Persistency for Synchronization-Free Regions

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

PLDI 2018

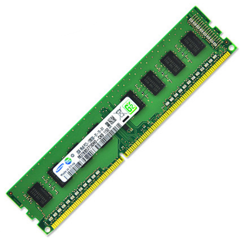
06/20/2018

UNIVERSITY OF MICHIGAN

arm [$]

Promise of persistent memory (PM)

Performance



+

Density

+

Non-volatility

Intel Announces New Optane DC Persistent Memory *

By Joel Hruska on May 31, 2018 at 8:15 am | 1 Comment

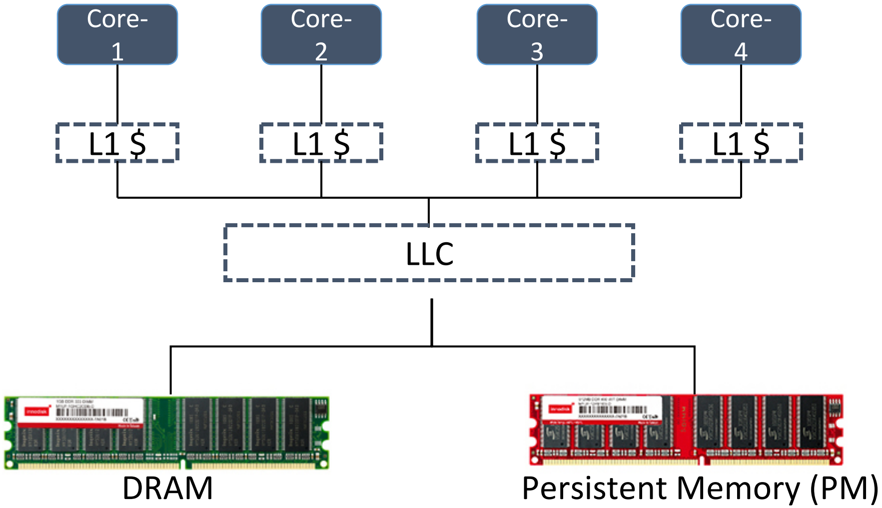
"Optane DC Persistent Memory will be offered in packages of up to 512GB per stick."

"… expanding memory per CPU socket to as much as 3TB."

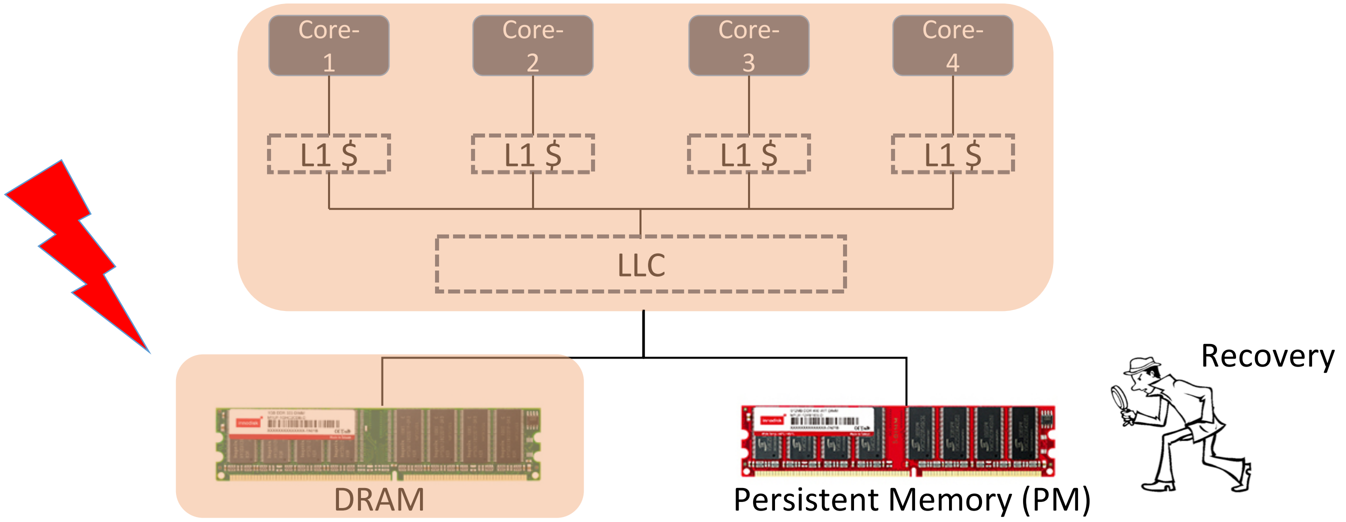
* Source: www.extremetech.com

Byte-addressable, load-store interface to storage

Persistent memory system



Persistent memory system



Recovery can inspect the data-structures in PM to restore system to a consistent state

Memory persistency models

- Provide guarantees required for recoverable software
  - Academia [Condit '09][Pelley '14][Joshi '15][Kolli '17] …
  - Industry [Intel '14][ARM '16]
- Define the program state that recovery observes post failure
- Key primitives:
  - Ensure failure atomicity for a group of persists
  - Govern ordering constraints on memory accesses to PM

Contributions

- Persistency model using synchronization-free regions
  - Define precise state of the program post failure
  - Employ existing synchronization primitives in C++
- Provide semantics as a part of language implementation
  - Build compiler pass that emits logging code for persistent accesses
- Propose two implementations: Coupled-SFR and Decoupled-SFR
- Achieve 65% better performance over state-of-the-art tech.

Outline

- Design space for language-level persistency models
- Our proposal: Persistency for SFRs
  - Coupled-SFR implementation
  - Decoupled-SFR implementation
- Evaluation

Language-level persistency models [Chakrabarti '14][Kolli '17]

- Enables writing **portable, recoverable** software
- Extend language memory-model with persistency semantics
- Persistency model guarantees:

```
Atomic_begin()
    A = 100;
    B = 200;
Atomic_end()
```

Failure-atomicity:
Which group of stores persist atomically?

```
A = 100;
L1.rel();
                L1.acq();
                A = 200;
```

$St(A) <_p St(B)$

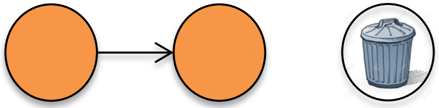
Ordering:
How can programmers order accesses?

Why failure-atomicity?

Task: Fill node and add to linked list, safely

In-memory data

Initial linked-list



MichiganEngineering

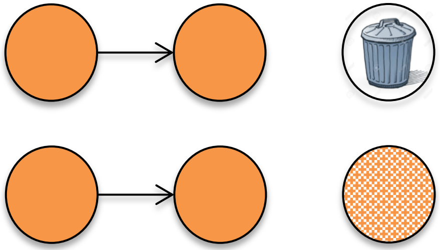
Why failure-atomicity?

Task: Fill node and add to linked list, safely

In-memory data

Initial linked-list

fillNewNode()



Why failure-atomicity?

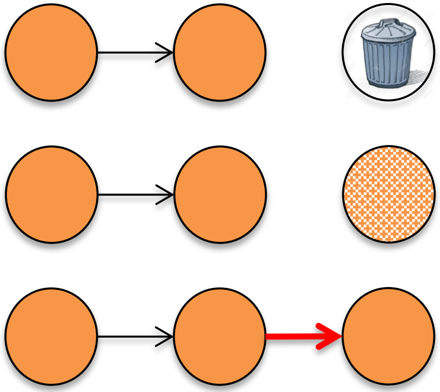
Task: Fill node and add to linked list, safely

In-memory data

Initial linked-list

fillNewNode()

updateTailPtr()



Why failure-atomicity?

Task: Fill node and add to linked list, safely

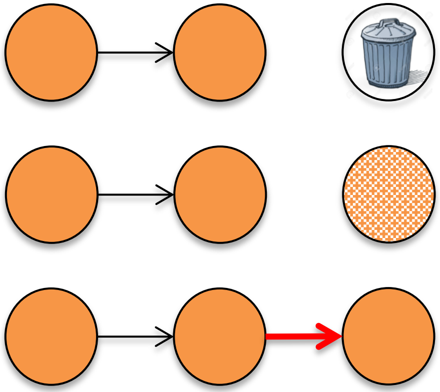
In-memory data

Initial linked-list

fillNewNode()

Fence

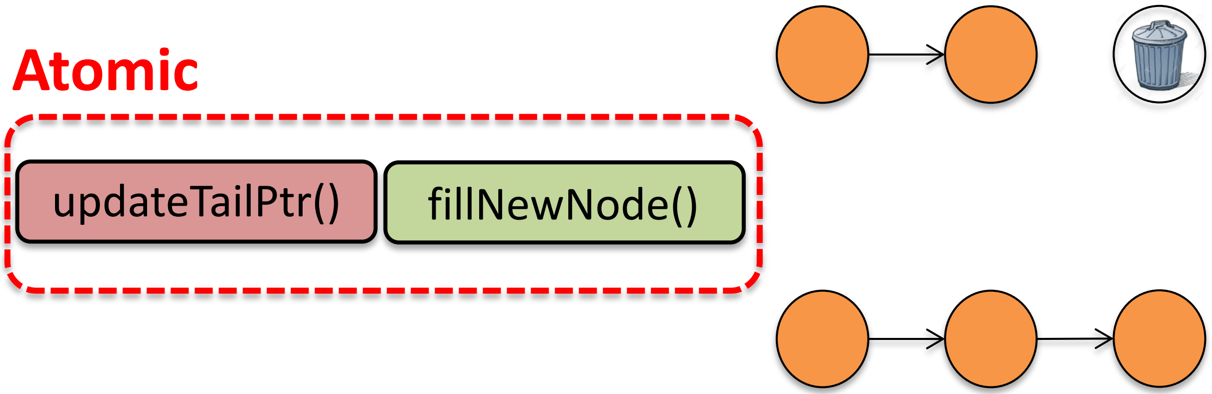
updateTailPtr()



Why failure-atomicity?

Task: Fill node and add to linked list, safely

In-memory data



Failure-atomicity → Persistent memory programming easier

Semantics for failure-atomicity

- Assures that either all or none of the updates visible post failure
- Guaranteed by hardware, library or language implementations
- Design space for semantics
  - Individual persists
  - Outer critical-sections

Existing mechanisms provide unsatisfying semantics or suffer high performance overhead

Granularity of failure-atomicity - I

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

MichiganEngineering arm

Granularity of failure-atomicity - I

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

Individual persists [Condit '09][Pelley '14][Joshi '16][Kolli '17]

- Mechanisms ensure atomicity of individual persists
- Non-sequentially consistent state visible to recovery
  → **Need additional custom logging**

Granularity of failure-atomicity - I

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

Individual persists [Condit '09][Pelley '14][Joshi '16][Kolli '17]

- Mechanisms ensure atomicity of individual persists
- Non-sequentially consistent state visible to recovery → **Need additional custom logging**

Granularity of failure-atomicity - II

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

Outer critical sections [Chakrabarti '14][Boehm '16]

- Guarantees recovery to observe SC state
  → Easier to build recovery code

Granularity of failure-atomicity - II

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

Outer critical sections [Chakrabarti ‘14][Boehm ‘16]

- Guarantees recovery to observe SC state
  → Easier to build recovery code
- Require complex dependency tracking between logs
  → **> 2x performance cost**
- Do not generalize to certain sync. constructs
  → *eg. condition variables*

Our proposal: Failure-atomic SFRs

Synchronization free regions (SFR)

Thread regions delimited by **synchronization operations** or **system calls**

```
l1.acq();
    x -= 100;
    y += 100;        SFR1
    l2.acq();
        a -= 100;
        b += 100;    SFR2
    l2.rel();
l1.rel();
```

Enable precise post-failure state with low performance overhead

Guarantees by failure-atomic SFRs

Thread 1

```
l1.acq();
    x -= 100;     } SFR 1
    y += 100;
l1.rel();
```

- Intra-thread guarantees
  - Ensure failure-atomicity of updates within SFR

Guarantees by failure-atomic SFRs

Thread 1

```
l1.acq();
x -= 100;
y += 100;
l1.rel();
```

SFR 1

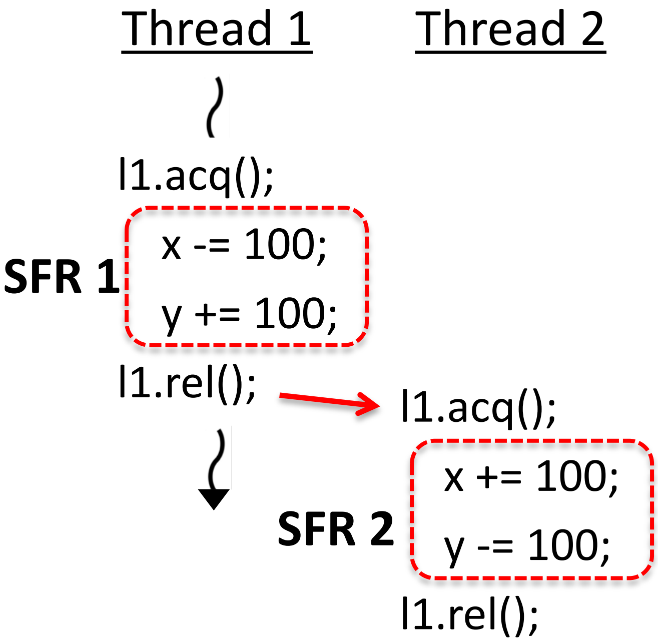
- Intra-thread guarantees
  - Ensure failure-atomicity of updates within SFR
  - Define precise points for post-failure program state

Guarantees by failure-atomic SFRs

Thread 1	Thread 2
l1.acq();	
SFR 1: x -= 100;	
SFR 1: y += 100;	
l1.rel();	
	l1.acq();
	x += 100;
	y -= 100;
	l1.rel();

- Intra-thread guarantees
  - Ensure failure-atomicity of updates within SFR
  - Define precise points for post-failure program state
- Inter-thread guarantees

Guarantees by failure-atomic SFRs

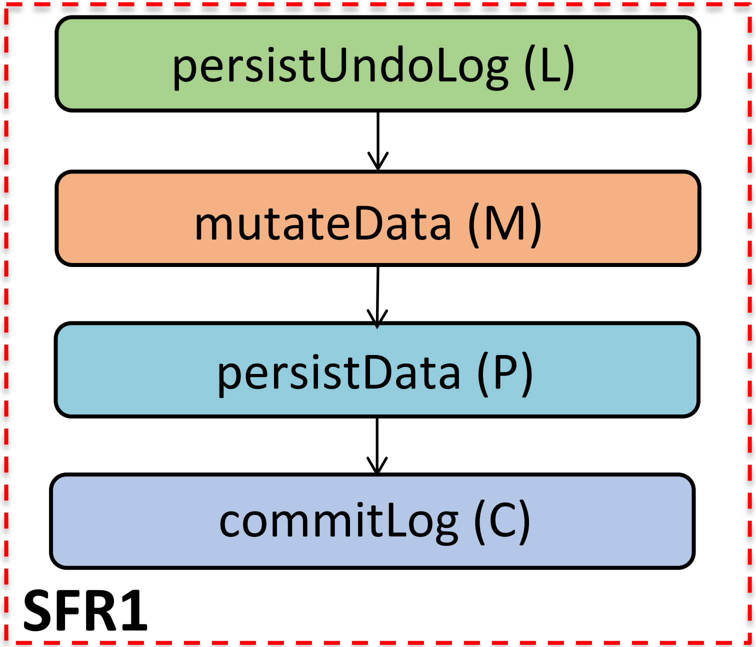


- Intra-thread guarantees
  - Ensure failure-atomicity of updates within SFR
  - Define precise points for post-failure program state
- Inter-thread guarantees
  - Order SFRs using synchronizing *acq* and *rel* ops
  - Serialize ordered SFRs in PM

Two logging impl. → **Coupled-SFR** and **Decoupled-SFR**

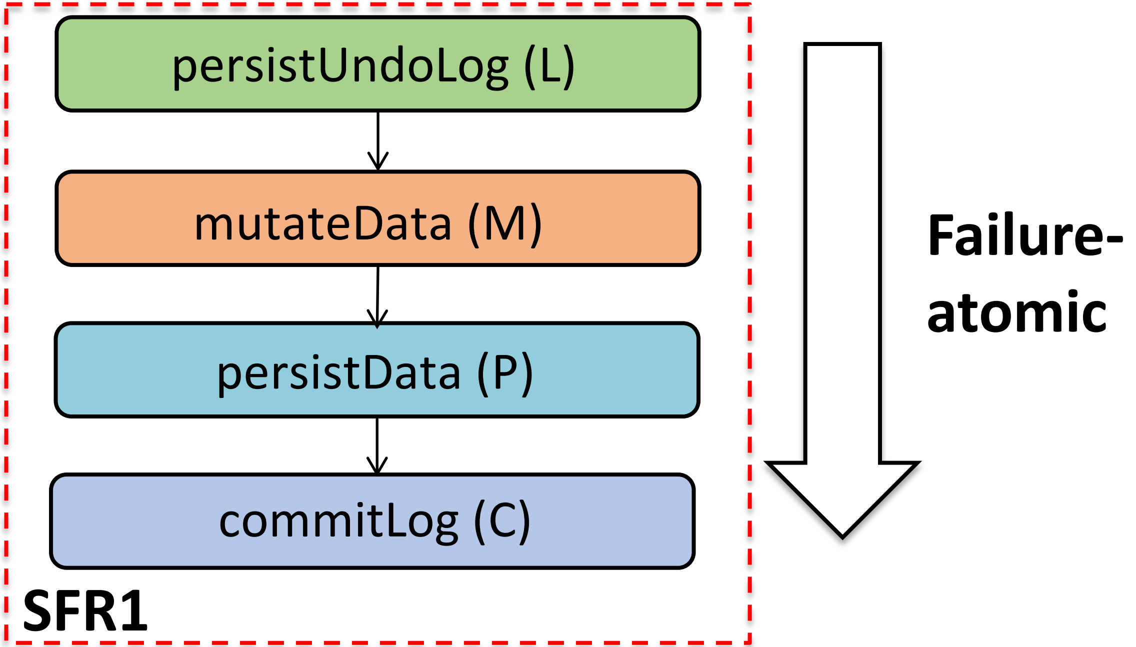
Undo-logging for SFRs

```
        L1.acq();
SFR1      x = 100;
        L1.rel();
```



Undo-logging for SFRs

```
L1.acq();
SFR1    x = 100;
L1.rel();
```



Need to ensure the ordering of steps in undo-logging for SFRs to be failure-atomic

Impl. 1: Coupled-SFR

Thread 1 | Thread 2

L1.acq();

SFR1 x = 100;

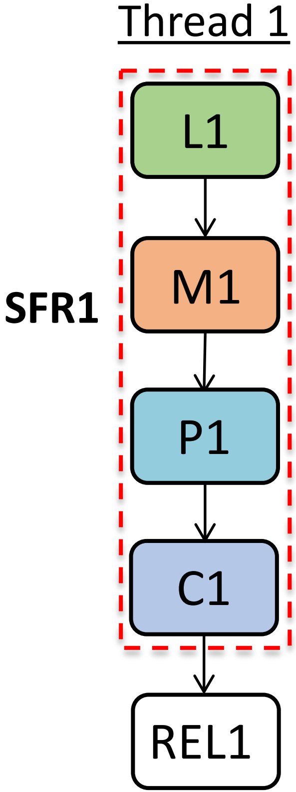
L1.rel(); → L1.acq();

SFR2 x = 200;

L1.rel();

Impl. 1: Coupled-SFR

Thread 1	Thread 2
L1.acq();	
SFR1 x = 100;	
L1.rel(); →	L1.acq();
	SFR2 x = 200;
	L1.rel();

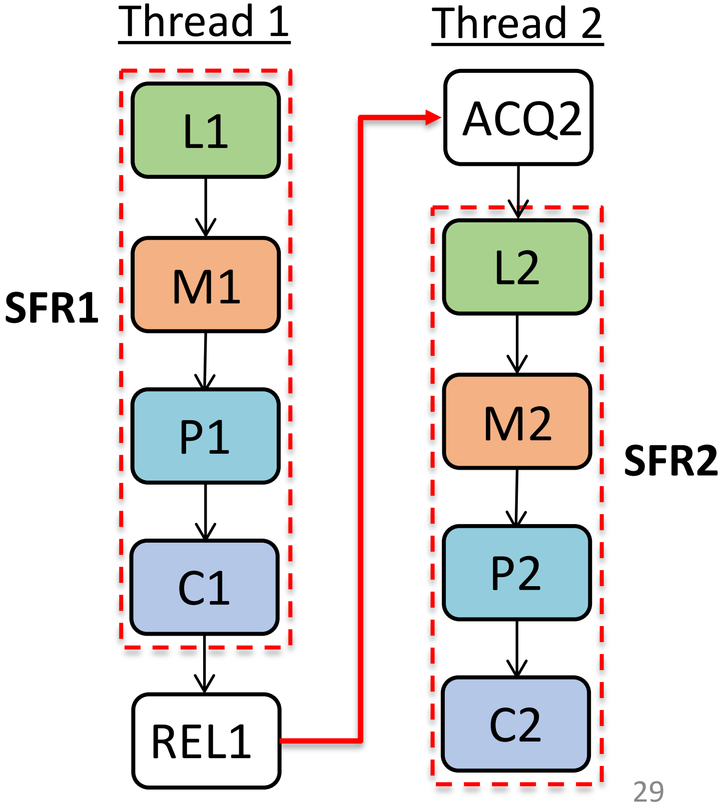


MichiganEngineering

arm

Impl. 1: Coupled-SFR

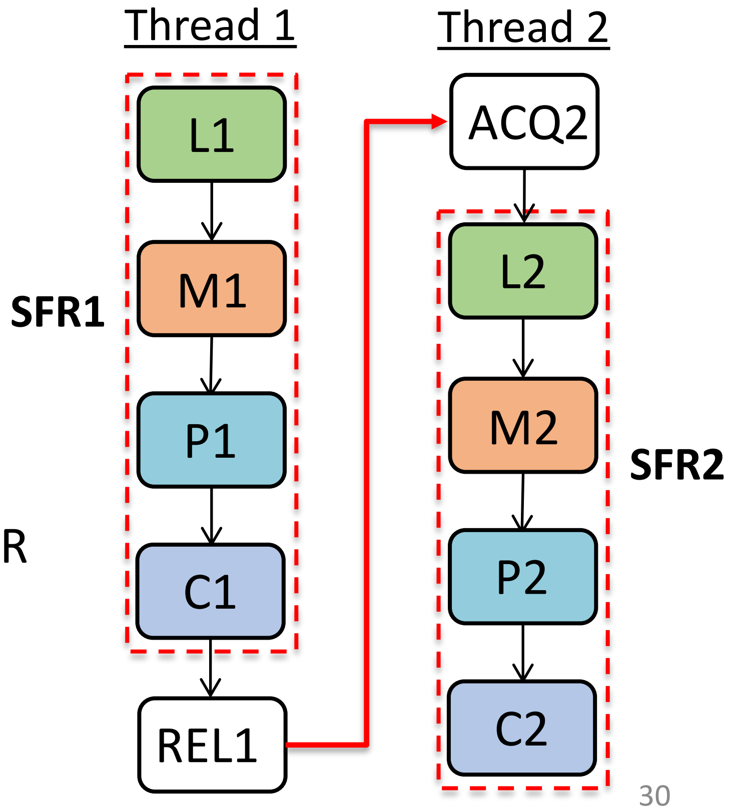
Thread 1	Thread 2
L1.acq();	
SFR1 x = 100;	
L1.rel(); →	L1.acq();
	SFR2 x = 200;
	L1.rel();



Impl. 1: Coupled-SFR

Thread 1 | Thread 2

```
L1.acq();
SFR1  x = 100;
L1.rel();  →  L1.acq();
        SFR2  x = 200;
              L1.rel();
```



+ Persistent state lags execution by at most one SFR

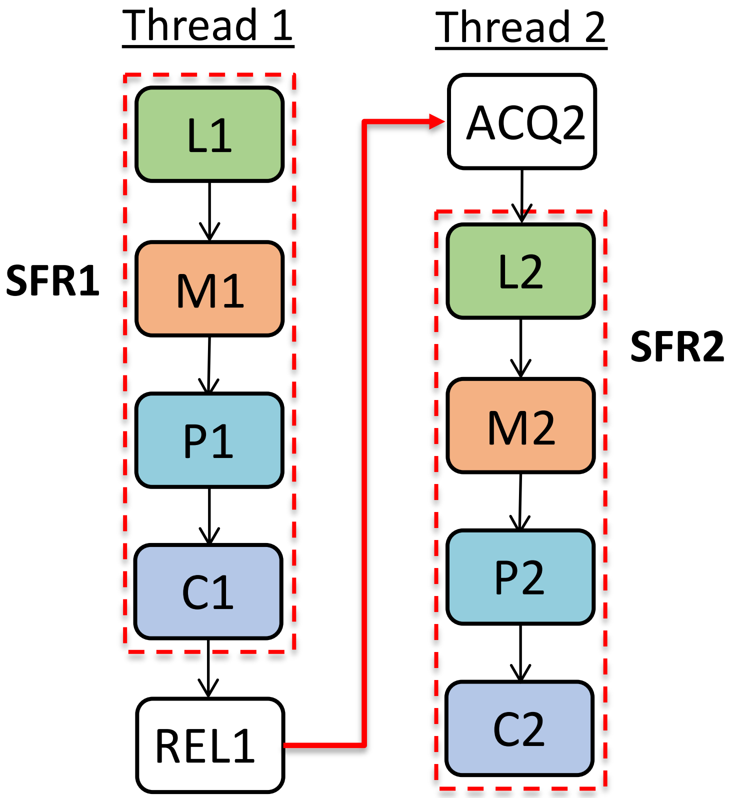
→ **Simpler implementation, latest state at failure**

- Need to flush updates at the end of each SFR

→ **High performance cost**

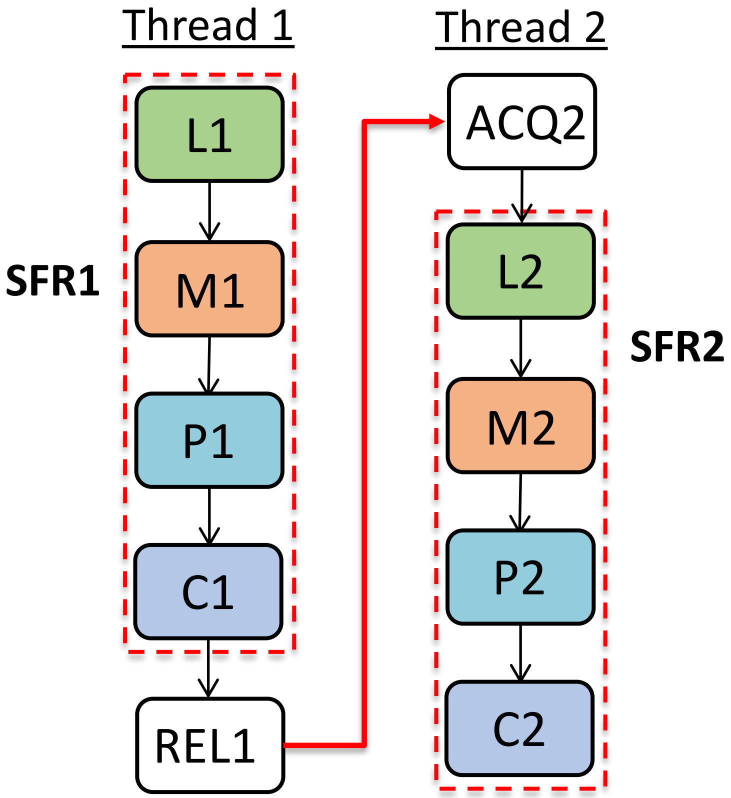
Impl. 2: Decoupled-SFR

Thread 1	Thread 2
L1.acq();	
SFR1 x = 100;	
L1.rel(); →	L1.acq();
	SFR2 x = 200;
	L1.rel();



Impl. 2: Decoupled-SFR

Thread 1	Thread 2
L1.acq();	
SFR1 x = 100;	
L1.rel(); →	L1.acq();
	SFR2 x = 200;
	L1.rel();



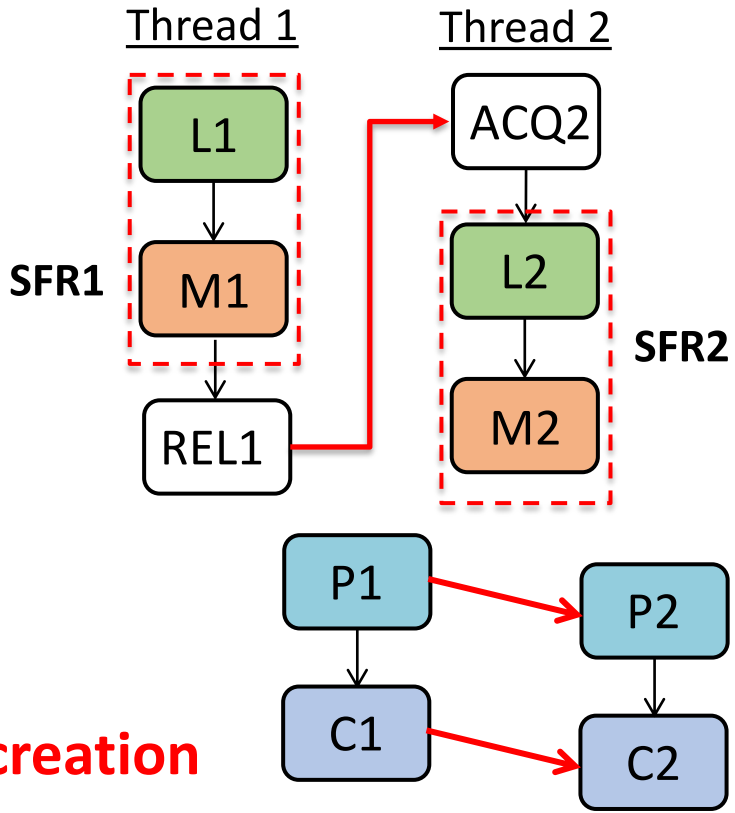
Key idea: Persist updates and commit logs **in background**

MichiganEngineering arm

Impl. 2: Decoupled-SFR

Thread 1 | Thread 2

```
L1.acq();
SFR1   x = 100;
L1.rel();  →
                 L1.acq();
          SFR2   x = 200;
                 L1.rel();
```



Key idea: Persist updates and commit logs **in background**

Require recording order of log creation

Log ordering in Decoupled-SFR

Init x = 0

Thread 1 | Thread 2

x = 100;

L1.rel();

→ L1.acq();

x = 200;

Thread 1 Header

Thread 2 Header

Sequence Table

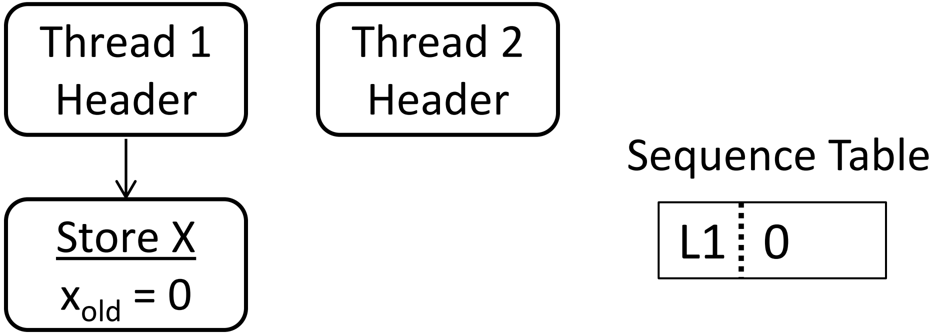
L1	0

Log ordering in Decoupled-SFR

Init x = 0

Thread 1 | Thread 2

```
x = 100;
L1.rel();
          →  L1.acq();
                 x = 200;
```



Log ordering in Decoupled-SFR

Init x = 0

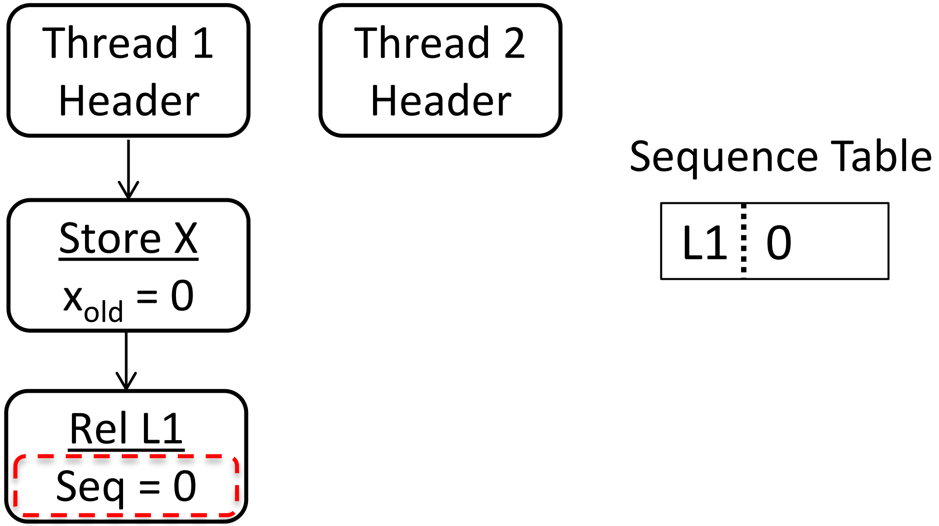
Thread 1 | Thread 2

x = 100;

L1.rel();

→ L1.acq();

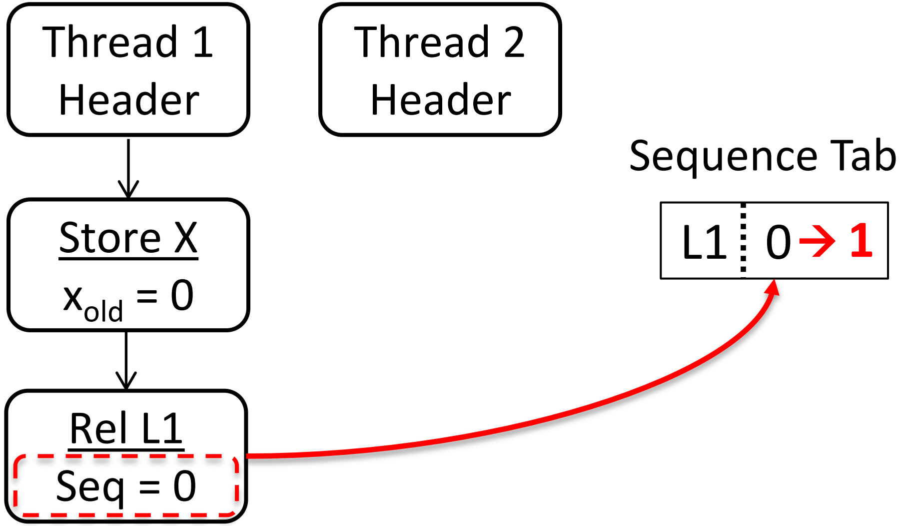
x = 200;



Log ordering in Decoupled-SFR

Init x = 0

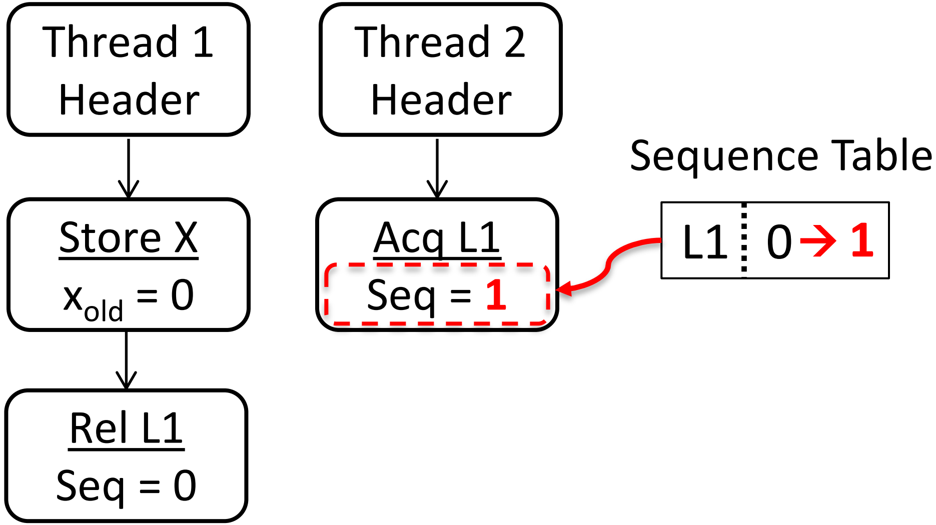
Thread 1	Thread 2
x = 100;	
L1.rel();	
	L1.acq();
	x = 200;



Log ordering in Decoupled-SFR

Init x = 0

Thread 1	Thread 2
x = 100;	
L1.rel();	
	L1.acq();
	x = 200;

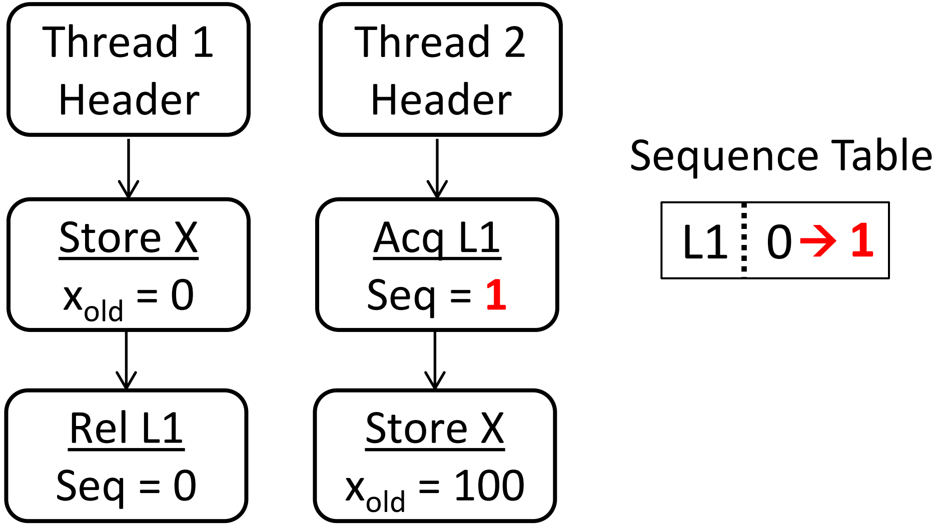


Log ordering in Decoupled-SFR

Init x = 0

Thread 1 | Thread 2

```
x = 100;
L1.rel();
            L1.acq();
                x = 200;
```



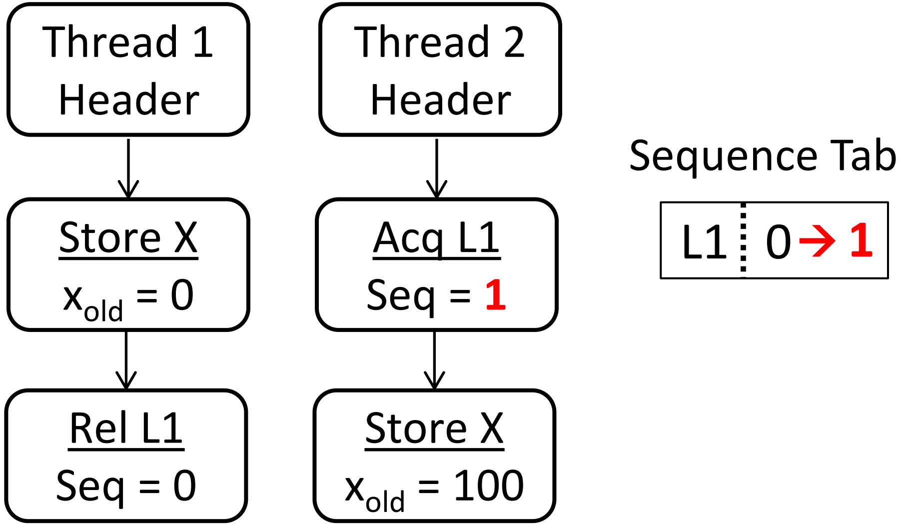
Log ordering in Decoupled-SFR

Init x = 0

Thread 1 | Thread 2

```
Thread 1                Thread 2

    x = 100;
L1.rel();
              --->  L1.acq();
                        x = 200;
```



Sequence numbers record inter-thread order of log creation

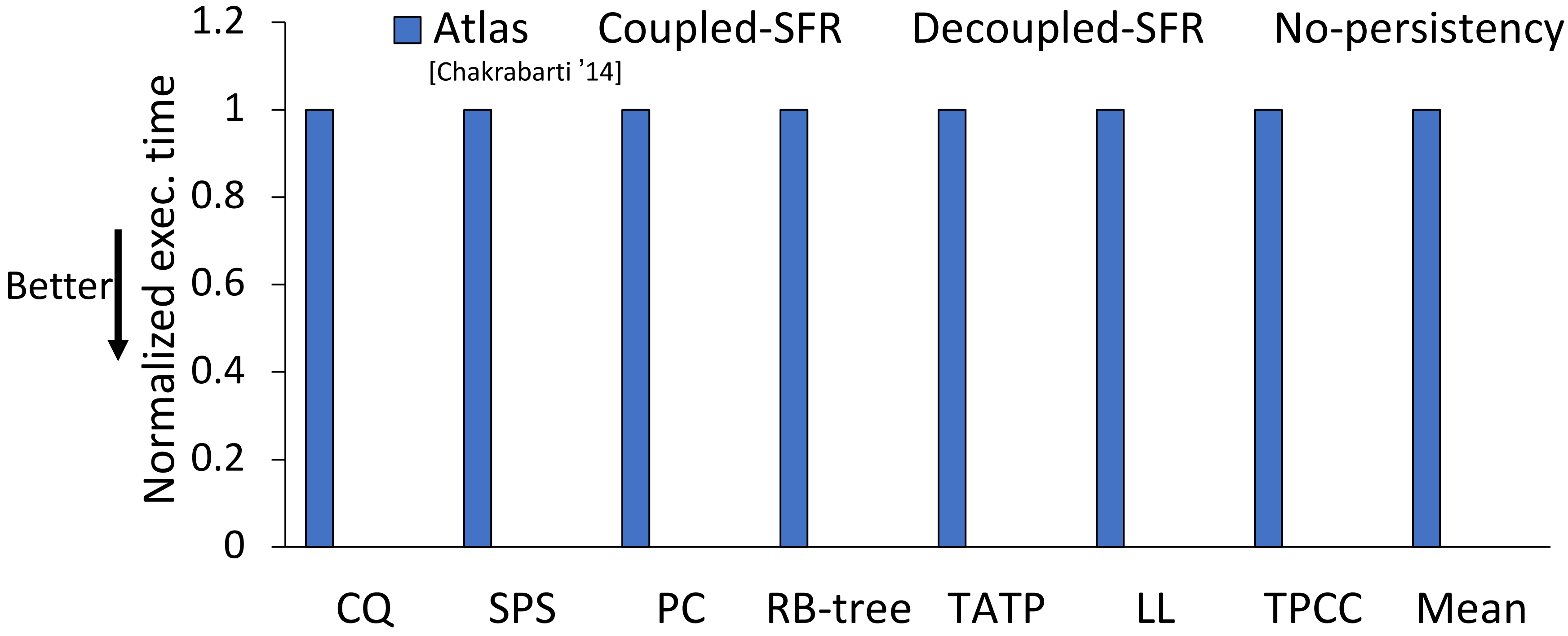
Background threads commit logs using recorded sequence nos.

In paper: Racing sync. operations, commit optimizations etc.

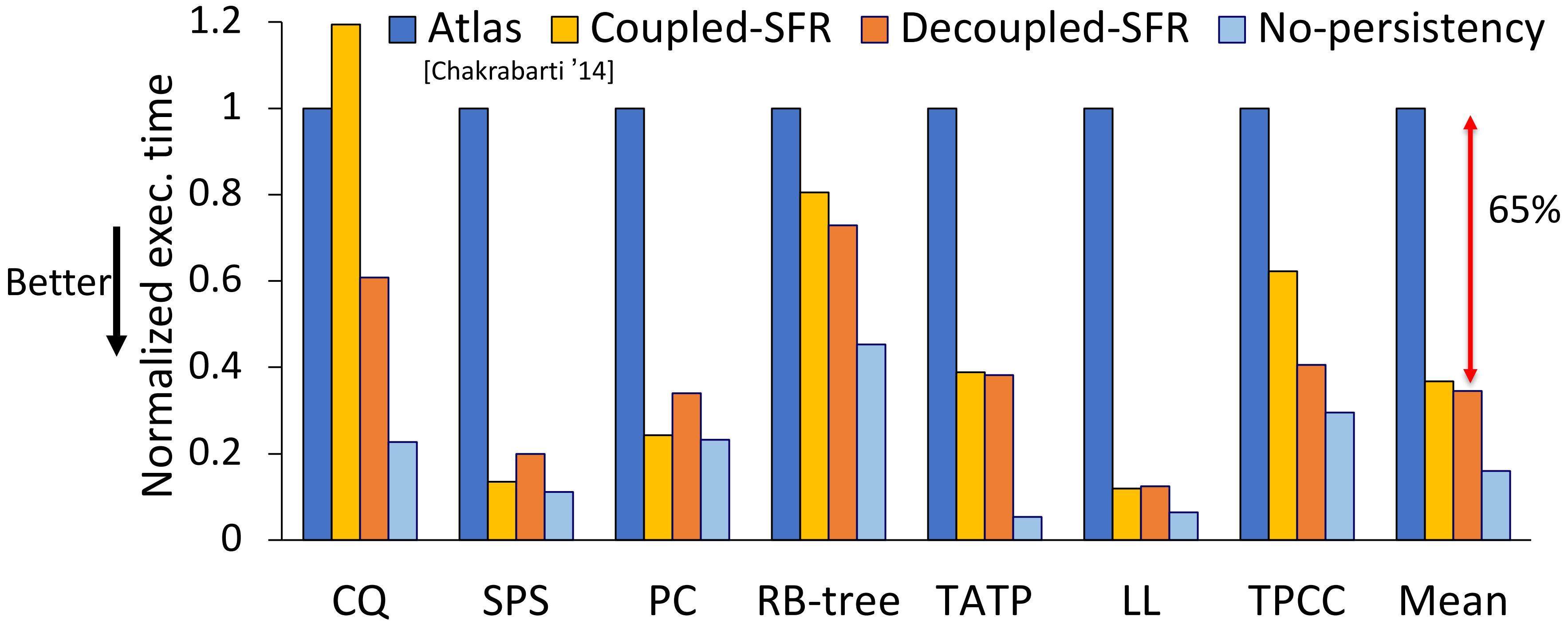
Evaluation setup

- Designed our **logging approaches in LLVM v3.6.0**
  - Instruments stores and sync. ops. to emit undo logs
  - Creates log space for managing per-thread undo-logs
  - Launches background threads to flush/commit logs in Decoupled-SFR
- Workloads: write-intensive micro-benchmarks
  - 12 threads, 10M operations
- Performed experiments on Intel E5-2683 v3
  - 2GHz, 12 physical cores, 2-way hyper-threading

Performance evaluation

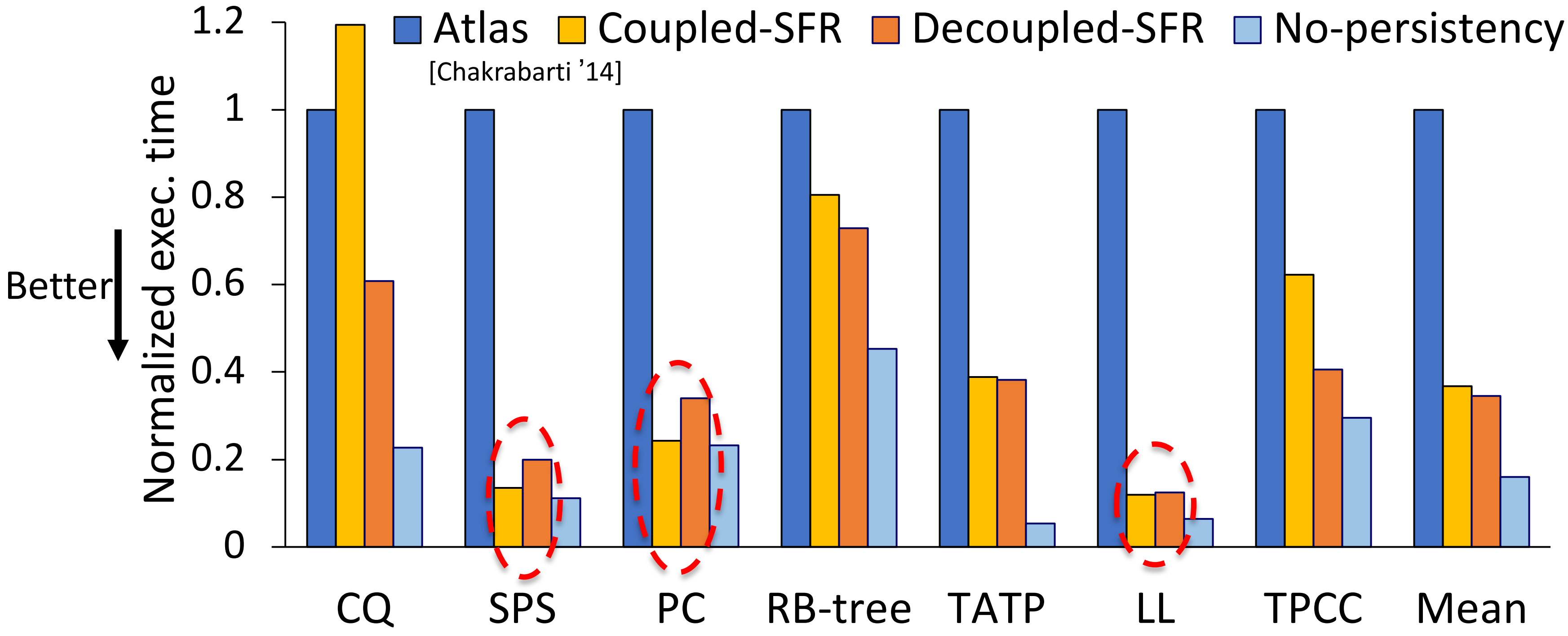


Performance evaluation



Decoupled-SFR performs 65% better than state-of-the-art ATLAS design

Performance evaluation



Coupled-SFR performs better that Decoupled-SFR when fewer stores/SFR

Conclusion

- Failure-atomic synchronization-free regions
  - Persistent state moves from one sync. operation to the next
  - Extends clean SC semantics to post-failure recovery
- Coupled-SFR
  - Easy to reason about PM state after failure; high performance cost
- Decoupled-SFR
  - Persistent state lags execution; performs 65% better than ATLAS

Persistency for Synchronization-Free Regions

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

UNIVERSITY OF MICHIGAN
arm [$]

If the surgery proves unnecessary, we'll revert your architectural state at no charge.[*]

*Source: Paul Kocher, ISCA 2018