Persistency Semantics of the Intel-x86 Architecture

Azalea Raad\textsuperscript{1,2} John Wickerson\textsuperscript{2} Gil Neiger\textsuperscript{3} Viktor Vafeiadis\textsuperscript{1}

\textsuperscript{1} Max Planck Institute for Software Systems (MPI-SWS)
\textsuperscript{2} Imperial College London
\textsuperscript{3} Intel Corporation

azalea@mpi-sws.org SoundAndComplete.org @azalearaad
Computer Storage

✓ fast
✗ volatile

✓ 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 | Non-Volatile memory |
|-----------------|---------------------|
|                 |                     |
| // x = 0        | // x = 0            |
| x := 1          | x := 1              |
| // x = 1        | // x = 1            |
|                 |                     |
| // no recovery  |                     |
| // x = 0        |                     |
|                 | // recovery routine  |
|                 | // x = 1            |
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
```

A: Program *Verification*
Q: Why *Formal* NVM Semantics?

What about *Concurrency*?

```
// x = y = ... = 0
C_1 || C_2 || ... || C_n
// ???

// recovery routine
// ???
```
Formal Semantic Models

Difficulty

Sequential (1940s)  SC (1979)  WMC (1990s)

6
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 \((to)\) \(\Rightarrow\)

\textit{weak} behaviour absent under SC, caused by:

- instruction reordering by compiler
- write propagation across cache hierarchy

\textbf{Consistency Model}

the \textit{order} in which writes are made visible to other threads

e.g. x86 (TSO), ARMv8, C11, Java
Formal Semantic Models

Difficulty

Sequential (1940s)

SC (1979)

WMC (1990s)

WNVMC (2017)

This Talk
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
This Talk

Px86
(Persistent x86):
NVM Semantics of the x86 Architecture
Warmup: Sequential Px86
x86: (Sequential) Persistent Hardware Model

\[ x := 1 \text{ : adds } x := 1 \text{ to } p\text{-buffer} \]
x86: (Sequential) Persistent Hardware Model

Unbuffered at \textit{non-deterministic} points in time!

\( x := 1 \) : adds \( x := 1 \) to p-buffer

\textit{unbuffer}*: p-buffer to memory

* at non-deterministic times
x86: (Sequential) Persistent Hardware Model

Unbuffered at *non-deterministic* points in time!
Buffering & unbuffering orders may disagree!

\[ x := 1 : \text{adds } x := 1 \text{ to } p\text{-buffer} \]

\[ \text{unbuffer}^* : p\text{-buffer to memory} \]

* 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

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;
// recovery routine
y := 1;

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

!! out of order persists
_iff persist barriers?_
Persist Barriers: Desiderata

// x=0; y=0

// x=1; y=0

!! out of order persists

x86 does not provide persist barriers!

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

```plaintext
// 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

.FileWriter.Data = nil

// Explicit persists?
```
Explicit Persists: *Reality on x86*

// x=0; y=0

x := 1;

\[\text{clwb } x;\]

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*

\[\text{explicit persists?}\]

\[\text{clwb } x/\text{clflushopt } x/\text{clflush } x:\]

*asynchronously* persist cache line containing \(x\)
Explicit Persists: *Reality on x86*

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

x86 explicit persists are **asynchronous** and can themselves **persist out of order**!
Solution: *Persist Sequence*

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

- `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)

Thread1  ...  Thread2
Buffer  Buffer
(Volatile) Memory

\[ x := 1 \] : adds \( x := 1 \) to buffer

unbuffer* : buffer to memory

\[ a := x \] : if buffer contains \( x \), reads latest entry else reads from memory

\[ \text{buffer and memory lost} \]

* at non-deterministic times
Px86: Persistent & Concurrent x86

Sequential, Persistent x86

Concurrent, Volatile x86
Persistent x86 (Px86)

buffer/unbuffer order: **consistency** model
Persistent x86 (Px86)

- Buffer
- Persistence Buffer
- (Persistent) Memory

buffer/unbuffer order: **consistency** model

buffer/unbuffer order: **persistency** model
Persistent x86 (Px86)

Buffer/Unbuffer order: **consistency** model

Buffer/Unbuffer order: **persistence** model

NVM Semantics (Px86)
“Executions of the \texttt{clwb} instruction are ordered with respect to fence instructions …”

“They are not ordered with respect to other executions of \texttt{clwb}, to executions of \texttt{clflush} and \texttt{clflushopt} …”

\textbf{Ambiguities} in text!

\textbf{Two} Px86 models
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

Px86$_{\text{man}}$

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

Px86$_{\text{sim}}$

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

✓ Formalised Intel-x86 NVM semantics:
  ✦ \textit{Px86}_{\text{man}}: \textbf{equivalent} operational & declarative models
  ✦ \textit{Px86}_{\text{sim}}: \textbf{equivalent} operational & declarative models

✓ More in the paper
  ✦ \textit{Persistent transactional} library implemented in Px86
  ✦ \textit{Persistent queue} library implemented in Px86

❓ Future Work:
  ✦ program logics
  ✦ model checking algorithms
  ✦ litmus testing
Summary

✓ Formalised Intel-x86 NVM semantics:
  ✷ $P_{x86_{man}}$: equivalent operational & declarative models
  ✷ $P_{x86_{sim}}$: 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!