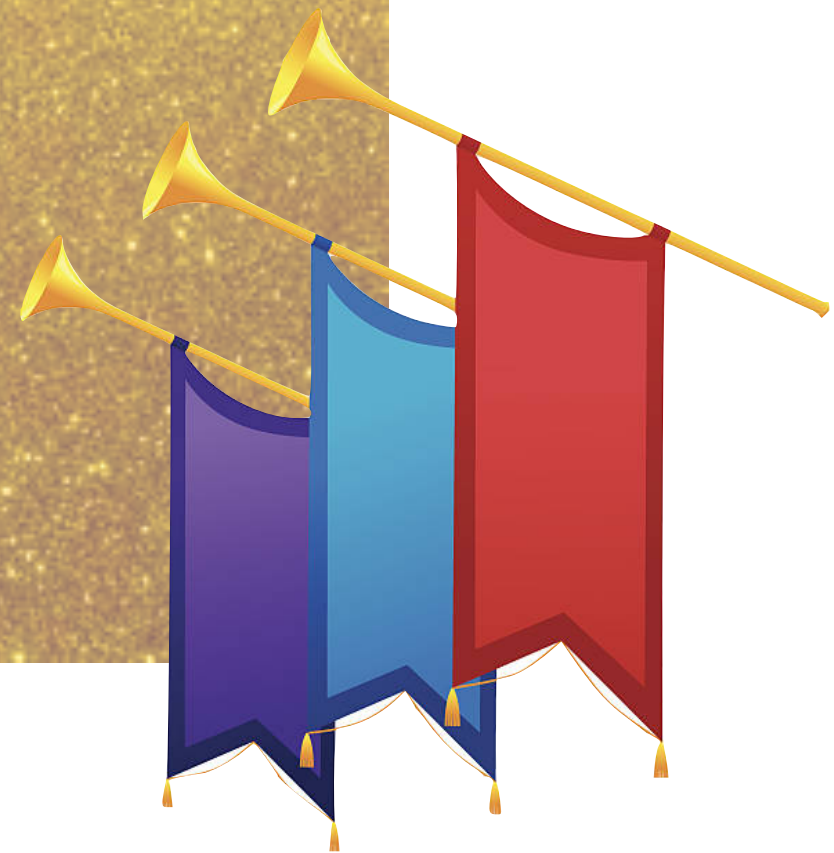
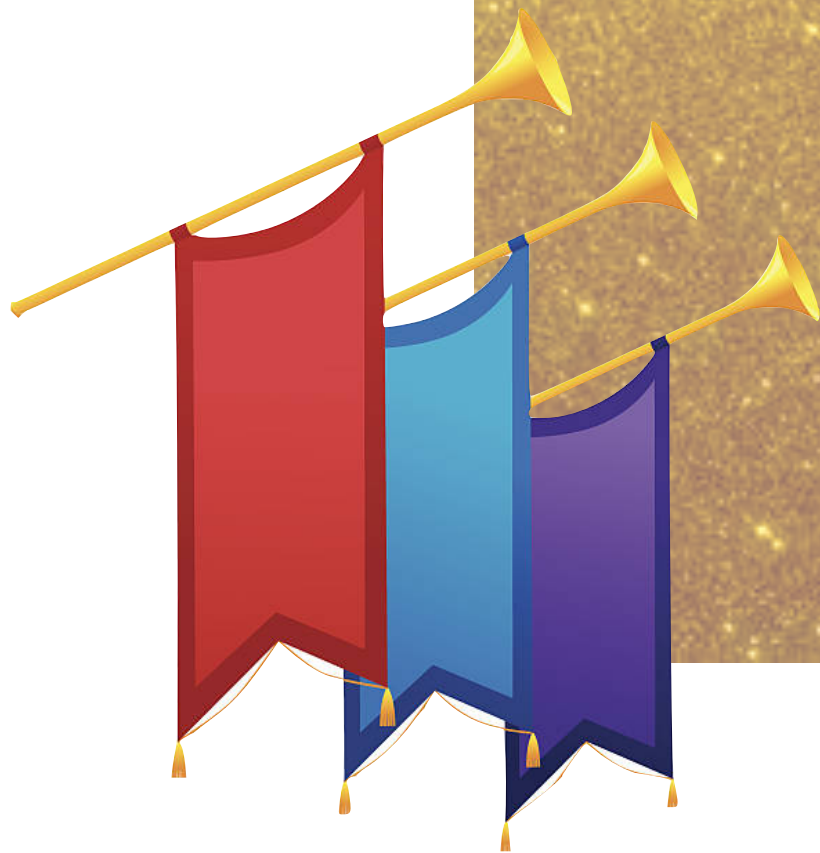
Px86

(Persistent x86):

NVM Semantics

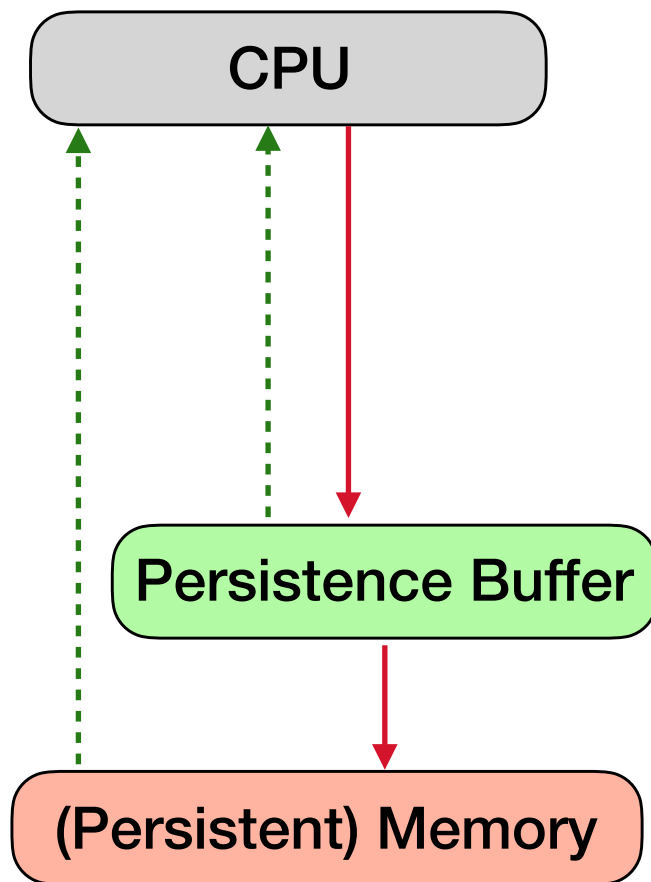
of the

x86 Architecture



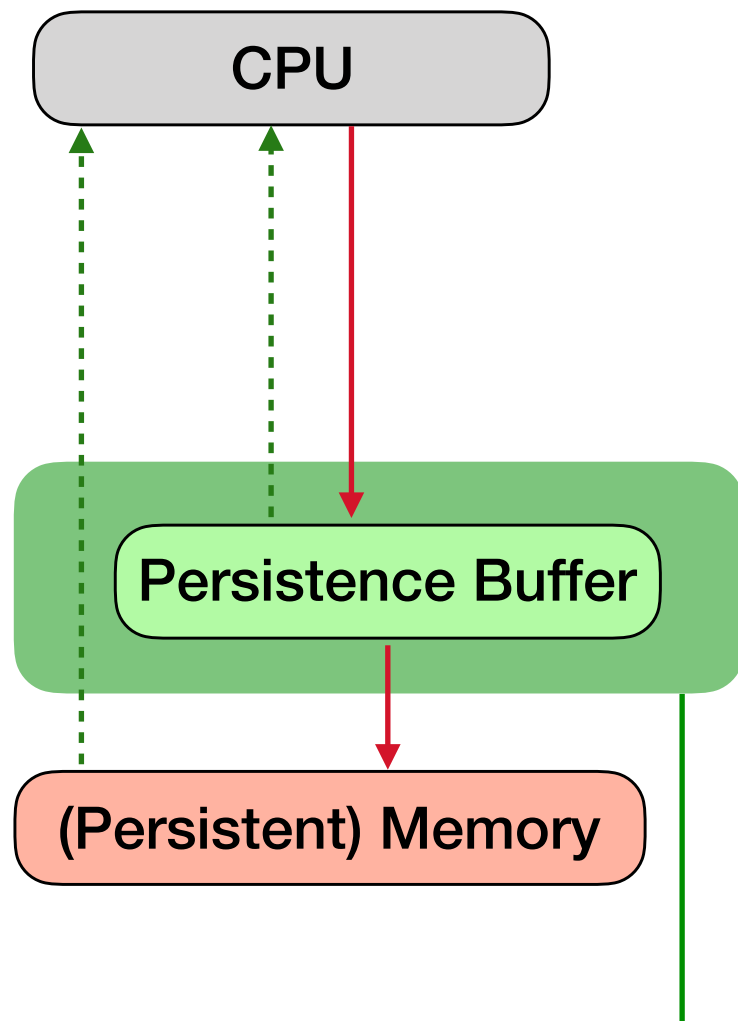
Warmup:
Sequential P_x86

x86: (Sequential) Persistent Hardware Model



$x := 1$: adds $x := 1$ to p-buffer

x86: (Sequential) Persistent Hardware Model

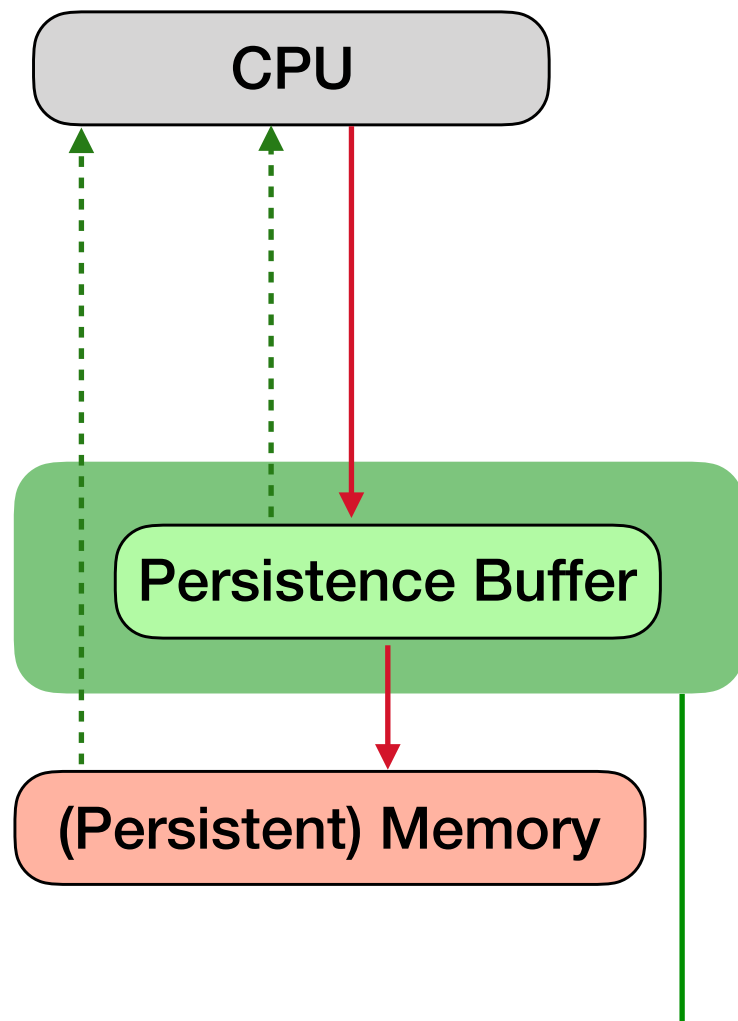


$x := 1$: adds $x := 1$ to p-buffer

unbuffer* : p-buffer to memory

Unbuffered at *non-deterministic* points in time!

x86: (Sequential) Persistent Hardware Model



$x := 1$: adds $x := 1$ to p-buffer

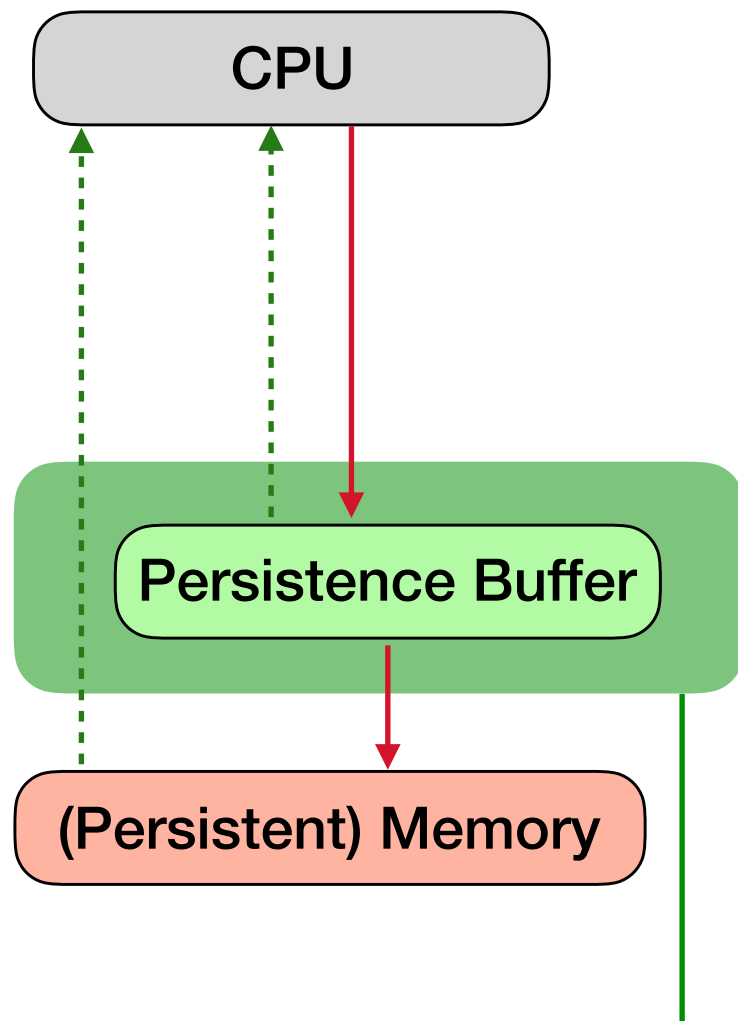
unbuffer* : p-buffer to memory

Unbuffered at *non-deterministic* points in time!

Buffering & unbuffering orders may disagree!

* at non-deterministic times

x86: (Sequential) Persistent Hardware Model



$x := 1$: adds $x := 1$ to p-buffer

unbuffer* : p-buffer to memory

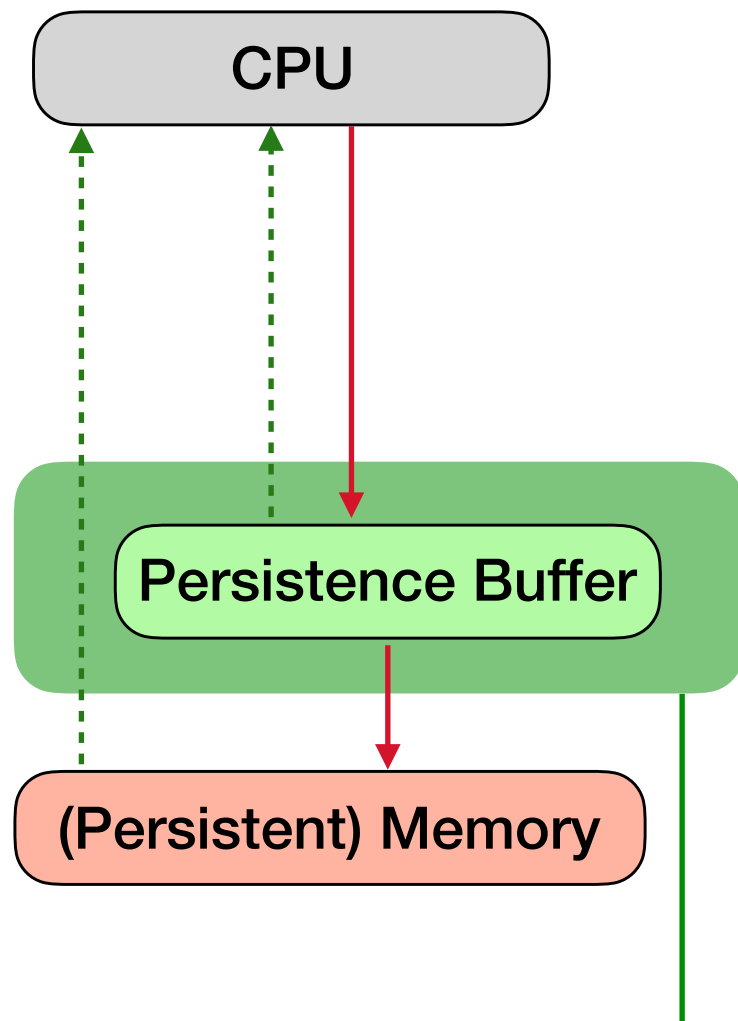
$a := x$: if p-buffer contains x , reads latest entry
else reads from memory

Unbuffered at *non-deterministic* points in time!

Buffering & unbuffering orders may disagree!

* at non-deterministic times

x86: (Sequential) Persistent Hardware Model



$x := 1$: adds $x := 1$ to p-buffer

unbuffer* : p-buffer to memory

$a := x$: if p-buffer contains x , reads latest entry
else reads from memory



p-buffer lost; memory **retained**

Unbuffered at *non-deterministic* points in time!

Buffering & unbuffering orders may disagree!

* at non-deterministic times

Fixing *Relaxed* Persists: Attempt #1

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

```
x := 1;
```

```
y := 1;
```





```
// recovery routine
```

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

!! out of order persists

👉 ***persist barriers?***

Persist Barriers: *Desiderata*

```
// x=0; y=0  
  
x := 1;  
 _____  
y := 1;  
  
  
  
// recovery routine  
  
// x=1; y=1 OR x=0; y=0 OR x=1; y=0 OR x=0; y=1
```

!! out of order persists

 ***persist barriers?***

Persist Barriers: *Desiderata*

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

x86

does not provide
persist barriers!

x86 memory barriers
(e.g. ***sfence***, ***mfence***)
do not enforce
persist ordering!

```
// x=1;
```

```
z=0; y=1
```

!! out of

➡ pers

Fixing *Relaxed* Persists: Attempt #2

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

```
x := 1;
```

```
y := 1;
```



```
// recovery routine
```

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

!! out of order persists

👉 explicit persists?



Explicit Persists: *Desiderata*

```
// x=0; y=0
x := 1;
➡ persist x;
y := 1;
      ⚡
// recovery routine
// x=1; y=1 OR x=0; y=0 OR x=1; y=0 OR x=0; y=1
```

!! out of order persists

➡ **explicit persists?**

Explicit Persists: *Reality on x86*

```
        // x=0; y=0
        x := 1;
         clwb x;
        y := 1;
        
        // recovery routine
// x=1; y=1 OR x=0; y=0 OR x=1; y=0 OR x=0; y=1
```

!! out of order persists

 **explicit persists?**

clwb x/clflushopt x/clflush x:

asynchronously persist cache line containing **x**

Explicit Persists: *Reality on x86*

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

```
x := 1;
```

➡ `clwb x;`

```
y := 1;
```

```
// x=1; y=
```

```
x=0; y=1
```

x86 explicit persists
are
asynchronous
and can themselves
persist out of order !

!! out of order persists

➡ ***explicit persists?***

```
clwb x/clflushopt x/clflush x:
```

asynchronously persist cache line containing **x**

Solution: *Persist Sequence*

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

```
x := 1;
```

```
clwb x;
```

```
s_fence;
```

```
y := 1;
```



```
// recovery routine
```

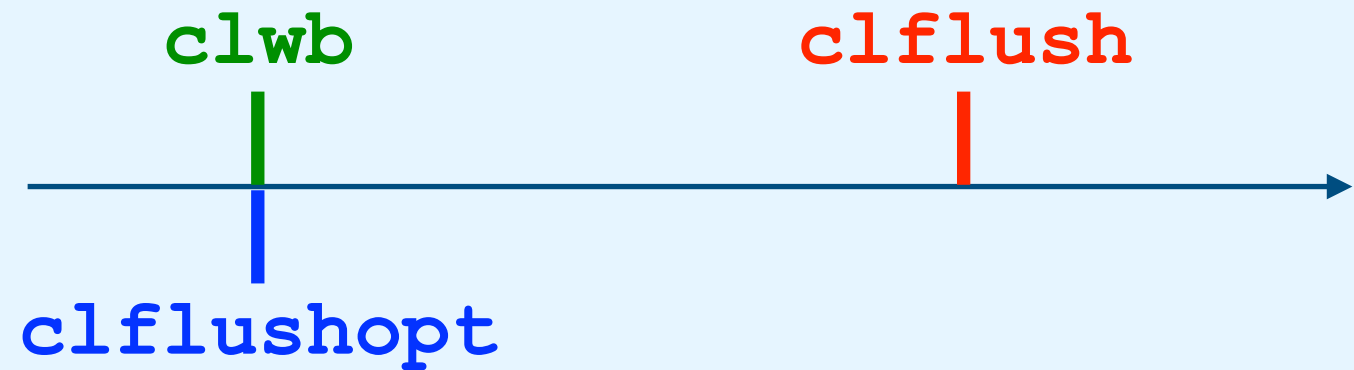
```
// x=1; y=1 OR x=0; y=0 OR x=1; y=0 OR x=0; y=1
```

- ❖ **Waits** until earlier writes on **x** are persisted
- ❖ **Disallows reordering**

- ✓ **synchronous** persists
- ✓ **no out of order** persists

x86 Persists: **clwb**, **clflushopt**, **clflush**

Strength
(ordering constraints)



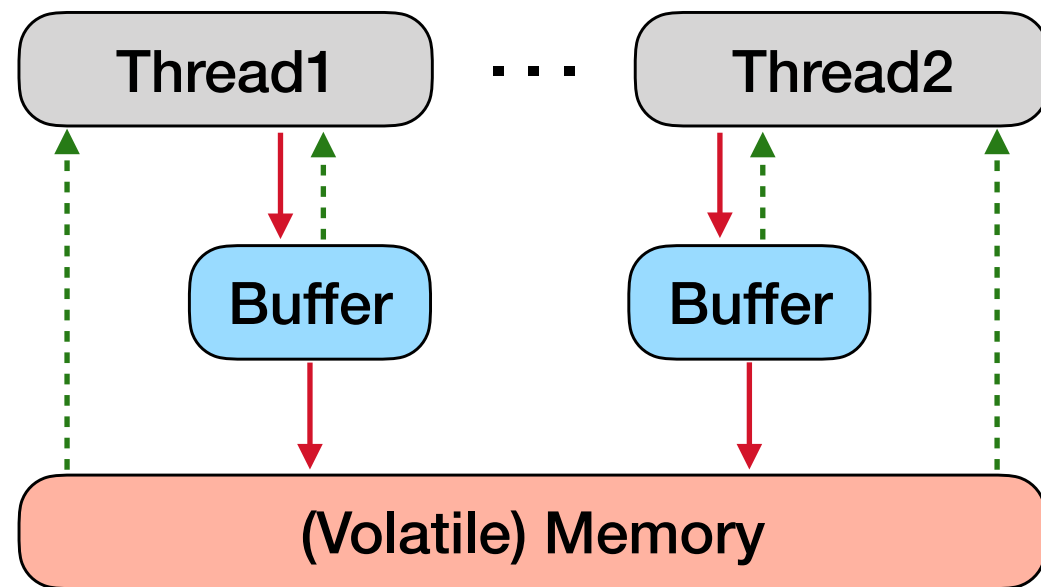
Performance



- ❖ **clwb** and **clflushopt**: **same ordering** constraints
- ❖ **clwb** **does not invalidate** cache line
- ❖ **clflushopt** **invalidates** cache line
- ❖ **clflush**: **strongest** ordering constraints; **invalidates** cache line

Concurrent Px86

x86: (Volatile) Concurrent Hardware Model (TSO)



$x := 1$: adds $x := 1$ to **buffer**

unbuffer* : **buffer** to **memory**

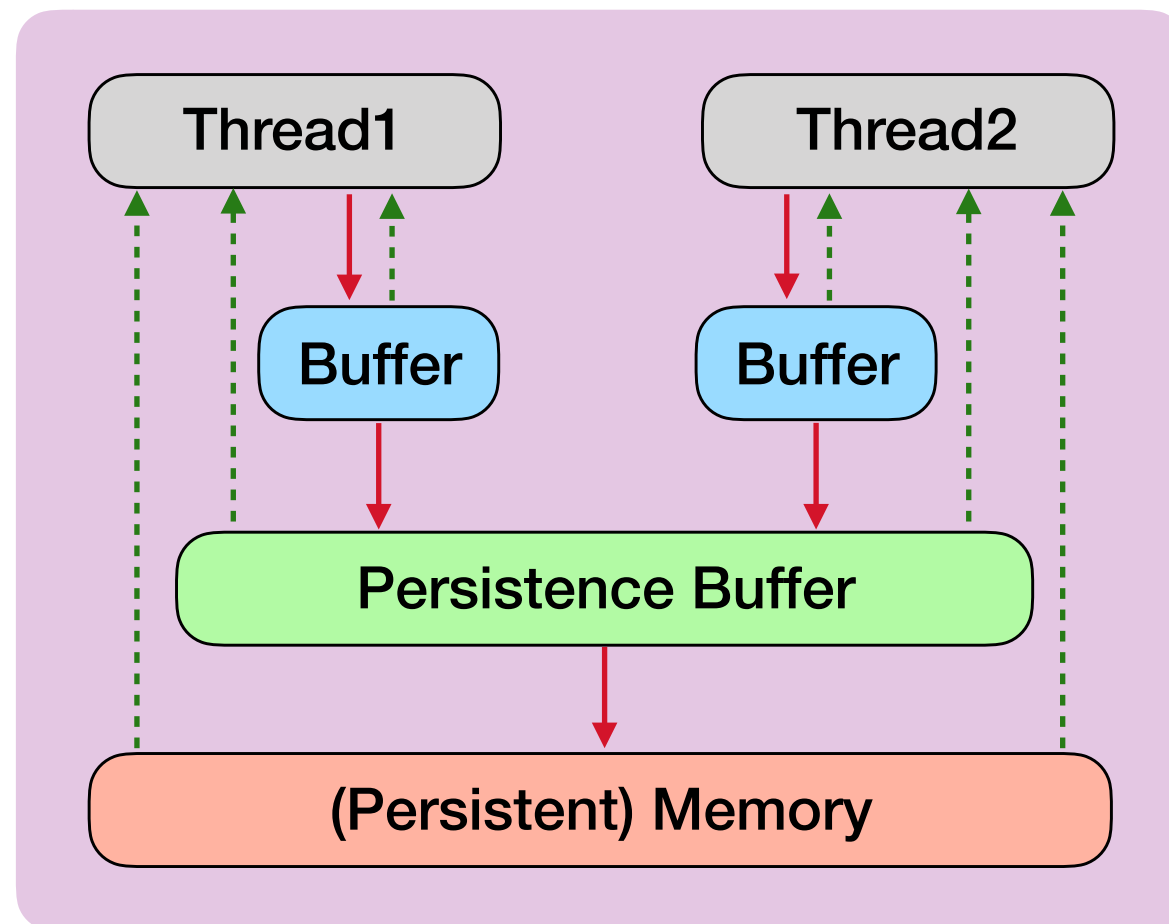
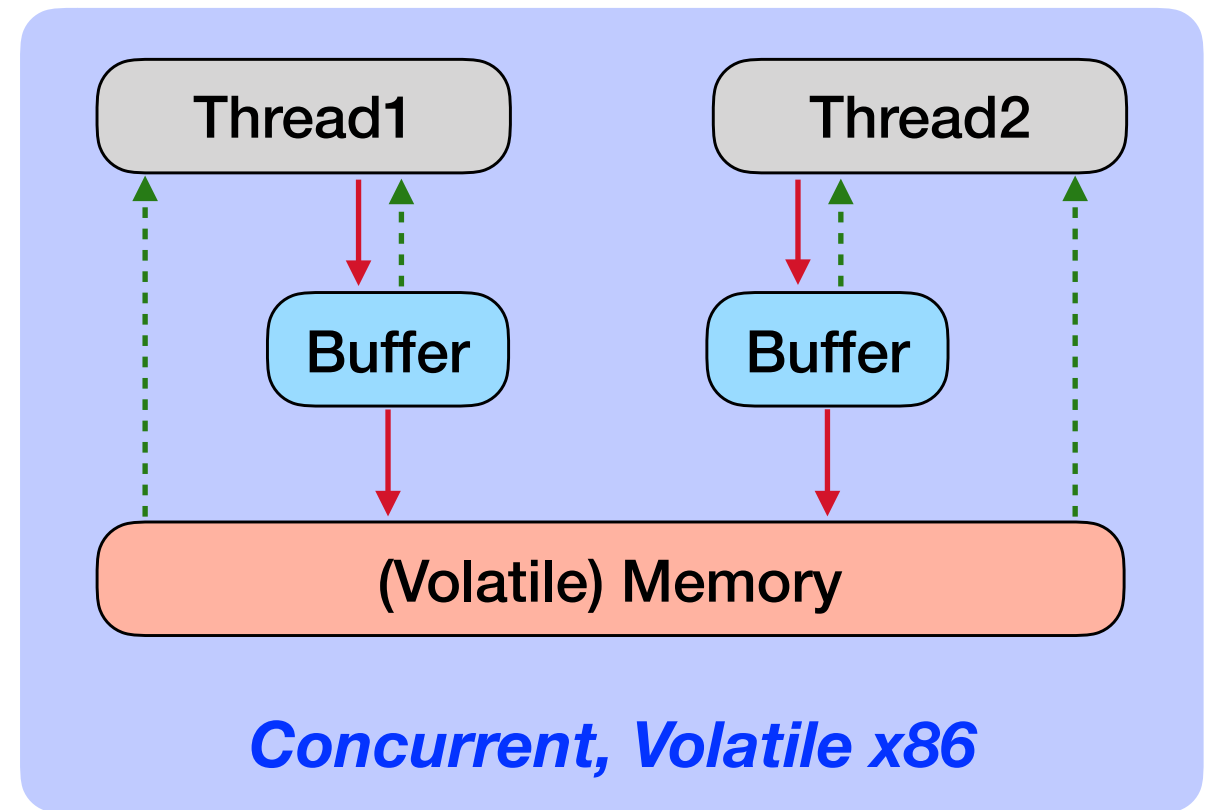
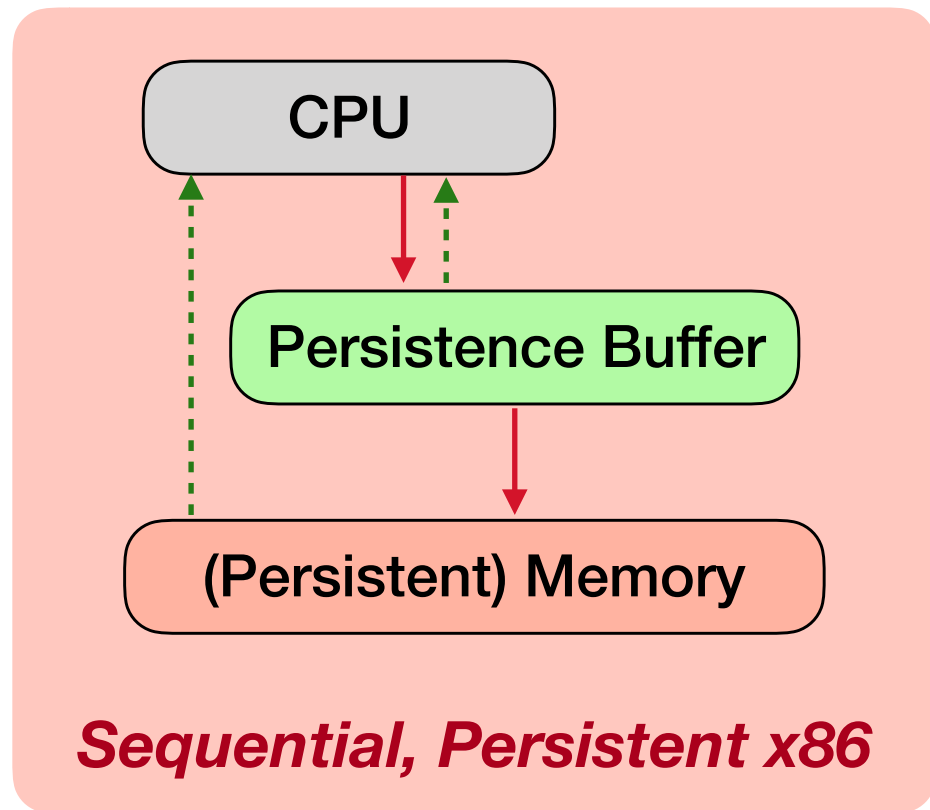
$a := x$: if **buffer** contains x , reads latest entry
else reads from **memory**



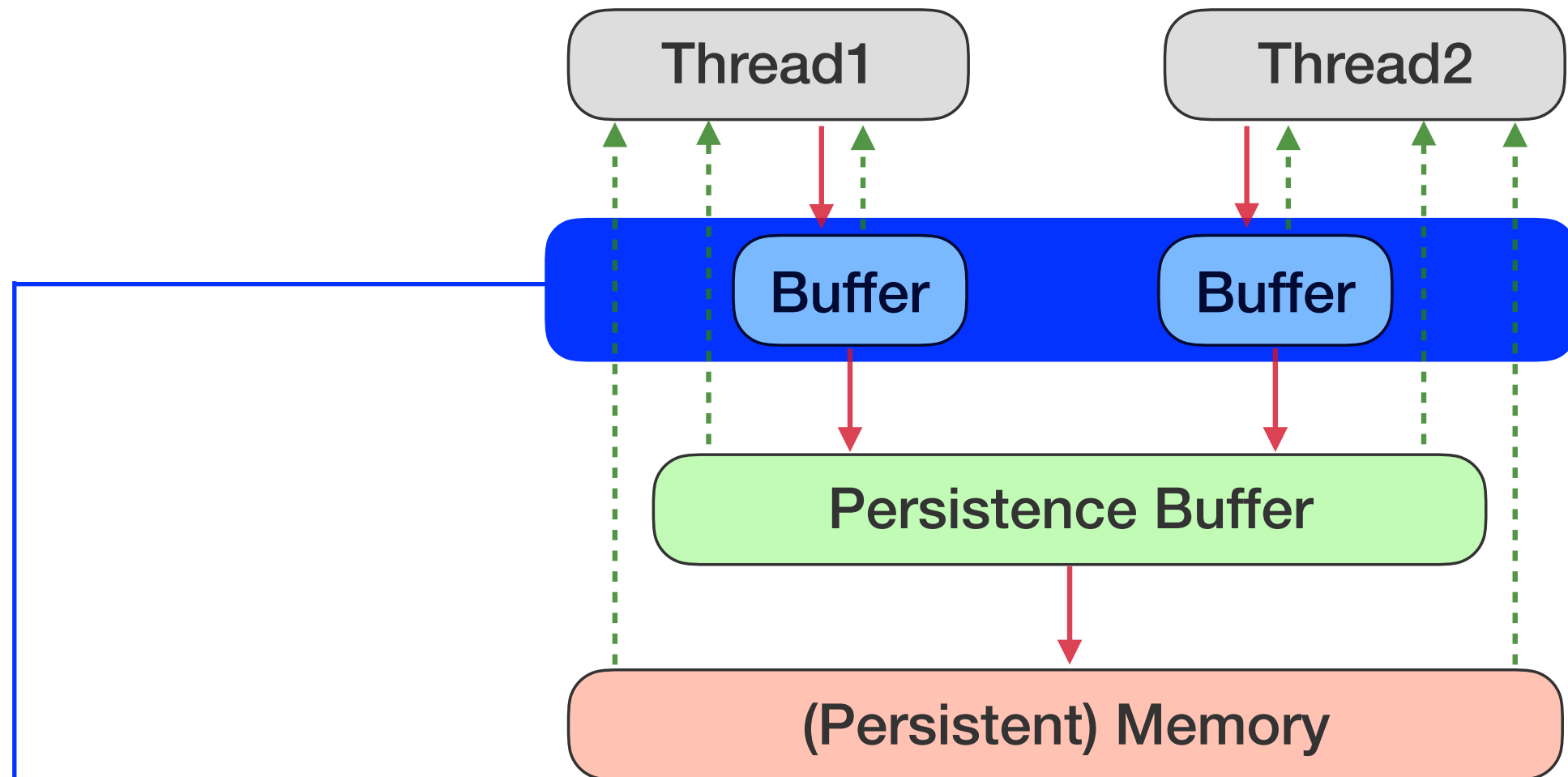
buffer and **memory** lost

* at non-deterministic times

Px86: Persistent & Concurrent x86

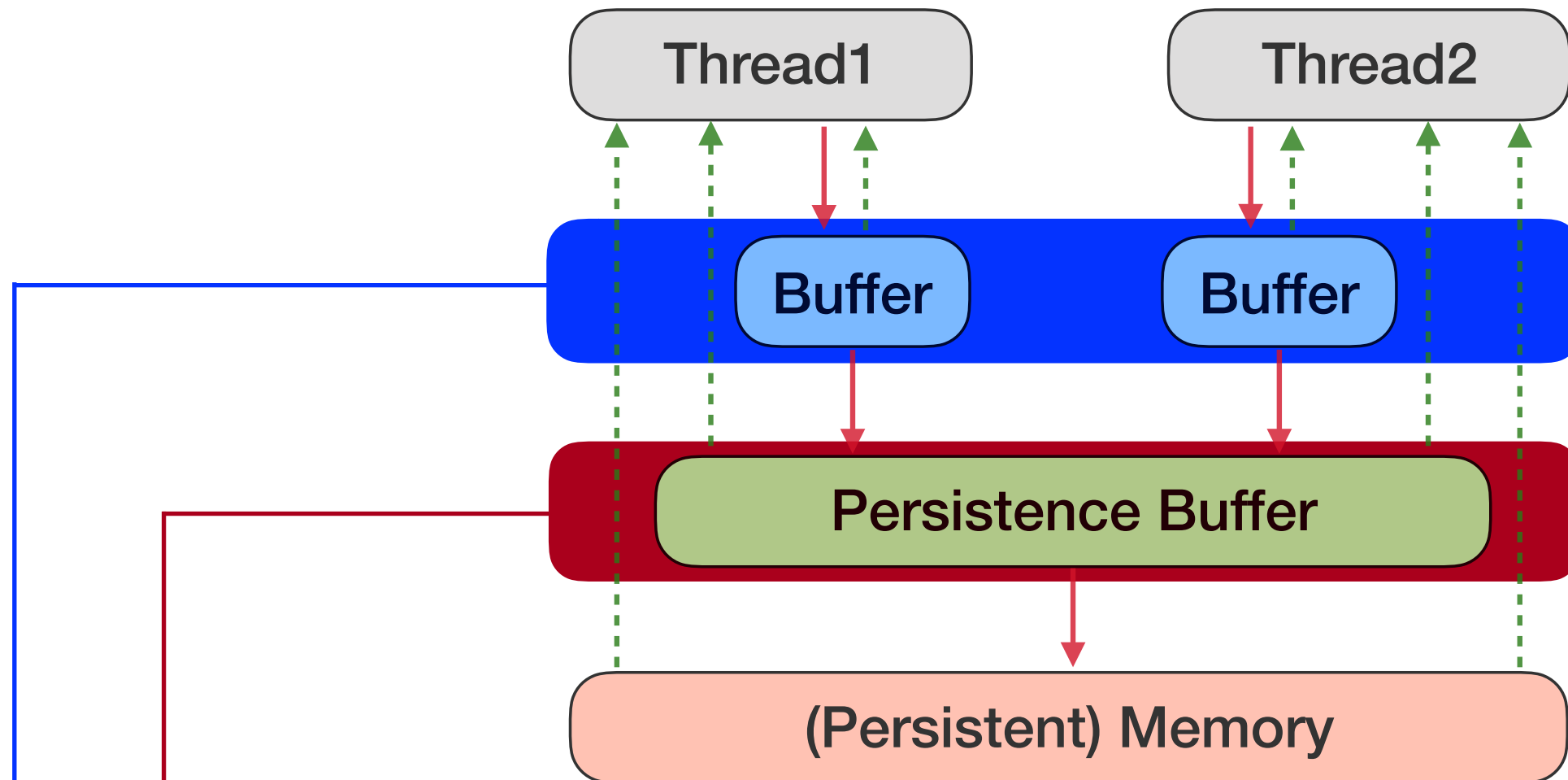


Persistent x86 (Px86)



buffer/unbuffer order: **consistency** model

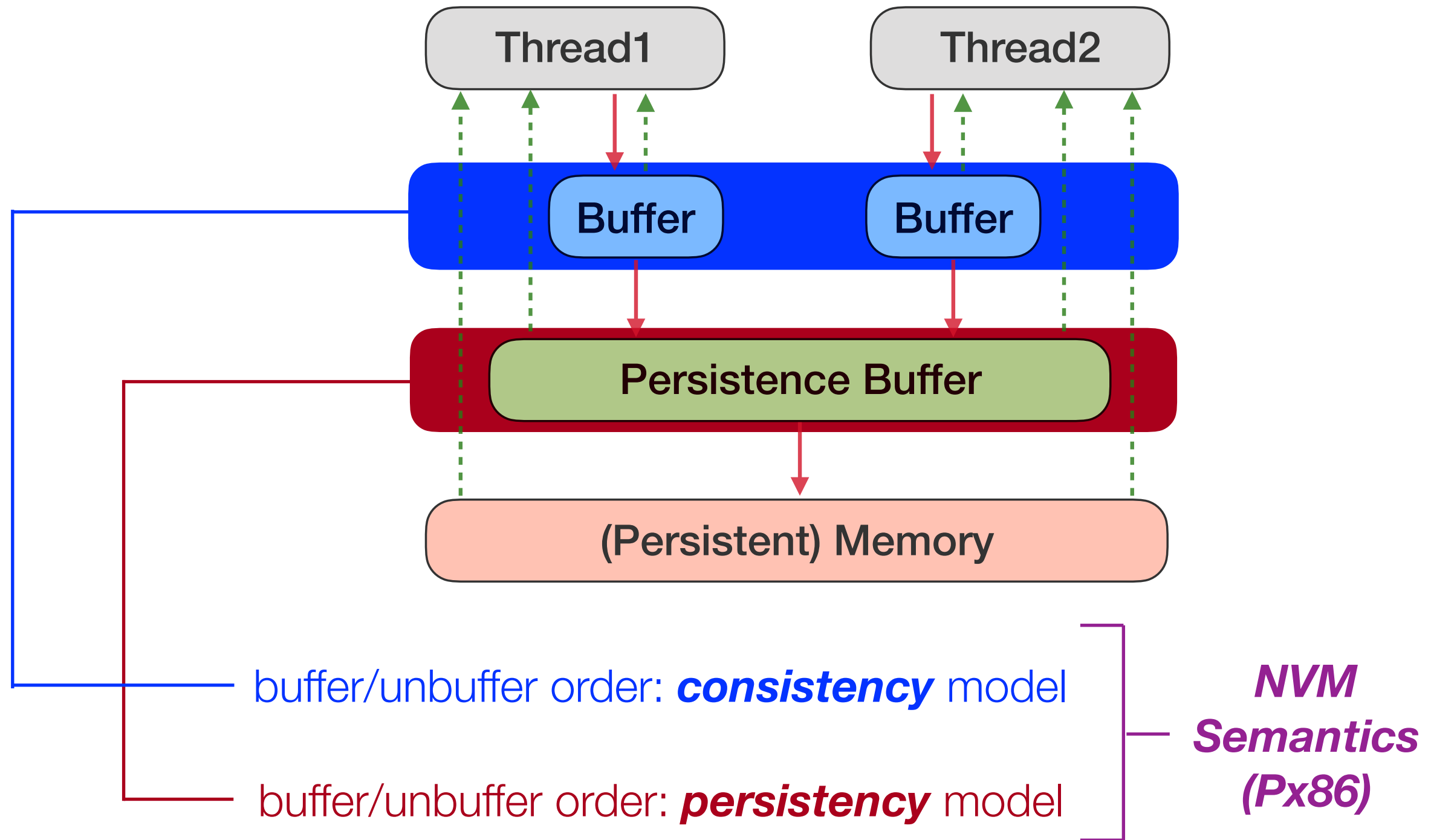
Persistent x86 (Px86)



buffer/unbuffer order: **consistency** model

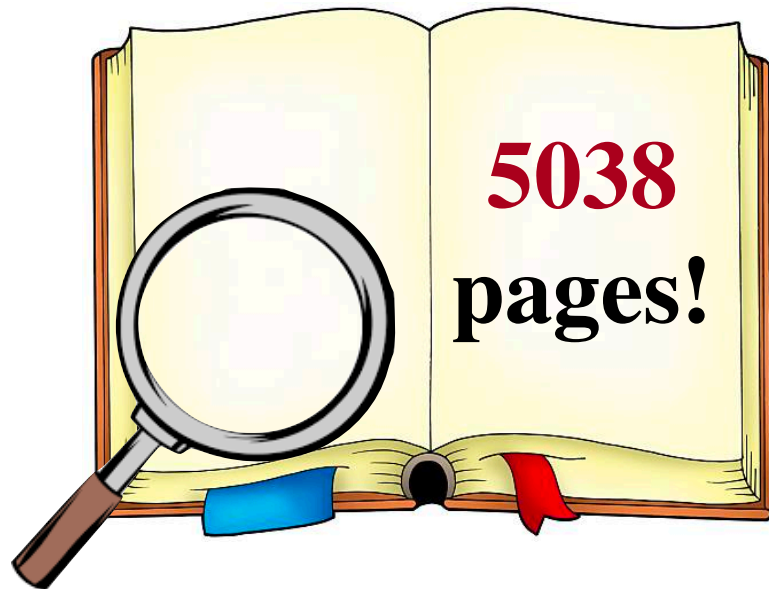
buffer/unbuffer order: **persistence** model

Persistent x86 (Px86)



Px86

Intel® Architecture Reference Manual



“Executions of the **clwb** instruction are ordered with respect to fence instructions ...”

“ They are not ordered with respect to other executions of **clwb**, to executions of **clflush** and **clflushopt** ...”

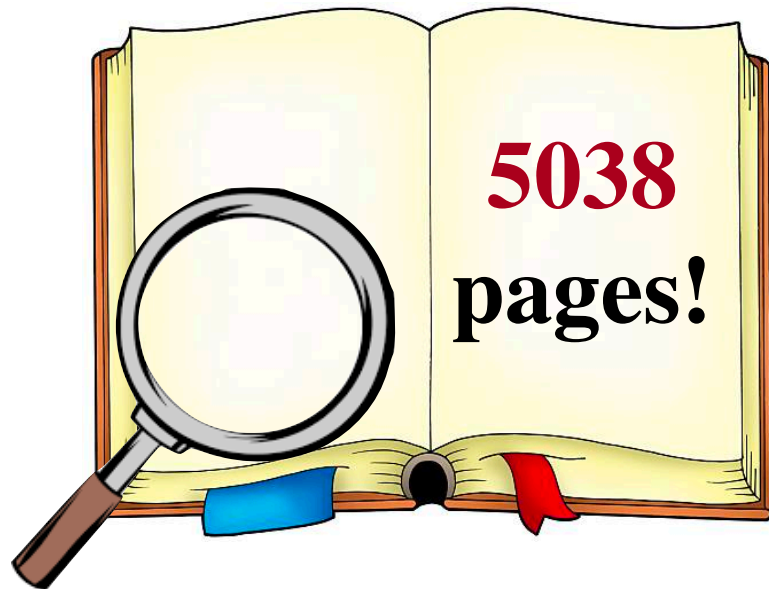
Ambiguities in text!



Two Px86 models

Px86

Intel® Architecture Reference Manual



“Executions of the **clwb** instruction are ordered with respect to fence instructions ...”

“ They are not ordered with respect to other executions of **clwb**, to executions of **clflush** and **clflushopt** ...”

Ambiguities in text!



Two Px86 models

Px86_{man}

- faithful to **manual** text
- **weaker** than **architectural intent**
- 2 models: **operational** & **declarative**
proved **equivalent**

Px86_{sim}

- captures **architectural intent**
- **stronger** than **manual** text
- 2 models: **operational** & **declarative**
proved **equivalent**

Summary

- ✓ Formalised Intel-x86 NVM semantics:
 - ✦ **Px86_{man}**: **equivalent** operational & declarative models
 - ✦ **Px86_{sim}**: **equivalent** operational & declarative models
- ✓ More in the paper
 - ✦ **Persistent transactional** library implemented in Px86
 - ✦ **Persistent queue** library implemented in Px86
- ? Future Work:
 - ✦ program logics
 - ✦ model checking algorithms
 - ✦ litmus testing

Summary

- ✓ Formalised Intel-x86 NVM semantics:
 - ✦ **Px86_{man}**: **equivalent** operational & declarative models
 - ✦ **Px86_{sim}**: **equivalent** operational & declarative models
- ✓ More in the paper
 - ✦ **Persistent transactional** library implemented in Px86
 - ✦ **Persistent queue** library implemented in Px86
- ? Future Work:
 - ✦ program logics
 - ✦ model checking algorithms
 - ✦ litmus testing

Thank You for Listening!