Strand Persistency

Vaibhav Gogte, William Wang§, Stephan Diestelhorst§, Peter M. Chen, Satish Narayanasamy, Thomas F. Wenisch

NVMW
03/12/2019
Promise of persistent memory (PM)

Performance +

Density +

Non-volatility
Promise of persistent memory (PM)

Performance + Density + Non-volatility

Intel Announces New Optane DC Persistent Memory *

“Optane DC Persistent Memory will be offered in packages of up to 512GB per stick.”

“... expanding memory per CPU socket to as much as 3TB.”

* Source: www.extremetech.com
Promise of persistent memory (PM)

Performance +
Density +
Non-volatility

"Optane DC Persistent Memory will be offered in packages of up to 512GB per stick."
"... expanding memory per CPU socket to as much as 3TB."

* Source: www.extremetech.com

Byte-addressable, load-store interface to durable storage
Persistent memory system

- CPU
- Writeback caches
- DRAM
- Persistent Memory (PM)
Persistent memory system

Failure

CPU

Writeback caches

DRAM

Persistent Memory (PM)
Persistent memory system

Failure

CPU

Writeback caches

DRAM

Persistent Memory (PM)

Recovery

Recovery can inspect PM data-structures to restore system to a consistent state
Recovery requires PM access ordering

\[ \text{St } a = x \]

\[ \text{for recovery} \]

\[ \text{St } b = y \]
Recovery requires PM access ordering

Consistency model

Intel x86 primitives

St a = x

St b = y
Recovery requires PM access ordering

St a = x
for recovery

St b = y

Consistency model

Writeback caches

Persistency model

CPU

PM

Intel x86 primitives

St a = x

CLWB(a)

St b = y

CLWB(b)
Recovery requires PM access ordering

St a = x
St b = y

for recovery

Intel x86 primitives

St a = x
CLWB(a)
SFENCE
St b = y
CLWB(b)

CPU
Writeback caches
PM

Consistency model
Persistency model
Recovery requires PM access ordering

Hardware systems provide primitives to express *persist* order to PM

**Consistency model**
- Intel x86 primitives
  - St a = x
  - CLWB(a)
  - SFENCE
  - St b = y
  - CLWB(b)

**Persistence model**
- St a = x
- St b = y

for recovery
Hardware imposes overly strict constraints

St A = 1; CLWB (A)
St B = 2; CLWB (B)
St C = 3; CLWB (C)
Hardware imposes overly strict constraints

St A = 1; CLWB (A)
St B = 2; CLWB (B)
St C = 3; CLWB (C)

Ideal DAG

A

B

C

DAG 1

St A = 1; CLWB (A)

SFENCE

St B = 2; CLWB (B)
St C = 3; CLWB (C)

A

B

C
Hardware imposes overly strict constraints

St A = 1; CLWB (A)
St B = 2; CLWB (B)
St C = 3; CLWB (C)

Ideal DAG

A
\rightarrow
B
C

St A = 1; CLWB (A)
St B = 2; CLWB (B)
St C = 3; CLWB (C)

SFENCE

DAG 1

A
\rightarrow
B
C

St A = 1; CLWB (A)
St C = 3; CLWB (C)

SFENCE

DAG 2

A
\rightarrow
B
C
Hardware imposes overly strict constraints

\[ \text{St A} = 1; \ \text{CLWB} \ (A) \]
\[ \text{St B} = 2; \ \text{CLWB} \ (B) \]
\[ \text{St C} = 3; \ \text{CLWB} \ (C) \]

Primitives in existing hardware systems overconstrain PM accesses

Ideal DAG

DAG 1

DAG 2
Contributions

• Employ *strand persistency* [Pelley14]
  – Hardware ISA primitives to specify precise ordering constraints

• Comprises two primitives: **PersistBarrier** and **NewStrand**
  – Can encode an arbitrary DAG

• Map language-level persistency models to ISA level primitives
  – Leverage strand persistency to build persistency models efficiently
Contributions

• Employ *strand persistency* [Pelley14]
  – Hardware ISA primitives to specify precise ordering constraints

• Comprises two primitives: **PersistBarrier** and **NewStrand**
  – Can encode an arbitrary DAG

• Map language-level persistency models to ISA level primitives
  – Leverage strand persistency to build persistency models efficiently

**Strand persistency improves perf. of language persistency models by 21.4% (avg.)**
Outline

• Contributions
• Example: Failure atomicity
• Existing hardware primitives
• Strand persistency
• Evaluation
Example: Failure atomicity

**Failure-atomicity:**
Which group of stores persist atomically?

```c
atomic_begin()

Failure-atomic region

x = 100;
y = 200;

atomic_end()
```
Example: Failure atomicity

**Failure-atomicity:**
Which group of stores persist atomically?

```
atomic_begin()

Failure-atomic region

x = 100;
y = 200;

atomic_end()
```

Failure-atomicity limits state that recovery can observe after failure
Undo-logging for failure atomicity

\[\text{Init: } x = 0; \ y = 0\]

\[
\begin{align*}
\text{atomic\_begin()} \\
x &= 1; \\
y &= 2; \\
\text{atomic\_end()}
\end{align*}
\]

\[
\begin{align*}
\text{persistUndoLog (L)} \\
\text{mutateData (M)} \\
\text{persistData (P)} \\
\text{commitLog (C)}
\end{align*}
\]
Undo-logging for failure atomicity

Init: \( x = 0; \ y = 0 \)

atomic_begin()

- \( x = 1; \)
- \( y = 2; \)

atomic_end()

Persist undo log (L)

Mutate data (M)

Persist data (P)

Commit log (C)

Failure-atomic

Undo logging steps ordered to ensure failure-atomicity
Undo-logging for failure atomicity

*Init:* $x = 0; y = 0$

atomic_begin()

$x = 1;$
$y = 2;$

atomic_end()

- **Persist undo log** ($L$)
- **Mutate data** ($M$)
- **Persist data** ($P$)
- **Commit log** ($C$)

Failure-atomic

Undo logging steps ordered to ensure failure-atomicity
Hardware imposes stricter constraints

Ideal ordering

```c
atomic_begin()
    x = 1;
    y = 2;
atomic_end()
```
Hardware imposes stricter constraints

**Ideal ordering**

```c
atomic_begin()
    x = 1;
    y = 2;
atomic_end()
```

**SFENCE ordering**

```
Log(L_x,x)  CLWB(L_x)
    SFENCE
Log(L_y,y)  CLWB(L_y)
    SFENCE
Store(x,1)  Store(y,2)
```

```
Log(L_x,x)  CLWB(L_x)
    SFENCE
Log(L_y,y)  CLWB(L_y)
    SFENCE
Store(x,1)  Store(y,2)
```
Hardware imposes stricter constraints

atomic_begin()
  x = 1;
  y = 2;
atomic_end()
Hardware imposes stricter constraints

atomic_begin()
  x = 1;
  y = 2;
atomic_end()
Strand persistency enables persist concurrency

- Provides primitives to express precise persist order
Strand persistency enables persist concurrency

- Provides primitives to express precise persist order

Orders persists within a thread $\leftarrow$ PersistBarrier

Persist A
Persist B
Persist C
Strand persistency enables persist concurrency

- Provides primitives to express precise persist order

Orders persists within a thread \( \downarrow \) PersistBarrier
 Persist A
 Persist B
 Persist C

Initiates new stream of persists \( \downarrow \) NewStrand

Strand 0

Strand 1
**Strand persistency enables persist concurrency**

- Provides primitives to express precise persist order

**Orders** persists within a thread

**Initiates** new stream of persists

Persist A

Persist B

Persist C

PersistBarrier

NewStrand
Strand persistency enables persist concurrency

- Provides primitives to express precise persist order

Orders persists within a thread

Initiates new stream of persists

Persist A
Persist Barrier
Persist B
Persist C

Persists on different strands can be issued concurrently to PM
What if ordering is needed across strands?

- Conflicting accesses establish persist order across strands

Persist A

**PersistBarrier**

Persist B

Strand 0

Strand 1

A

B
What if ordering is needed across strands?

- Conflicting accesses establish persist order across strands

Persist A
Persist Barrier
Persist B
New Strand
Persist A
Persist Barrier
Persist C
What if ordering is needed across strands?

- Conflicting accesses establish persist order across strands

![Diagram showing persist order across strands]

- Persist A
- Persist B
- Persist C
- PersistBarrier
- NewStrand
- PersistBarrier
- Inter-strand order
Logging using strand persistency

```c
atomic_begin()
    x = 1;
    y = 2;
atomic_end()

PersistBarrier
Log(L_x, x)
CLWB(L_x)

Strand 0
Log(L_x, x)
CLWB(L_x)
Store(x, 1)
PersistBarrier
Store(x, 1)

Strand 1
Log(L_y, y)
CLWB(L_y)
Store(y, 2)
```
Logging using strand persistency

atomic_begin()
    x = 1;
    y = 2;
atomic_end()

PersistBarrier
Log(L_x,x)
CLWB(L_x)
Store(x,1)
PersistBarrier
Log(L_y,y)
CLWB(L_y)
Store(y,2)

Strand 0
Log(L_x,x)
CLWB(L_x)
Store(x,1)

Strand 1
Log(L_y,y)
CLWB(L_y)
Store(y,2)

Need to implement log buffer that can manage concurrent log updates
Log space under strand persistency

Persistent head atomically commits logs

Volatile tail for concurrent log creation
Log space under strand persistency

- Persistent head atomically commits logs
- Volatile tail for concurrent log creation

- Failure exposes log write reorderings
  - Identify valid logs in case of failure
  - Record order of log creation
  - Recovery rolls back partial updates using valid logs

More details in the paper
Language persistency models to ISA primitives

Hardware ISA

ISA primitives: PersistBarrier and NewStrand
Language persistency models to ISA primitives

- Compiler: Logging impl. that map to hardware primitives
- Hardware ISA: ISA primitives: PersistBarrier and NewStrand
Language persistency models to ISA primitives

- High-level languages: Failure atomicity for language-level persistency models
- Compiler: Logging impl. that map to hardware primitives
- Hardware ISA: ISA primitives: PersistBarrier and NewStrand
Evaluation: Language-level persistency models

ATLAS [Chakrabarti14]

- Failure-atomic outermost critical sections

```java
L1.lock();
x -= 100;
y += 100;
L2.lock();
a -= 100;
b += 100;
L2.unlock();
L1.unlock();
```
Evaluation: Language-level persistency models

ATLAS [Chakrabarti14]
- Failure-atomic outermost critical sections

Coupled-SFR [Gogte18]
- Failure-atomic synchronization-free regions

```java
L1.lock();
x -= 100;
y += 100;
L2.lock();
a -= 100;
b += 100;
L2.unlock();
L1.unlock();
```
Evaluation: Language-level persistency models

ATLAS [Chakrabarti14]
- Failure-atomic outermost critical sections

Coupled-SFR [Gogte18]
- Failure-atomic synchronization-free regions

Integrate our logging mechanisms with ATLAS and Coupled-SFR

```cpp
L1.lock();
    x -= 100;
    y += 100;
L2.lock();
    a -= 100;
    b += 100;
L2.unlock();
L1.unlock();
```
Methodology

• Gem5 simulator
• Workloads: **write intensive micro-benchmarks**
  – **Queue**: insert/delete entries in a queue
  – **Hashmap**: update values in persistent hash table
  – **Array swaps**: random swaps of array elements
  – **RBTree**: insert/delete entries in red-black tree
  – **TPCC**: new order transaction from TPCC
Performance evaluation

Improves performance of ATLAS by up to 29.9% (18.2% avg.)
Improves performance of Coupled-SFR by up to 34.5% (21.4% avg.)
Conclusion

- Strand persistency to precisely order persists
- Two primitives: **PersistBarrier** and **NewStrand**
  - Work together to relax ordering constraints in undo logging
- Evaluation using language-level persistency models
- Performance improvement of up to 34.5%
Strand Persistency

Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, Thomas F. Wenisch

NVMW
03/12/2019