Correct and Fast Persistency Guarantees

Sara Mahdizadeh Shahri
Aasheesh Kolli

YArch Workshop - February 2019
Why Persistent Memory?
Why Persistent Memory?

Non-volatility
Why Persistent Memory?

Non-volatility

Density
Why Persistent Memory?

- Byte-addressability
- Non-volatility
- Density
Why Persistent Memory?

- **Byte-addressability**
- **Non-volatility**
- **Density**
- **Performance**
Memory Hierarchy

- Core
- Cache
- DRAM
- Persistent Memory
Memory Hierarchy

Core

Cache

DRAM

Persistent Memory

Recovery
Correct Recovery

Use Case: Insertion in Queue

1

C

B

A

tail

head
Correct Recovery

Use Case: Insertion in Queue

1

C B A

2

D C B A

tail

head
Correct Recovery

Use Case: Insertion in Queue

1. C B A
2. D C B A
3. D C B A
Correct Recovery

Use Case: Insertion in Queue

1. C B A
2. C B A
3. D C B A
Correct Recovery

Use Case: Insertion in Queue

☑️ Order of Updates
Correct Recovery

Use Case: Insertion in Queue

1. Order of Updates
2. Failure Atomicity

![Diagram of queue with insertion and recovery process]
Failure Atomicity Semantics

- Providing Failure Atomicity for Granularity Beyond Persist Access
Failure Atomicity Semantics

- Providing Failure Atomicity for Granularity Beyond Persist Access

Logging Mechanism
(Undo Logging)
Failure Atomicity Semantics

- Providing Failure Atomicity for Granularity Beyond Persist Access

Logging Mechanism
(Undo Logging)
Semantics for Delivering Persistency
Challenge #1: Correctness

Prior Work

Atlas [Chakrabarti' 14]
Coupled/Decoupled SFR [Gogte' 18]
Semantics for Delivering Persistency
Challenge #1: Correctness

Prior Work

Atlas [Chakrabarti' 14]
Coupled/Decoupled SFR [Gogte' 18]

Detect Persist Updates at Compile Time
Semantics for Delivering Persistency
Challenge #1: Correctness

Prior Work

Atlas [Chakrabarti' 14]
Coupled/Decoupled SFR [Gogte' 18]

Detect Persist Updates at **Compile Time**

What Can Go Wrong?
Challenge #1: One Possible Scenario

```c
void* p; // Pointer to Persistent Region of Memory
...
foo(p);
...
```

```c
foo(void *pr){
    ...
    *pr = value;
    ...
}
```
Challenge #1: One Possible Scenario

void* p;  // Pointer to Persistent Region of Memory
...
foo(p);
...

foo(void *pr){
  ...
  *pr = value;
  ...
}

Failure-Atomicity Region

Update Not Detectable at Compile Time
Semantics for Delivering Persistency
Challenge #2: Performance

High Logging Overheads (Atlas and Coupled SFR)
Semantics for Delivering Persistency
Challenge #2: Performance

High Logging Overheads (Atlas and Coupled SFR)

Decoupled SFR:
Background Threads for Persisting Data and Committing Log
Semantics for Delivering Persistency
Challenge #2: Performance

High Logging Overheads (Atlas and Coupled SFR)

Decoupled SFR: Background Threads for Persisting Data and Committing Log

❌ Utilization
Idea #1: Persistency for the Whole Program

Application

Unrecoverable

Baseline: No Recoverability

Code Visible at Compile Time

Code Not Visible at Compile Time
Idea #1: Persistency for the Whole Program

Application

Unrecoverable

Code Visible at Compile Time

Code Not Visible at Compile Time

Baseline: No Recoverability

Recoverable

Code Visible at Compile Time

Code Not Visible at Compile Time

Prior Work: Partial Recoverability
Idea #1: Persistency for the Whole Program

Application

- **Unrecoverable**
  - Code Visible at Compile Time
  - Code Not Visible at Compile Time

- **Recoverable**
  - Code Visible at Compile Time
  - Code Not Visible at Compile Time

Baseline: No Recoverability
Prior Work: Partial Recoverability
Our Goal: Complete Recoverability
Idea #1: Persistency for the Whole Program

Unrecoverable

- Code Visible at Compile Time
- Code Not Visible at Compile Time

Baseline: No Recoverability

Recoverable

- Code Visible at Compile Time
- Code Not Visible at Compile Time

Prior Work: Partial Recoverability

- Code Visible at Compile Time
- Code Not Visible at Compile Time

Our Goal: Complete Recoverability

But How?
Idea #1: Persistency for the Whole Program

**But How?**

**Binary Instrumentation**
Idea #2: More Optimization in Logging

Baseline: No Logging

Prior Work: Serial Logging

Our Goal: Batch Logging

- $U_1 = \text{createLog}(&A, A)$
  - \text{flush}(U_1)
  - FENCE
    - St A
  - \text{flush}(A)$

- $U_2 = \text{createLog}(&B, B)$
  - \text{flush}(U_2)
  - FENCE
    - St B
  - \text{flush}(B)$

- $U_3 = \text{createLog}(&C, C)$
  - \text{flush}(U_3)
  - FENCE
    - St C
  - \text{flush}(C)$

- $U_1 = \text{createLog}(&A, A)$
  - \text{flush}(U_1)
  - \text{flush}(U_2)
  - \text{flush}(U_3)$

- \text{FENCE}
  - St A
  - \text{flush}(A)$
  - St B
  - \text{flush}(B)$
  - St C
  - \text{flush}(C)$
Idea #2: More Optimization in Logging

Can Hardware Help Us?

Baseline: No Logging

Prior Work: Serial Logging

Our Goal: Batch Logging

$U_1 = \text{createLog}(&A, A)$
$\text{flush}(U_1)$
$\text{FENCE}$
$\text{St A}$
$\text{flush}(A)$

$U_2 = \text{createLog}(&B, B)$
$\text{flush}(U_2)$
$\text{FENCE}$
$\text{St B}$
$\text{flush}(B)$

$U_3 = \text{createLog}(&C, C)$
$\text{flush}(U_3)$
$\text{FENCE}$
$\text{St C}$
$\text{flush}(C)$
Evaluation Methodology

Tools:

• LLVM [Lattner' 04]
• DynamoRIO [Bruening' 12]

Benchmarks:

• WHISPER [Nalli' 17]
• Real-word Persistence Applications
Correct and Fast Persistency Guarantees

Sara Mahdizadeh Shahri
Aasheesh Kolli

YArch Workshop - February 2019
Backup Slides
Prior work

- **Atlas**
  - Ensure Atomicity Within
  - Outermost Critical Section

- **Coupled SFR**
  - Ensure atomicity within
  - Synchronization Free Regions

```
lock1_acquire()
lock2_acquire()
lock2_release()
lock1_release()
```

SFRs: Synchronization Free Regions