Persistence Semantics for Weak Memory
Integrating Epoch Persistency with the TSO Memory Model

Azalea Raad       Viktor Vafeiadis
Max Planck Institute for Software Systems (MPI-SWS)

Thursday 8 November
OOPSLA 2018
Boston, USA

azalea@mpi-sws.org  SoundAndComplete.org  @azalearaad
History

Difficulty

Sequential  SC  WMC

😊  ☹  ☹
Persistence Semantics for Weak Memory
Integrating Epoch-Persistence with the TSO Memory Model

AZALEA RADO, MPI/HD, Germany
VIKTOR VAFIANIS, MPI/HD, Germany

Emerging non-volatile memory (NVM) technologies promise the durability of disks with the performance of volatile memory (RAM). To describe the persistence guarantees of NVMs, several memory persistence models have been proposed in the literature. However, the formal semantics of such persistence models is the context of existing mainstream hardware has been simplified to data. To close this gap, we integrate the isolated epoch-persistence model with the “total-order” (TSO) memory model of the SPARC architecture. We then develop the PDS “perspective” (PDS) model and formalize its semantics both operationally and deductively. We demonstrate that the two characterizations of PDS are equivalent. We then formulate the notion of persistence linearity to establish the correctness of library implementations in the context of persistent memory. To showcase our formalism, we develop two persistent implementations of a space library, and apply persistent linearity to show their correctness.
What is Persistent Memory?

**Volatile** memory

// x = 0
x := 1
// x = 1

// x = v : reading x yields v
What is Persistent Memory?

Volatile memory

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

// x = v : reading x yields v
What is Persistent Memory?

**Volatile** memory

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

// no recovery
// x = 0
```

**Persistent** memory

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

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

// x = v : reading x yields v
What is Persistent Memory?

Volatile memory

\[
\begin{align*}
    &// \ x = 0 \\
    x &:= 1 \\
    &// \ x = 1 \\
    &// \text{no recovery} \\
    &// \ x = 0
\end{align*}
\]

Persistent memory

\[
\begin{align*}
    &// \ x = 0 \\
    x &:= 1 \\
    &// \ x = 1 \\
    &\text{recovery routine} \\
    &// \ x = 0 \ \text{OR} \ x = 1
\end{align*}
\]

persists are \textit{asynchronous} (buffered): may not persist immediately

\[
// \ x = v \ : \ \text{reading } x \ \text{yields } v
\]
(Sequential) Hardware
(Sequential) Hardware

CPU

(Volatile) Memory
(Sequential) Hardware

\[ x := 1 \quad \text{adds} \quad x := 1 \quad \text{to memory} \]
(Sequential) Hardware

\[
x := 1 : \text{adds } x := 1 \text{ to memory}
\]

\[
a := x : \text{reads } x \text{ from memory}
\]
(Sequential) Hardware

CPU

(Volatile) Memory

\[
x := 1 : \text{ adds } x := 1 \text{ to memory}
\]

\[
a := x : \text{ reads } x \text{ from memory}
\]

memory lost
**Sequential** Hardware

- $x := 1$ : adds $x := 1$ to memory
- $a := x$ : reads $x$ from memory

Memory lost
(Sequential) Hardware

- \( x := 1 \) : adds \( x := 1 \) to memory
- \( a := x \) : reads \( x \) from memory
- \( x := 1 \) : adds \( x := 1 \) to p-buffer
- Memory lost
(Sequential) Hardware

**CPU**

- `x:=1` : adds `x:=1` to memory
- `a:=x` : reads `x` from memory
  - `memory` lost

**Volatile) Memory**

**CPU**

- `x:=1` : adds `x:=1` to p-buffer
- `a:=x` : if p-buffer contains `x`, reads latest entry
  - else reads from memory

**Persistence Buffer**

**Persistent) Memory**
(Sequential) Hardware

CPU

(Volatile) Memory

- $x := 1$ : adds $x := 1$ to memory
- $a := x$ : reads $x$ from memory
- ⚡ memory lost

Persistence Buffer

CPU

(Persistent) Memory

- $x := 1$ : adds $x := 1$ to p-buffer
- $a := x$ : if p-buffer contains $x$, reads latest entry
  else reads from memory
- ⚡ p-buffer lost; memory *retained*
(Sequential) Hardware

CPU

(Volatile) Memory

\[ x := 1 : \text{ adds } x := 1 \text{ to memory} \]

\[ a := x : \text{ reads } x \text{ from memory} \]

\[ \text{memory lost} \]

Persistence Buffer

CPU

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

\[ a := x : \text{ if p-buffer contains } x, \text{ reads latest entry} \]
\[ \text{else reads from memory} \]

\[ p\text{-buffer lost; memory retained} \]

(Persistent) Memory

unbuffer* : p-buffer to memory

* at non-deterministic times
What is Memory Persistency Model?

- Memory **consistency** model describes: the order writes are made visible to other threads e.g. SC, TSO, ...
What is Memory Persistency Model?

- Memory **consistency** model describes:
  the order writes are made visible to other threads
  e.g. SC, TSO, ...

- Memory **persistency** model describes:
  the order writes are persisted to memory
  e.g. Epoch Persistency
What is Memory Persistency Model?

- Memory **consistency** model describes:
  
  Problem
  
  **Formal**
  
  Epoch Persistency Model
  
  for
  
  **Mainstream Hardware** (Weak Memory Models)

  the order writes are persisted to memory
  
  e.g. Epoch Persistency
What Can Go Wrong?

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

// recovery routine
// x=0; y=0 OR x=1; y=1 OR x=1; y=0 OR x=0; y=1
What Can Go Wrong?

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

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

‼️ Writes may persist out of order
What Can Go Wrong?

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

// recovery routine

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

!! Writes may persist out of order

persistent fence pfence
Persistent Fence

// x=0; y=0
x := 1;
// recovery routine
pfence;
y := 1;

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

\[ x := 1; \]
\[ y := 2; \]
\[ x := 3; \]
\[ pfence; \]
\[ z := 4; \]
Persistent Fence

- writes on \textbf{same locations} persist in \textbf{execution order}

\begin{align*}
\text{a} & : x := 1; \\
\text{b} & : y := 2; \\
\text{c} & : x := 3; \\
\text{pfence}; \\
\text{d} & : z := 4;
\end{align*}

\text{a} \quad \text{persists before} \quad \text{c}
Persistent Fence

- writes on **same locations** persist in execution order
- writes on **different locations** are unordered

```
x := 1;
y := 2;
x := 3;
pfence;
z := 4;
```

- a persists before c
- a, b may persist in any order
Persistent Fence

• writes on **same locations** persist in execution order
• writes on **different locations** are unordered
• pfence adds a new **epoch**

```
   a    x := 1;
   b    y := 2;
   c    x := 3;
       pfence;
   d    z := 4;
```

- a persists before c
- a, b may persist in any order
Persistent Fence

- writes on **same locations** persist in **execution order**
- writes on **different locations** are **unordered**
- `pfence` adds a new **epoch**
- writes persist in **epoch order**

```
x := 1;
y := 2;
x := 3;
pfence;
z := 4;
```
What Can Go Wrong (Continued)?

// $x=0; y=0$

$x := 1;$
$y := 1;$

// recovery routine

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

!! Execution continues ahead of persistence
What Can Go Wrong (Continued)?

Execution continues ahead of persistence

```plaintext
x := 1;
pfence;
y := 1;

// recovery routine

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

!! Execution continues ahead of persistence

persistent sync \( psync \)
What Can Go Wrong (Continued)?

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

// recovery routine

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

!! Execution continues ahead of persistence

persistent sync psync
C1; psync; C2

• same persist-ordering as pfence
• C2 executed only when all C1 writes have persisted
Persistent Sync

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

// recovery routine

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

!! Execution continues ahead of persistence

persistent sync psync

C1; psync; C2

• same persist-ordering as pfence
• C2 executed only when all C1 writes have persisted
x:=1 : adds x:=1 to p-buffer

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

p-buffer lost; memory retained
(Sequential) Hardware

CPU

epoch n
pfence

ePOCH 2
pfence
epoch 1

(Persistent) Memory

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

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

p-buffer lost; memory retained

unbuffer* : p-buffer to memory (in epoch order)

* at non-deterministic times
(Sequential) Hardware

- \( x := 1 \): adds \( x := 1 \) to p-buffer
- \( a := x \): if p-buffer contains \( x \), reads latest entry, else reads from memory
- \( \text{p-buffer lost; memory retained} \)
- \( \text{unbuffer}^* \): p-buffer to memory (in epoch order)
- \( \text{pfence} \): introduces a new epoch in p-buffer

* at non-deterministic times
(Sequential) Hardware

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

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

\text{p-buffer lost; memory retained}

unbuffer*: p-buffer to memory (in epoch order)

pfence : introduces a new epoch in p-buffer

psync : flushes the entire p-buffer to memory

* at non-deterministic times
What about Concurrency?

- TSO
- POWER
- ARMv8
- ...
What about Concurrency?

TSO
POWER
ARMv8
...
Contributions
Contributions

• PTSO: First formal epoch persistency semantics under mainstream hardware
  ▶ *Operational* model
  ▶ *Declarative* model
  ▶ *Equivalence* of the two models
Contributions

• PTSO: First formal epoch persistency semantics under mainstream hardware
  ▶ Operational model
  ▶ Declarative model
  ▶ Equivalence of the two models
Contributions

- PTSO: First formal epoch persistency semantics under mainstream hardware
  - Operational model
  - Declarative model
  - Equivalence of the two models

- Verifying programs under PTSO
  - PTSO programming pattern
  - Correctness condition: persistent linearisability
  - Verified several examples under PTSO
Total Store Ordering (TSO)
Total Store Ordering (TSO)
Total Store Ordering (TSO)

Thread1

x := 1;
a := y;

Thread2

y := 1;
c := x;

Store Buffering (SB)
Total Store Ordering (TSO)

Thread1

x = 1

Thread2

x = 1

y := 1;
c := x;

a := y;

x = 0; y = 0;

Thread1

y := 1;
c := x;

Thread2

Store Buffering (SB)
Total Store Ordering (TSO)

Thread1

\[ x = 1 \]

Thread2

\[ y = 1 \]

\[ x = 0; \ y = 0; \]

Store Buffering (SB)

Thread1

\[ x := 1; \]
\[ a := y; \]

Thread2

\[ y := 1; \]
\[ c := x; \]
Total Store Ordering (TSO)

Thread1

\[ x = 1 \]

Thread2

\[ y = 1 \]

\[ x = 0; \ y = 0; \]

Thread1

\[ x := 1; \]
\[ a := y; \]

Thread2

\[ y := 1; \]
\[ c := x; \]

Store Buffering (SB)
Total Store Ordering (TSO)

Thread1

\[ x = 1 \]

Thread2

\[ y = 1 \]

\[ x = 0; \ y = 0; \]

Thread1

\[ x := 1; \]
\[ a := y; \]

Thread2

\[ y := 1; \]
\[ c := x; \]

Store Buffering (SB)
Total Store Ordering (TSO)

Thread1

\[ x = 1 \]

Thread2

\[ y = 1 \]

\[ x = 0; y = 0; \]

Thread1

\[ x := 1; \]
\[ a := y; \quad // \ 0 \]

Thread2

\[ y := 1; \]
\[ c := x; \]

Store Buffering (SB)
Total Store Ordering (TSO)

Thread1

\[ x = 1 \]

Thread2

\[ y = 1 \]

\[ x = 0; \ y = 0; \]

Store Buffering (SB)

\[ x := 1; \]
\[ a := y; \ // \ 0 \]

\[ y := 1; \]
\[ c := x; \]
Total Store Ordering (TSO)

Thread1

\[ x = 1 \]

Thread2

\[ y = 1 \]

\[ x = 0; y = 0; \]

Thread1

\[ x := 1; \]
\[ a := y; \quad // \quad 0 \]

Thread2

\[ y := 1; \]
\[ c := x; \]

Store Buffering (SB)
Total Store Ordering (TSO)

Thread1

\[ x = 1 \]

Thread2

\[ y = 1 \]

\[ x = 0; \ y = 0; \]

Thread1

\[ x := 1; \]
\[ a := y; \quad // \quad 0 \]

Thread2

\[ y := 1; \]
\[ c := x; \quad // \quad 0 \]

Store Buffering (SB)
Total Store Ordering (TSO)

Thread1

\[ x := 1; \]
\[ a := y; \quad // \ 0 \]

Thread2

\[ y := 1; \]
\[ c := x; \quad // \ 0 \]

Store Buffering (SB)
Total Store Ordering (TSO)

Thread1

\[ x := 1; \]
\[ a := y; \quad // \; 0 \]

Thread2

\[ y := 1; \]
\[ c := x; \quad // \; 0 \]

Store Buffering (SB)
Persistent TSO (PTSO)
Persistent TSO (PTSO)

1. CPU → (Volatile) Memory
2. CPU → Persistence Buffer → (Persistent) Memory
3. Thread1 → Buffer → (Volatile) Memory
4. Thread2 → Buffer → (Volatile) Memory
Persistent TSO (PTSO)
Contributions

• PTSO: First formal epoch persistency semantics under mainstream hardware
  ▶ Operational model
  ▶ Declarative model
  ▶ Equivalence of the two models
Contributions

• PTSO: First formal epoch persistency semantics under mainstream hardware
  ▶ Operational model
  ▶ Declarative model
  ▶ Equivalence of the two models

• Verifying programs under PTSO
  ▶ PTSO programming pattern
  ▶ Correctness condition: persistent linearisability
  ▶ Verified several examples under PTSO
Contributions

• PTSO: First formal epoch persistency semantics under mainstream hardware
  ▶ Operational model
  ▶ Declarative model
  ▶ Equivalence of the two models

• Verifying programs under PTSO
  ▶ PTSO programming pattern
  ▶ Correctness condition: persistent linearisability
  ▶ Verified several examples under PTSO
Verifying programs under PTSO

The **persistent** variant of the Michael-Scott queue and its **recovery** mechanism
Verifying programs under PTSO

The persistent variant of the Michael-Scott queue and its recovery mechanism

What constitutes a **correct persistent** implementation?

The **persistent** variant of the Michael-Scott queue and its **recovery** mechanism
Linearisability

thread 1

thread 2

c deq(1)

a enq(1)

b enq(2)
time
Linearisability

- Define happens-before relation $hb$
  
  $\text{Thread 1}$  
  - $a$: enq(1)  
  - $b$: enq(2)  

  $\text{Thread 2}$  
  - $c$: deq(1)  

  $\text{Time}$

$\text{Thread 1}$  
  - $a$: enq(1)  
  - $b$: enq(2)  

  $\text{Thread 2}$  
  - $c$: deq(1)  

- $(e_1, e_2) \in hb \iff e_1.\text{end} < \text{time} e_2.\text{begin}$
Linearisability

- Define happens-before relation $hb$
  \[ (e_1, e_2) \in hb \iff e_1.end \lt \text{time} e_2.begin \]
  - e.g. \((a, b) \in hb\) \(\not\in hb\)
• Define happens-before relation $hb$
  
  $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin}$
  
  -- e.g. $(a, b) \in hb$  $(a, c) \notin hb$

• **Linearisable** $\iff \exists H. H$ totally orders events
  
  $H$ respects $hb$
  
  $H$ is a **legal** sequence (library-specific)
Linearisability

- Define happens-before relation $hb$
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin}$
    - e.g. $(a, b) \in hb$  $(a, c) \notin hb$

- **Linearisable** $\iff \exists H. H$ totally orders events
  - $H$ respects $hb$
  - $H$ is a **legal** sequence (library-specific)
    - e.g. FIFO sequences for queue

- Thread 1:
  - $a$ \text{enq}(1)
  - $b$ \text{enq}(2)

- Thread 2:
  - $c$ \text{deq}(1)
• Define happens-before relation $hb$
  - $(e_1, e_2) \in hb \iff e_1.end <_{\text{time}} e_2.begin$
    - e.g. $(a, b) \in hb$  $(a, c) \notin hb$

• **Linearisable** $\iff \exists H. H$ totally orders events
  - $H$ respects $hb$
  - $H$ is a **legal** sequence (library-specific)
    - e.g. FIFO sequences for queue
• Define happens-before relation $hb$
  - $(e_1, e_2) \in hb \iff e_1.end < time e_2.begin$
    - e.g. $(a, b) \in hb$ $(a, c) \notin hb$

• Linearisable $\iff \exists H. H$ totally orders events
  - $H$ respects $hb$
  - $H$ is a legal sequence (library-specific)
    - e.g. FIFO sequences for queue
Linearisability

- Define happens-before relation $hb$
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin}$
    - e.g. $(a, b) \in hb$ $(a, c) \notin hb$

- **Linearisable** $\iff \exists H. H$ totally orders events
  - $H$ respects $hb$
  - $H$ is a **legal** sequence (library-specific)
    - e.g. FIFO sequences for queue

\[ a \quad b \quad c \quad a \quad c \quad b \quad c \quad a \quad b \]
- Define happens-before relation $hb$
  \[ (e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin} \]
  -- e.g. $(a, b) \in hb$ \hspace{1cm} $(a, c) \notin hb$

- **Linearisable** $\iff \exists H. H$ totally orders events
  \[ a \hspace{0.5cm} b \hspace{0.5cm} c \hspace{0.5cm} a \hspace{0.5cm} c \hspace{0.5cm} b \hspace{0.5cm} c \hspace{0.5cm} a \hspace{0.5cm} b \]
  non-linearisable (not legal)
• Define happens-before relation $hb$

\[(e_1, e_2) \in hb \iff e_1.end <_{time} e_2.begin\]

-- e.g. $(a, b) \in hb$  \quad $(a, c) \not\in hb$
• Define happens-before relation $hb$
  
  \[ (e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin} \]

  -- e.g. $(a, b) \in hb$ \quad $(a, c) \notin hb$

• **Persistently linearisable** $\iff \exists H. H$ totally orders a *subset* $S$ of events
  
  \[ H \text{ respects } hb \]
  
  \[ H \text{ is a legal sequence} \]
• Define happens-before relation $hb$
  - $(e_1, e_2) \in hb \iff e_1.end <_{time} e_2.begin$
    -- e.g. $(a, b) \in hb$  $(a, c) \notin hb$

• **Persistently linearisable** $\iff \exists H. H$ totally orders a subset $S$ of events
  - $H$ respects $hb$
  - $H$ is a legal sequence
  - $S$ is $hb$-prefix-closed: $(a, b) \in hb$  and  $b \in S \implies a \in S$
    -- persists are asynchronous: only a prefix may persist after a crash
• Define happens-before relation $hb$
  
  $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin}$

  -- e.g. $(a, b) \in hb$  $(a, c) \not\in hb$

• **Persistently linearisable** $\iff \exists H. H$ totally orders a **subset** $S$ of events
  
  $H$ respects $hb$

  $H$ is a legal sequence

  $S$ is **$hb$-prefix-closed**: $(a, b) \in hb$ and $b \in S \implies a \in S$

  -- persists are **asynchronous**: only a **prefix** may persist **after a crash**

Persistently linearisable
What about Multiple Crashes?
What about Multiple Crashes?
What about Multiple Crashes?

The diagram illustrates the execution and recovery phases over time. There are multiple execution and recovery phases marked with different labels: $G_1$, $G_2$, and $G_n$. The diagram also shows a sequence of crashes indicated by lightning symbols, followed by recovery. The blue line at the end indicates that there are no crashes.
What about Multiple Crashes?

- A chain $G_1 \ldots G_n$ is **persistently linearisable** $\iff \exists H_1 \ldots H_n$.
  - $H_i$ persistently linearises $G_i$ — as before
  - $H_1 \leftrightarrow \ldots \leftrightarrow H_n$ is a legal sequence
Conclusions

• PTSO: First formal epoch persistency semantics under mainstream hardware
  ▶ Operational model
  ▶ Declarative model
  ▶ Equivalence of the two models

• Verifying programs under PTSO
  ▶ PTSO programming pattern
  ▶ Correctness condition: persistent linearisability
  ▶ Verified several examples under PTSO
Conclusions

• PTSO: First formal epoch persistency semantics under mainstream hardware
  ▶ Operational model
  ▶ Declarative model
  ▶ Equivalence of the two models

• Verifying programs under PTSO
  ▶ PTSO programming pattern
  ▶ Correctness condition: persistent linearisability
  ▶ Verified several examples under PTSO

Thank you for listening!

azalea@mpi-sws.org  SoundAndComplete.org  @azalearaad
Programming Pattern

1. // log progress
2. pfence
3. // do the work
4. pfence
Programming Pattern

1. // log progress
2. pfence
3. // do the work
4. pfence

Log *at most one step* ahead of work
Programming Pattern

Log at most one step ahead of work

q.enq(v) ≜

1. pc := getPC(); t := getTC();
   n := newNode(v,t,pC,null);
   map[t][pc] := n;
2. pfence;
3. h := q.head;
   find: while (q.data[h] != null)
      h := h+1;
      if (!CAS(q.data[h],null,n))
         goto find;
4. pfence;