Empirical Analysis of Architectural Primitives for NVRAM Consistency

HiPC 2021

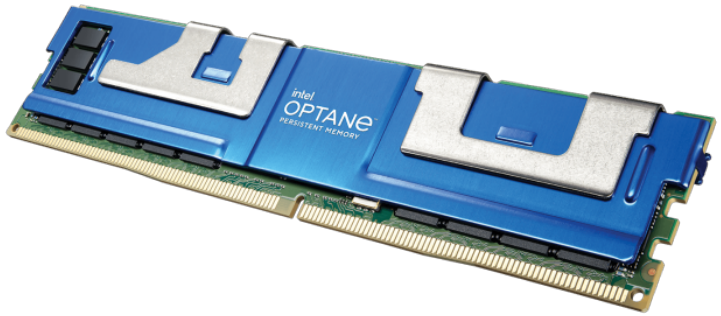
Arun KP[1] Debadatta Mishra [1] Biswabandan Panda[2]

[1] Indian Institute of Technology, Kanpur

[2] Indian Institute of Technology, Bombay

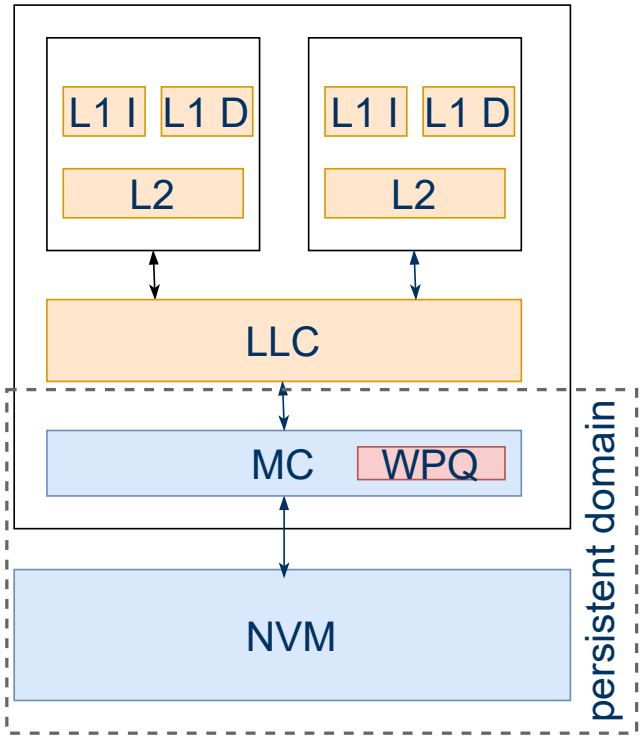
Non-volatile Memory (NVM)

- Persistence of data with better read/write latency.
- Access through load/store interface.
- Provides high memory capacity.



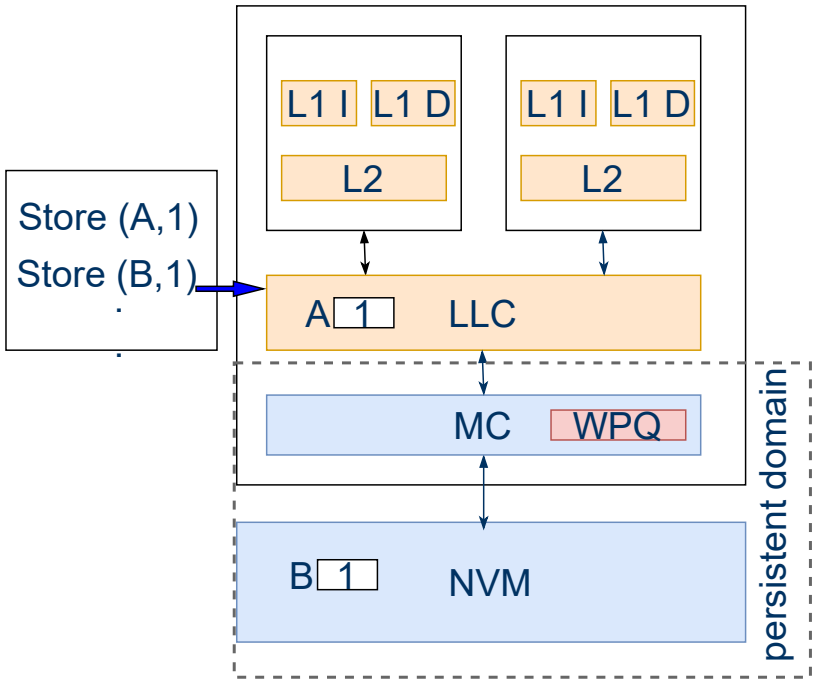
Data Consistency in NVM

```
.
.
store(A, 1)
.
store(B, 1)
.
.
```



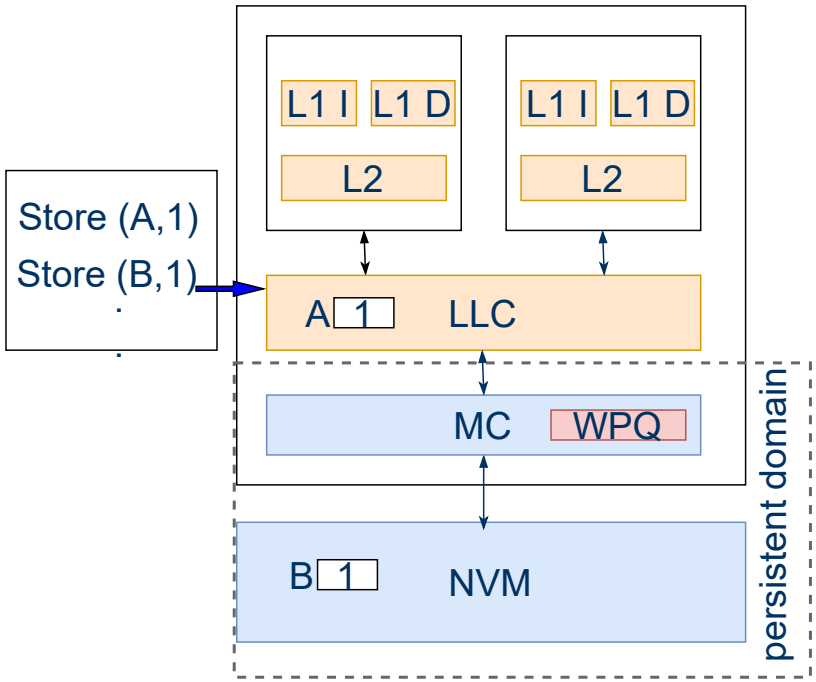
Initial memory state

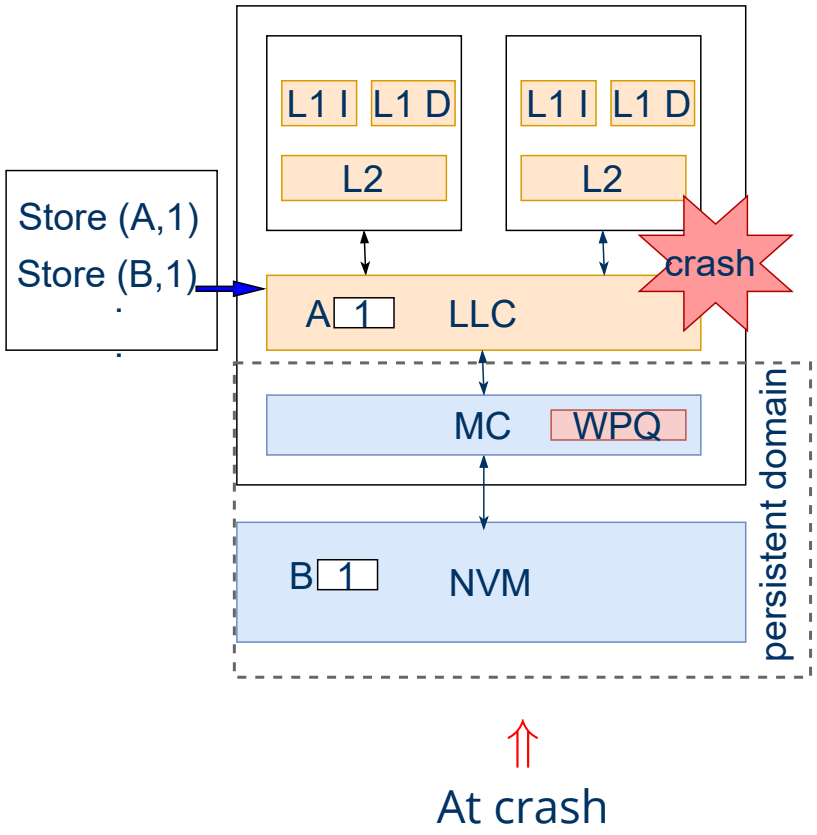
Data Consistency in NVM



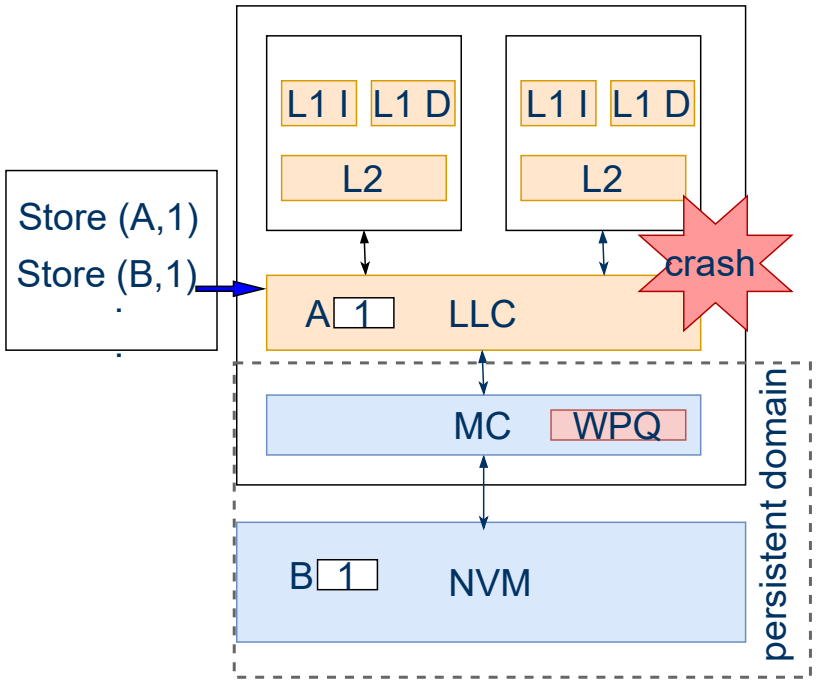
After store to B

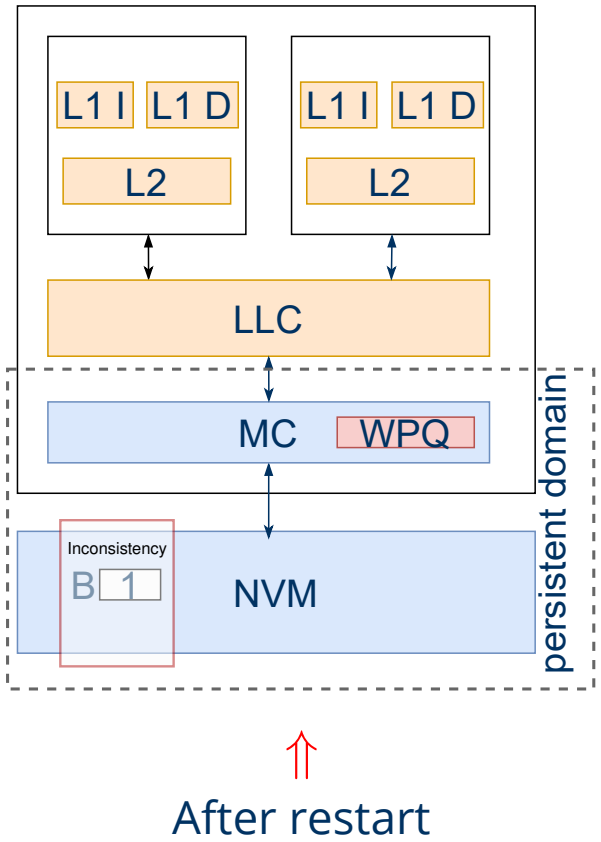
Data Consistency in NVM



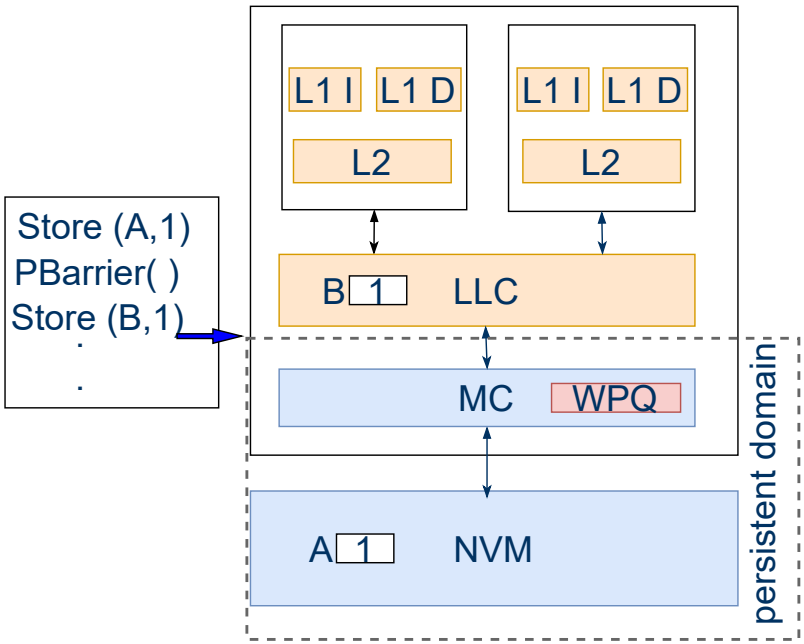


Data Consistency in NVM



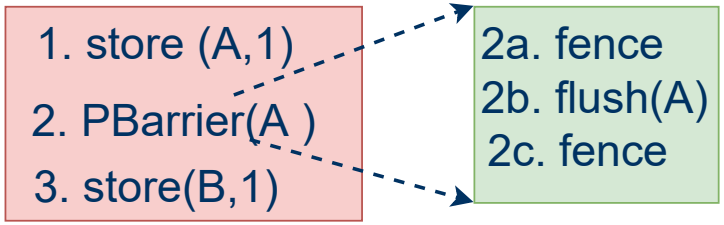


Data Consistency With Persistent Barrier

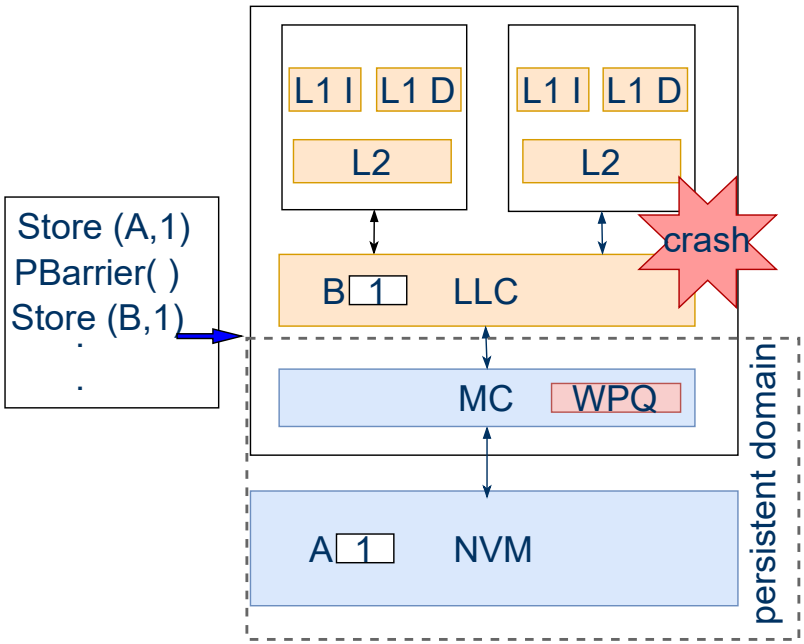


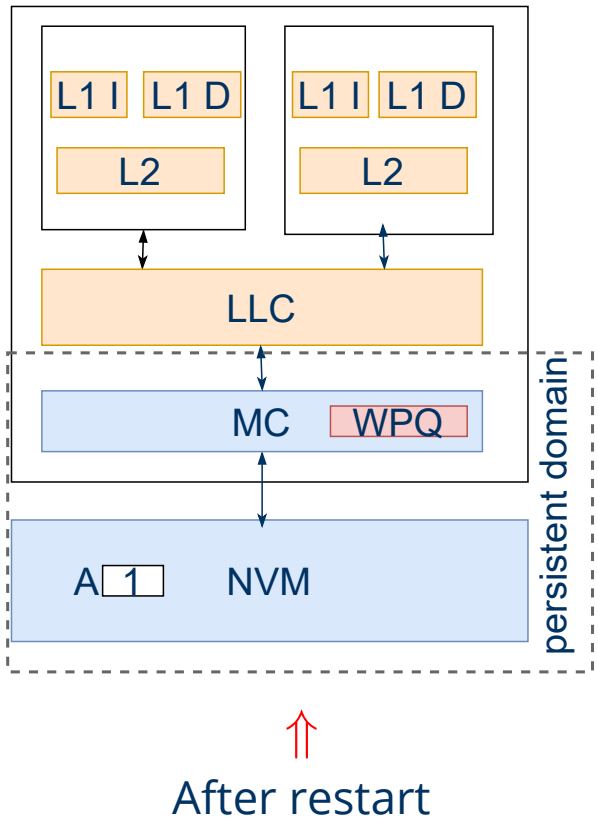
After store to B

Persistent Barrier



Data Consistency With Persistent Barrier



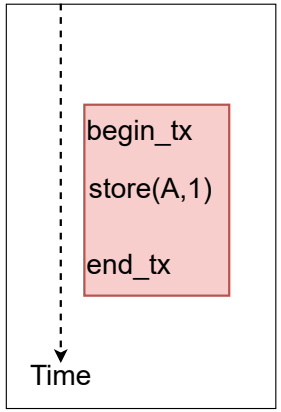


After restart

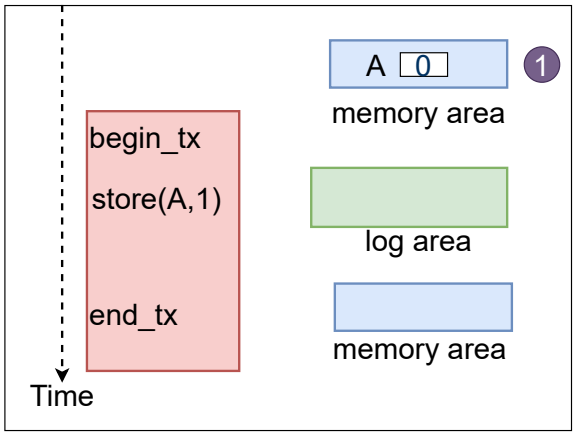
Data Consistency using Logging

- Provide atomicity of a code segment
- Using `begin_tx` & `end_tx` semantics

Redo and undo logging are commonly used



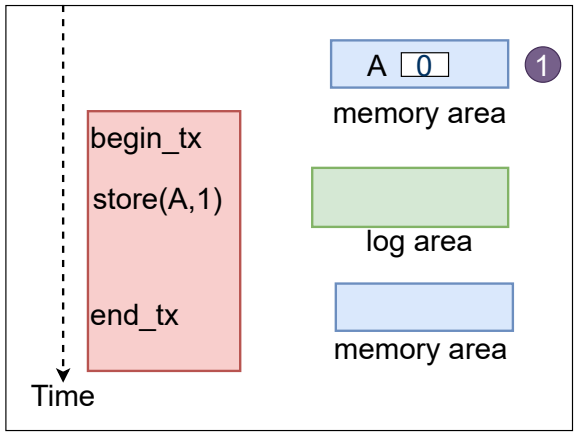
Undo Logging: Example

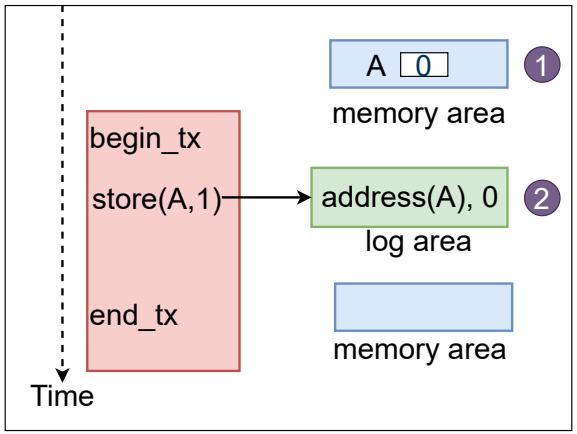


⇑

Memory has A's initial value

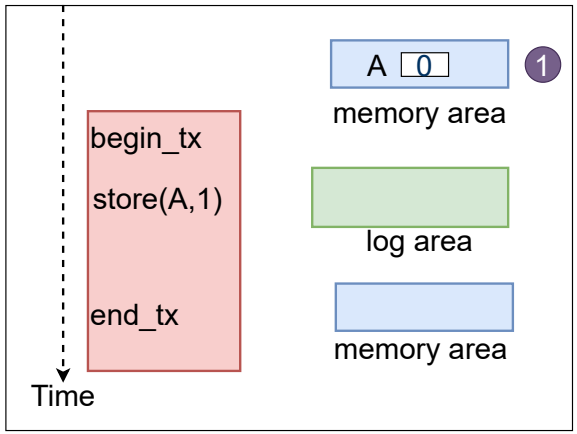
Undo Logging: Example

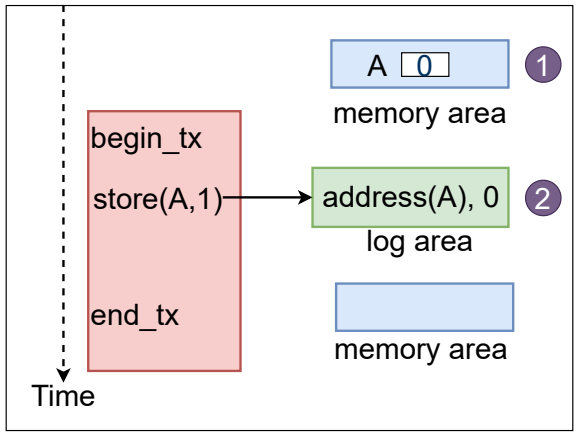


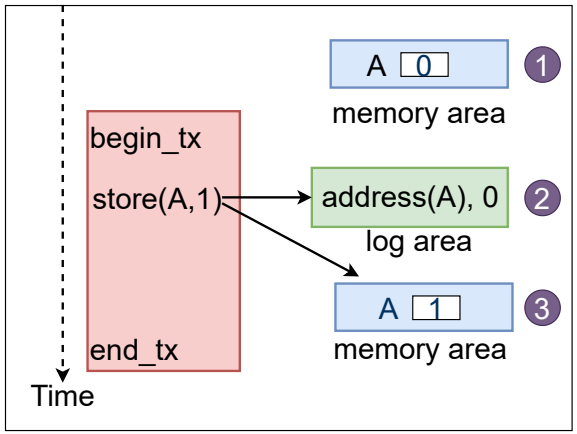


Log has A's initial value

Undo Logging: Example

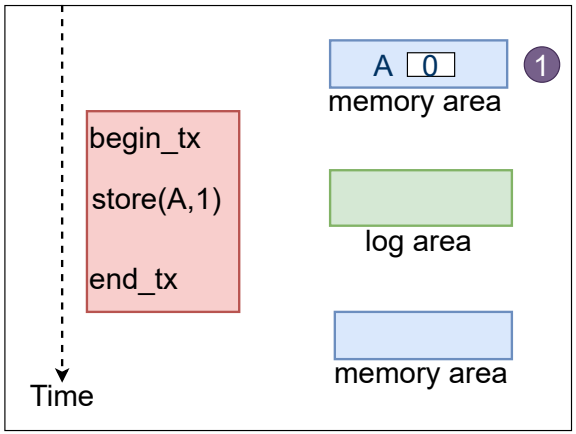






Memory has A's new value

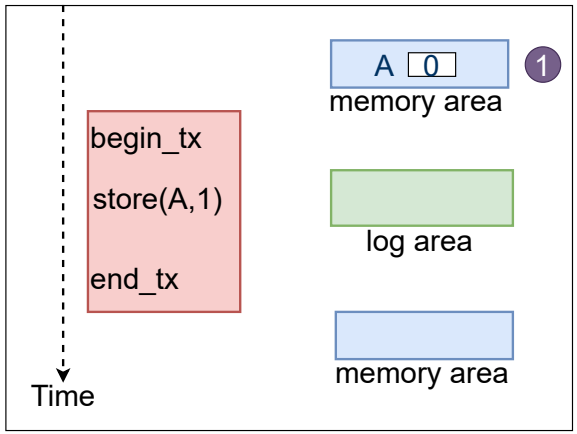
Redo Logging: Example

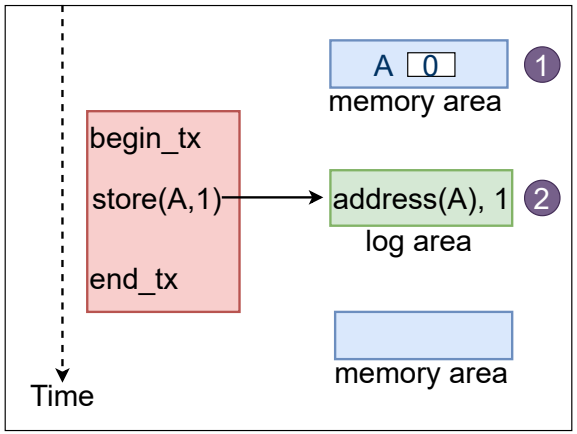


⇑

Memory has A's initial value

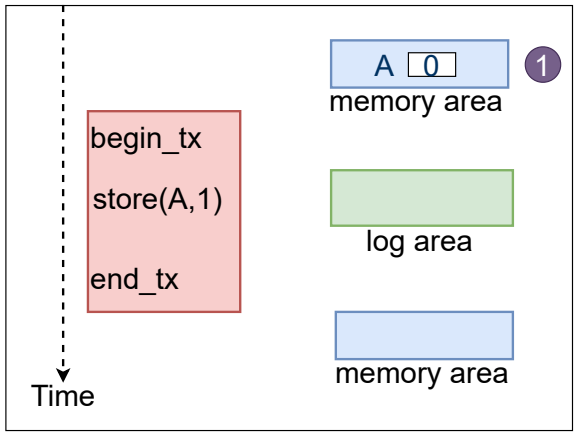
Redo Logging: Example

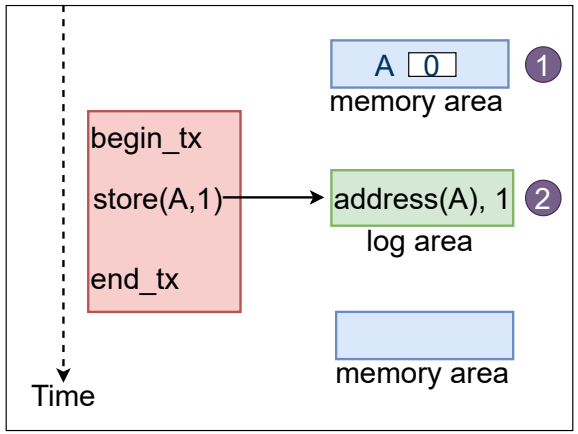


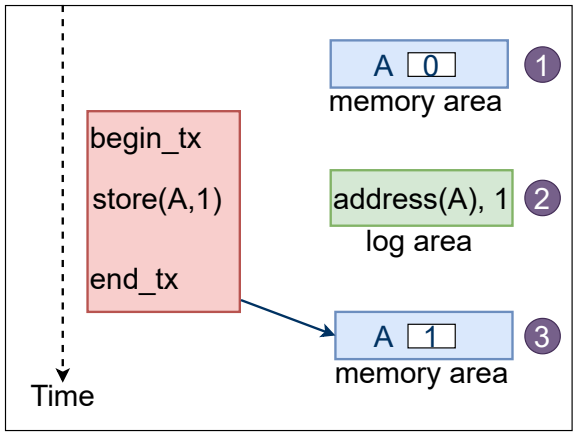


Log has A's new value

Redo Logging: Example







Memory has A's new value

Persistent Barrier: Architecture Primitives

- X86-64
  - clflush ⇐

Persistent Barrier: Architecture Primitives

- X86-64
  - clflush
  - clflushopt ⇐

Persistent Barrier: Architecture Primitives

- X86-64
  - clflush
  - clflushopt
  - clwb ⇐

Persistent Barrier: Architecture Primitives

- X86-64
  - clflush
  - clflushopt
  - clwb
- ARM64
  - civac ⇐

Persistent Barrier: Architecture Primitives

- `X86-64`
  - clflush
  - clflushopt
  - clwb
- `ARM64`
  - civac
  - cvac ⇐

clflush, clflushopt, clwb are ordered by store-fence instructions.
civac, cvac use data memory barrier to the inner shareable domain (DSB ISH) as fence.

Questions

- Performance overhead of different architecture primitives for `X86-64` and `ARM64`.

Questions

- Performance overhead of different architecture primitives for `X86-64` and `ARM64`.
- Application memory footprint, read-to-write ratio influence on performance.

Questions

- Performance overhead of different architecture primitives for `X86-64` and `ARM64`.
- Application memory footprint, read-to-write ratio influence on performance.
- Redo, Undo logging performance with different architecture primitives.

Performance Overhead of Architecture Primitives for X86-64 and ARM64

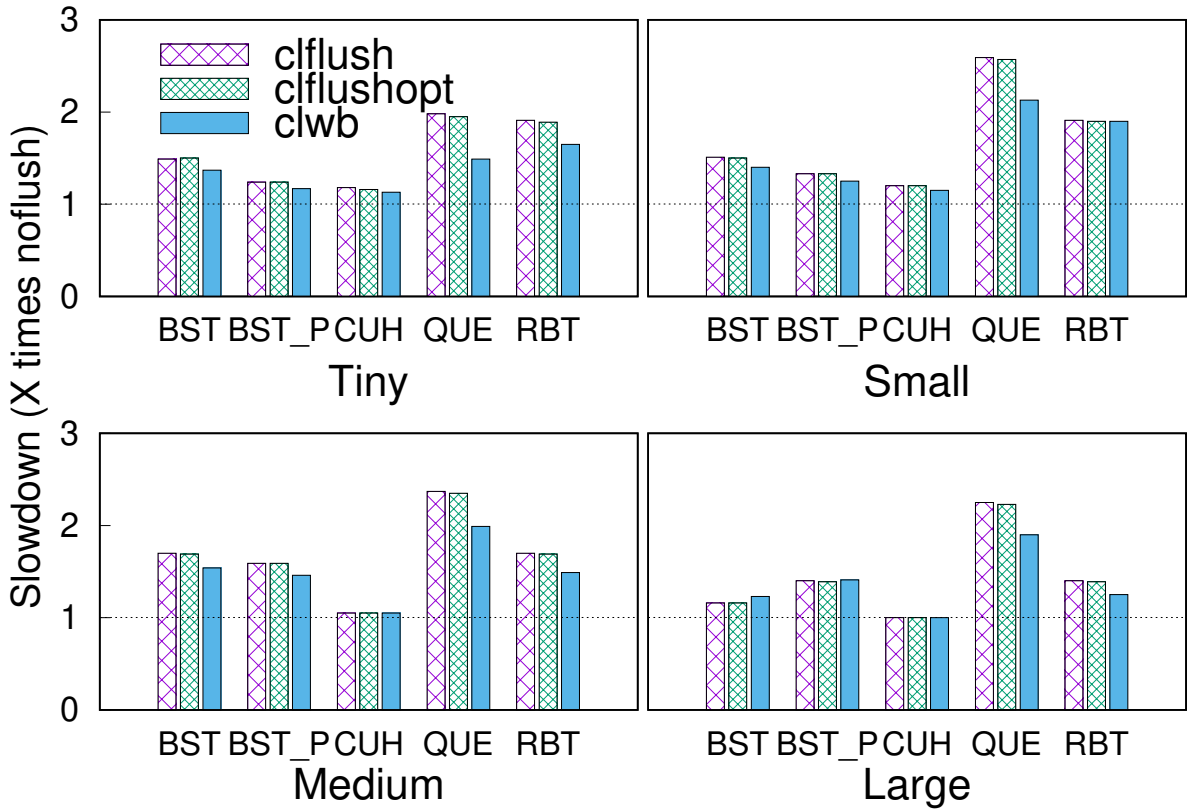
Gem5 configuration

CPU	Out-of-order CPU
L1-D/I	32 KiB/core (8 way)
L2	512KiB/core (16 way)
L3	2 MiB/core shared (16 way)
MSHRs	16, 32, 32/core at L1-D, L2, L3
Cache data access latency	2, 9, 15 cycles at L1-D, L2, L3
Cache line size	64 B in L1, L2, L3
Replacement policy	LRU
L1 prefetcher	StridePrefetcher with degree=4
Memory controller	Nvmain*[1]
Memory	PCM with configuration[2]
Memory capacity	10 GB (20GB for `X86-64` redo/undo result)
*Gem5 NVM Interface for `X86-64` redo/undo result	

Micro-benchmarks

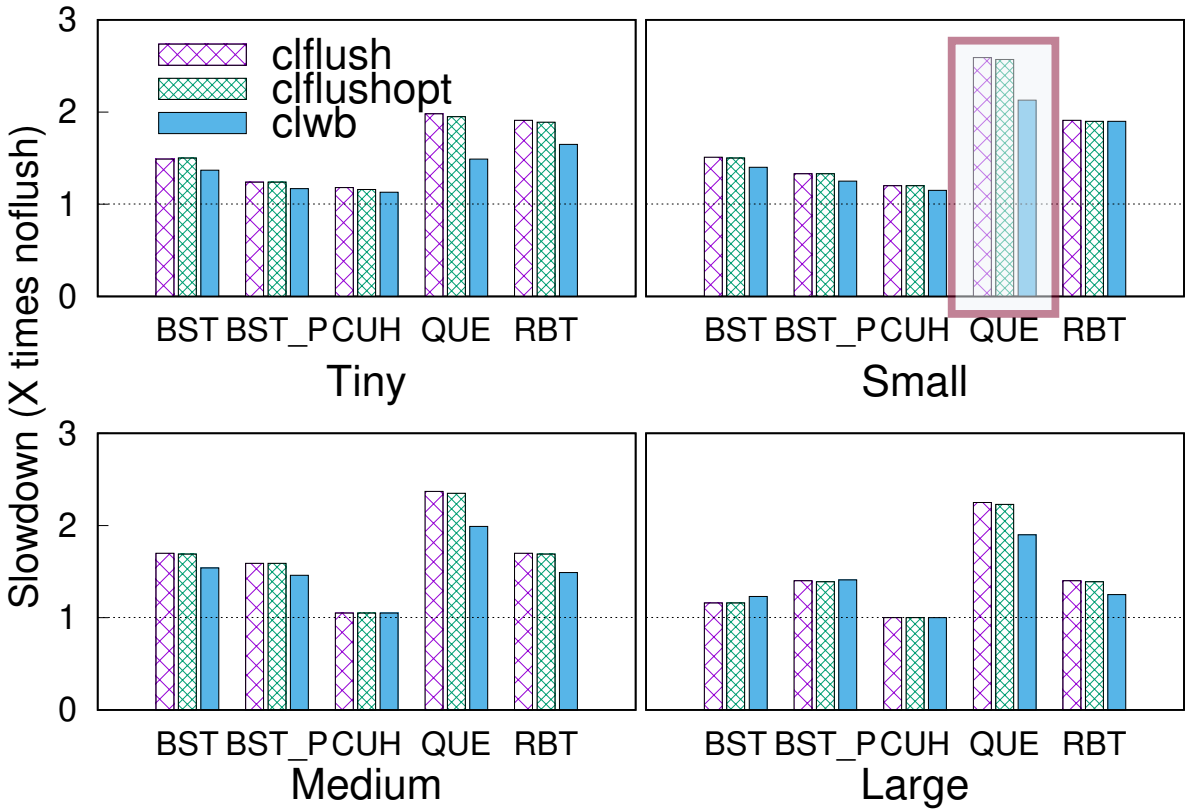
Benchmarks	Description
BST	Binary Search Tree
BST_P	Parallel Binary Search Tree
CUH	Cuckoo Hashing Table
QUE	Linear Queue
RBT	Red-black Tree

Performance with X86-64



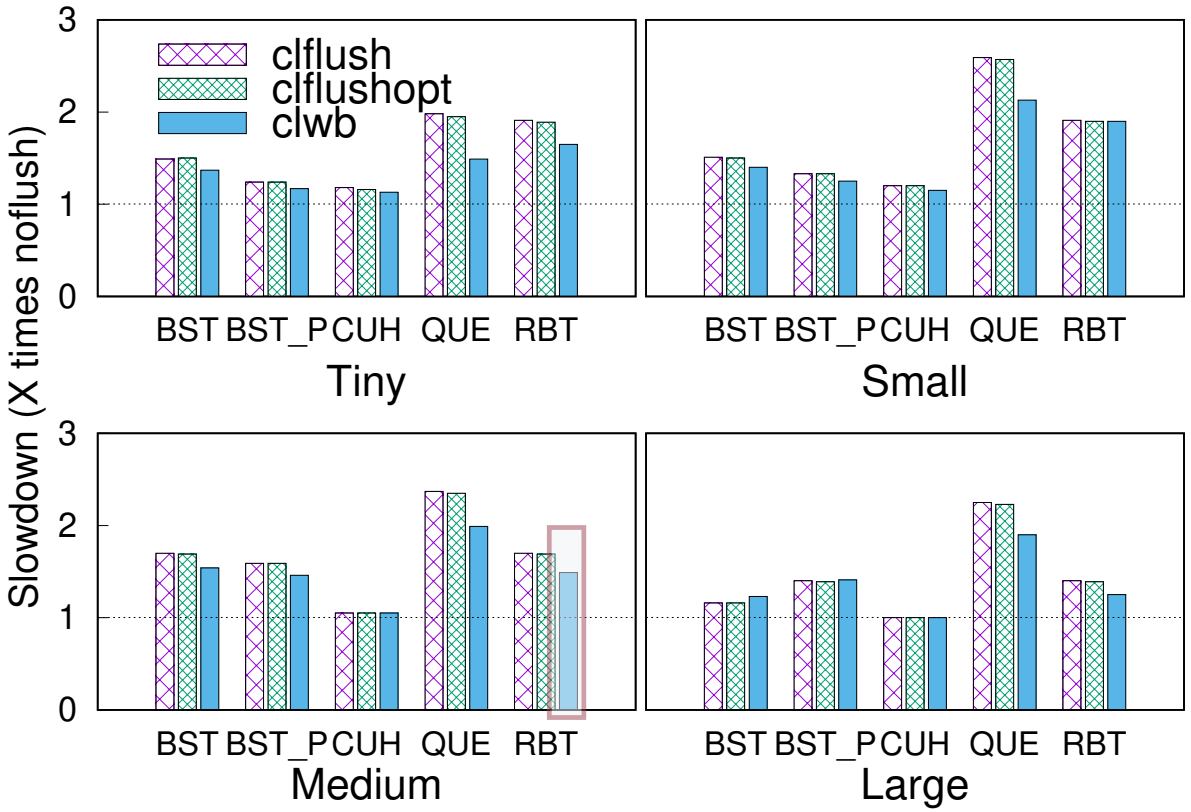
Tiny	0.90 x L1-D Size	Small	0.90 x L2 Size	Medium	0.90 x LLC Size	Large	4 x LLC Size

Performance with X86-64



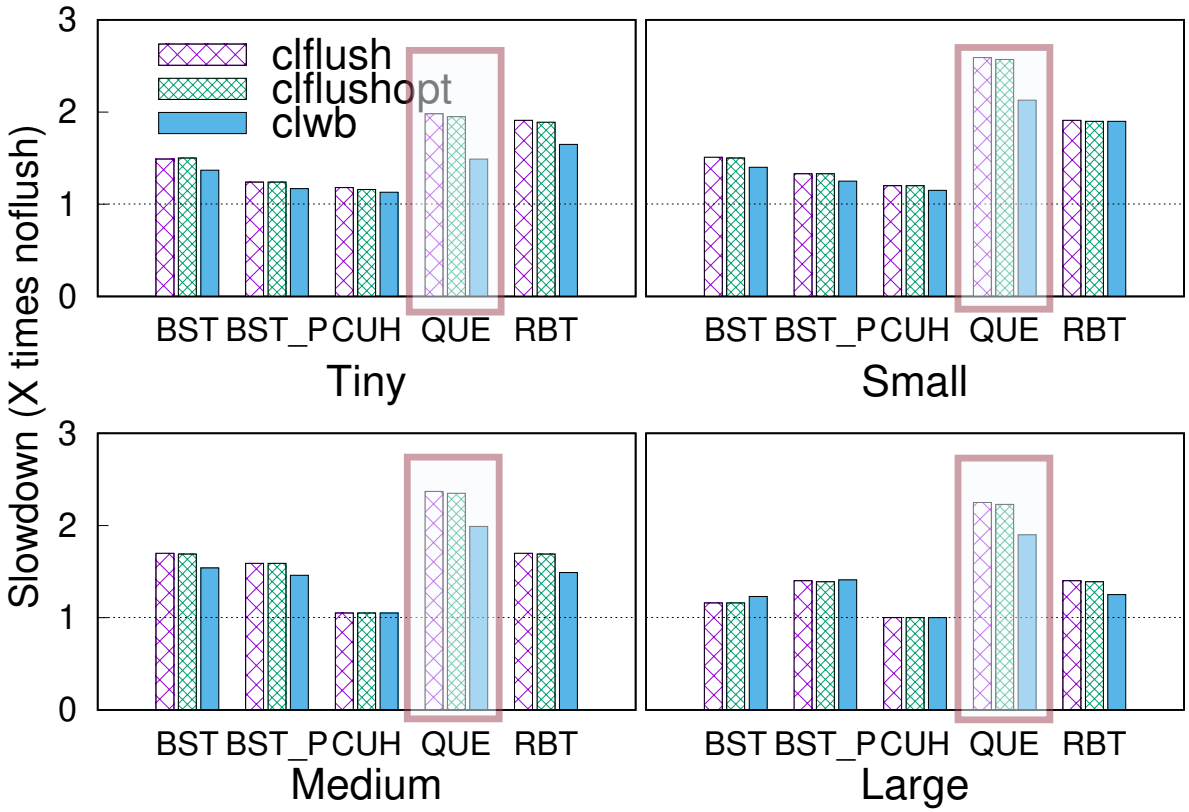
Performance overheads vary between 1X - 2.5X

Performance with X86-64



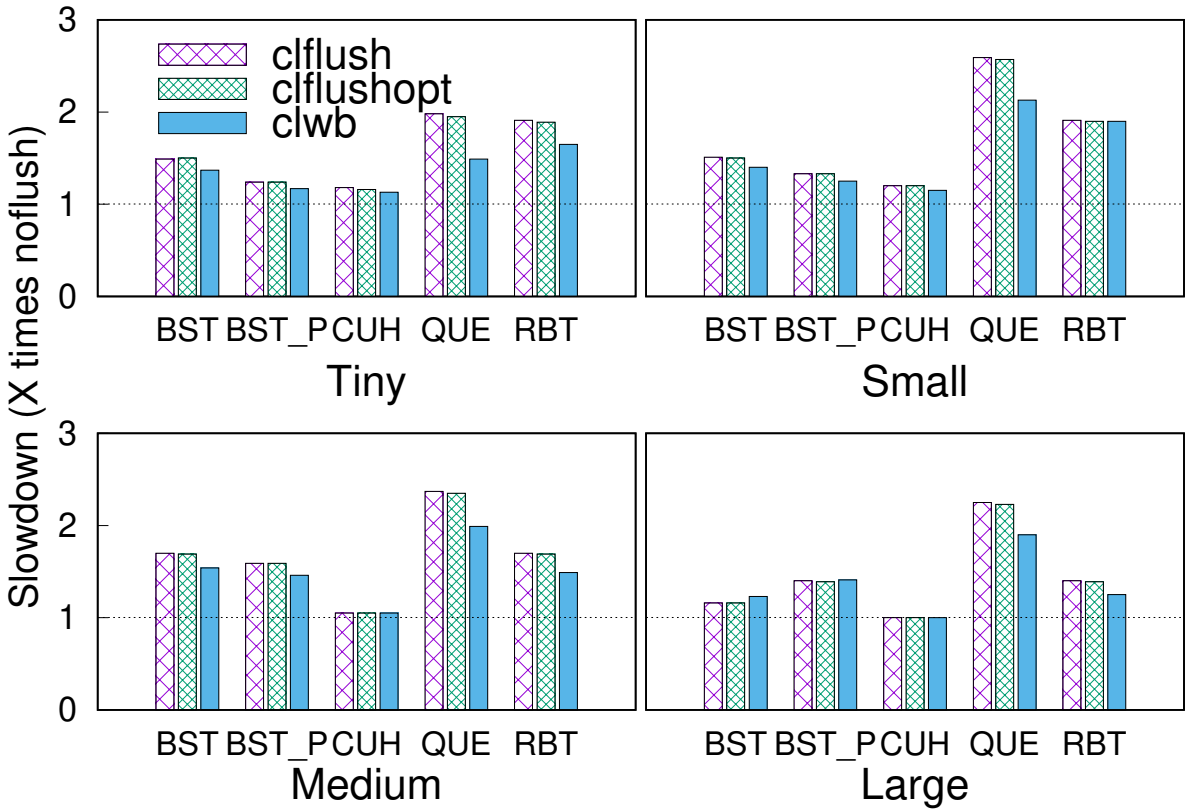
clwb performed better by 1X - 1.3X

Performance with X86-64



QUE resulted in highest performance overhead

Performance with X86-64



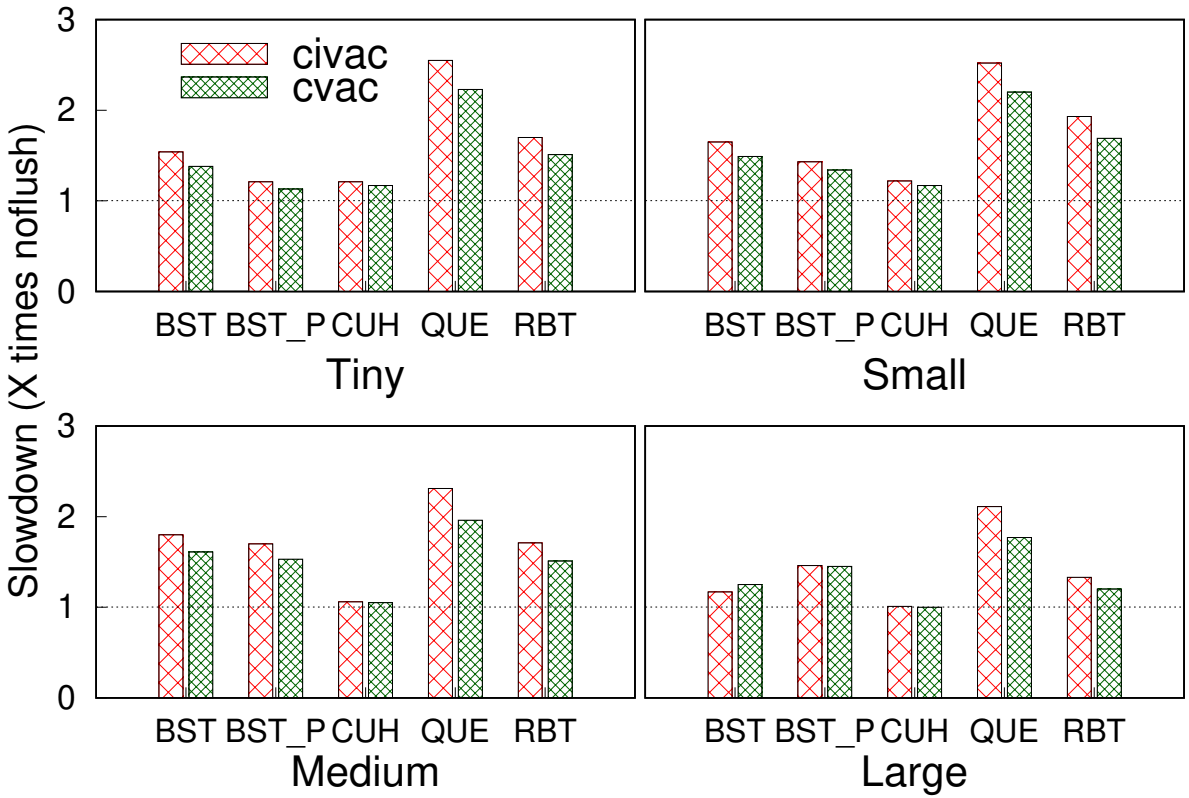
Write ordering contribute significantly to the slowdown

Performance with X86-64

#commit stalls at ROB			
noflush	6	34	6
clflush	798,423	3,348,775	536,286
clflushopt	532,284	2,232,528	270,146
clwb	532,284	2,232,528	270,146

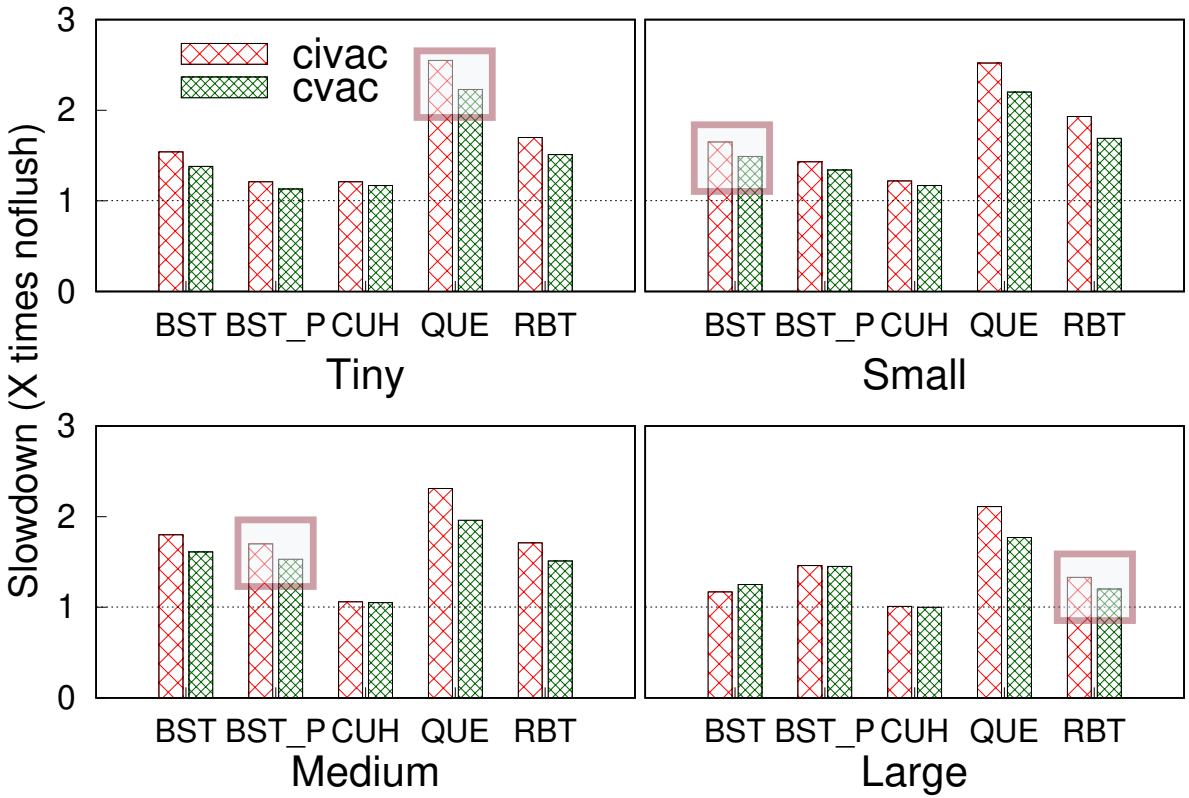
The highest number of commit stalls at ROB head is for clflush.

Performance with ARM64



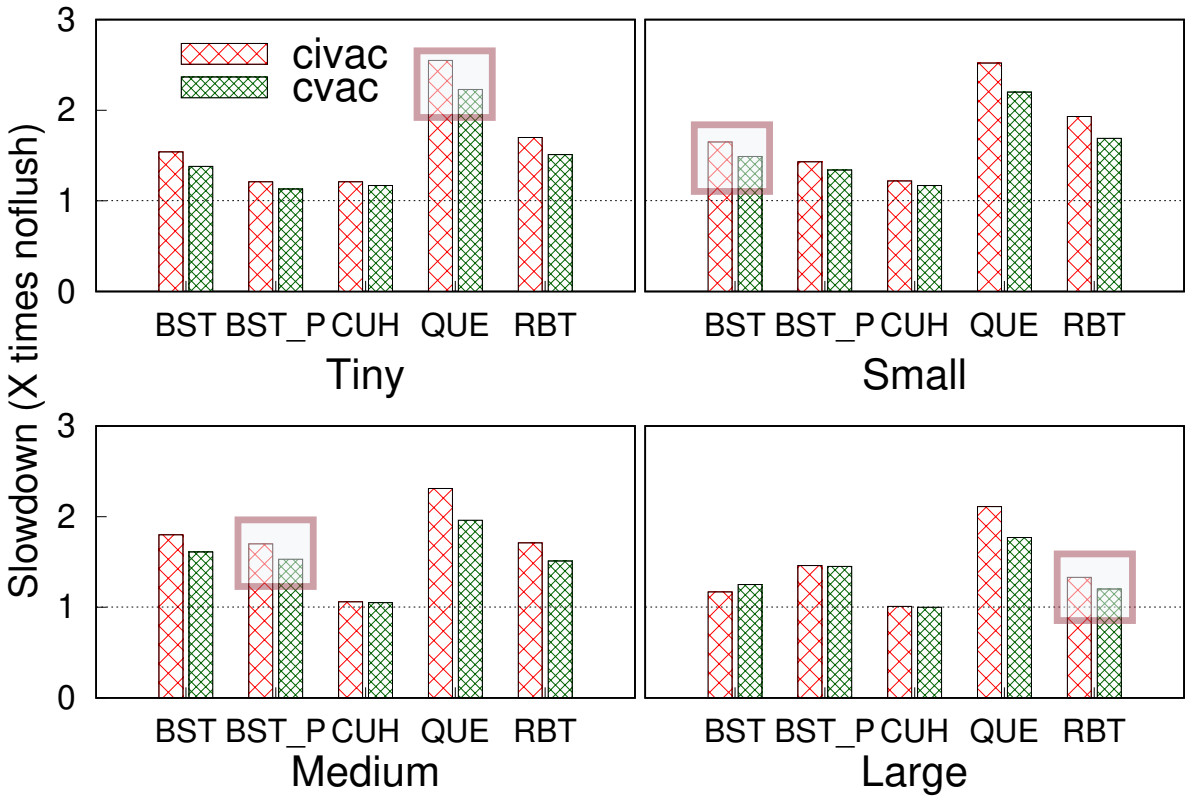
Performance trends are similar to X86−64

Performance with ARM64



cvac performed better than civac

Performance with ARM64



cvac is functionally equivalent to clwb

Memory Footprint, Read-to-Write Ratio Influence on Performance.

Influence of Read-to-Write Ratio

- Memory access pattern $\Rightarrow$ Read % influence on performance.

Performance slowdown

Benchmark	Read-light	Read-balanced	Read-heavy
civac			
BST	1.67	1.78	1.89
BST_P	1.54	1.64	1.97
QUE	2.47	2.34	2.22
RBT	1.66	1.71	1.82
cvac			
BST	1.53	1.63	1.83
BST_P	1.45	1.48	1.88
QUE	2.08	1.98	1.88
RBT	1.48	1.51	1.61

For medium working set size

Read-light	read:write ratio as 10:90	Read-balanced	read:write ratio as 50:50	Read-heavy	read:write ratio as 90:10

Influence of Read-to-Write Ratio

- Memory access pattern $\Rightarrow$ Read % influence on performance.
- RBT read % $\uparrow$ $\Rightarrow$ performance $\downarrow$ same with BST as well.

Performance slowdown

Benchmark	Read-light	Read-balanced	Read-heavy
civac			
BST	1.67	1.78	1.89
BST_P	1.54	1.64	1.97
QUE	2.47	2.34	2.22
RBT	1.66	1.71	1.82
cvac			
BST	1.53	1.63	1.83
BST_P	1.45	1.48	1.88
QUE	2.08	1.98	1.88
RBT	1.48	1.51	1.61

For medium working set size

Read-light	read:write ratio as 10:90	Read-balanced	read:write ratio as 50:50	Read-heavy	read:write ratio as 90:10

Influence of Read-to-Write Ratio

- Memory access pattern $\Rightarrow$ Read % influence on performance.
- RBT read % $\uparrow$ $\Rightarrow$ performance $\downarrow$ same with BST as well.
- QUE read % $\uparrow$ $\Rightarrow$ performance $\uparrow$.

Performance slowdown

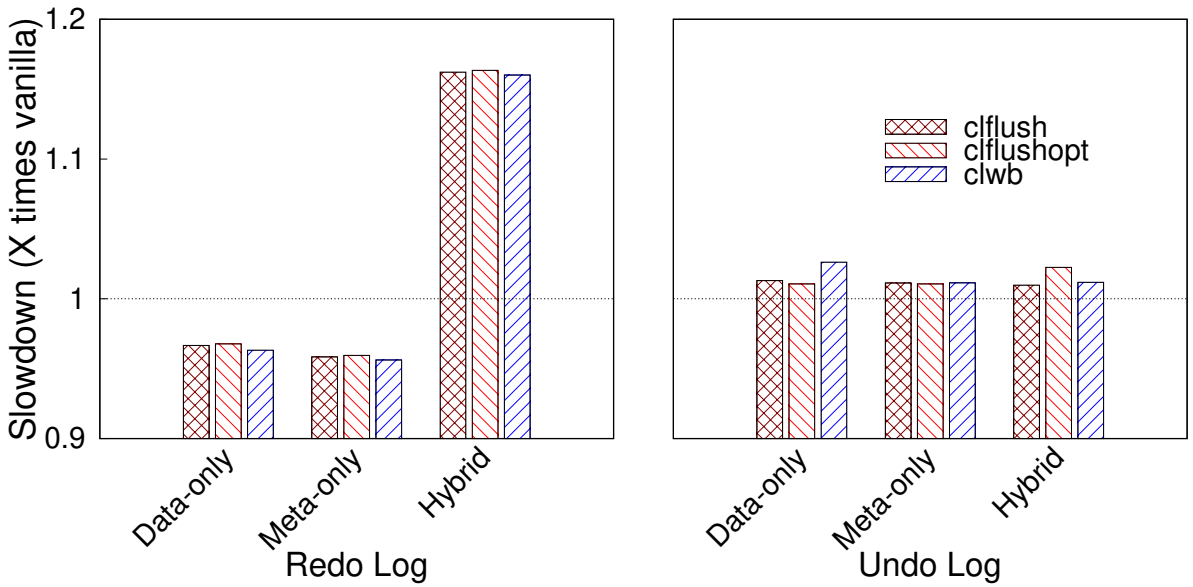
Benchmark	Read-light	Read-balanced	Read-heavy
civac			
BST	1.67	1.78	1.89
BST_P	1.54	1.64	1.97
QUE	2.47	2.34	2.22
RBT	1.66	1.71	1.82
cvac			
BST	1.53	1.63	1.83
BST_P	1.45	1.48	1.88
QUE	2.08	1.98	1.88
RBT	1.48	1.51	1.61

For medium working set size

Read-light	read:write ratio as 10:90	Read-balanced	read:write ratio as 50:50	Read-heavy	read:write ratio as 90:10

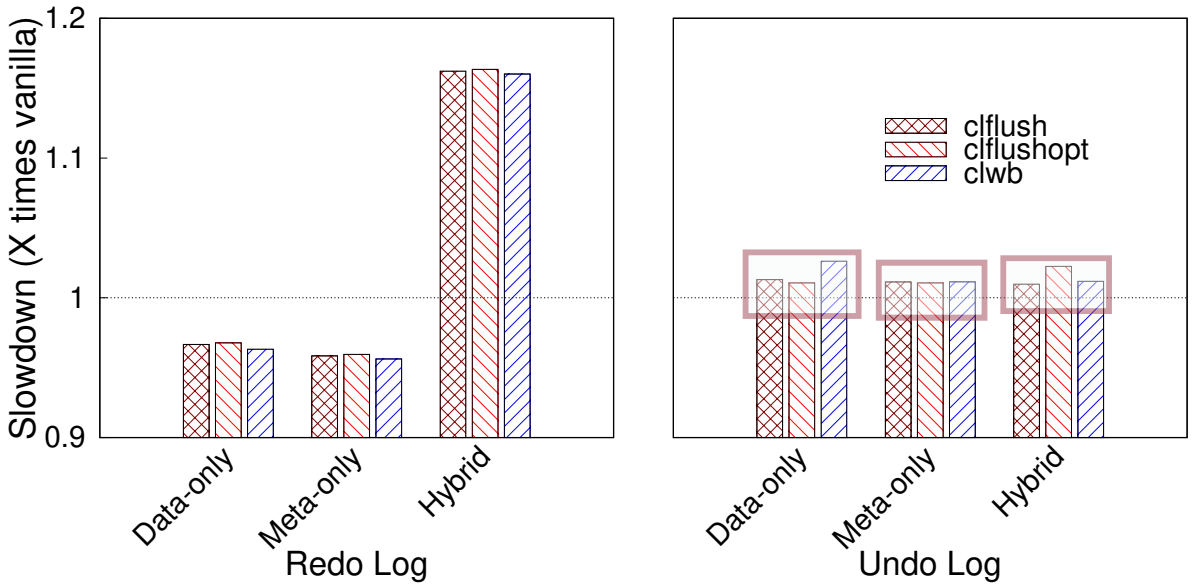
Redo, Undo Logging Performance for X86-64 and ARM64

Heavy Logging Performance (X86-64)



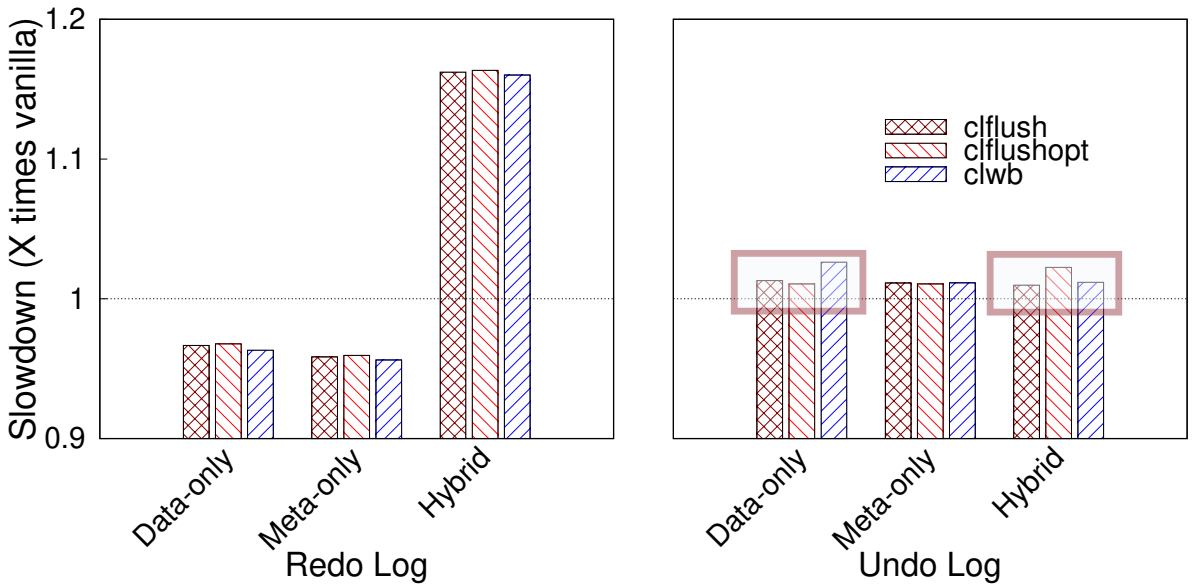
Log Heavy	No Logging:Logging ratio 10:90	Log Medium	No Logging:Logging ratio 50:50	Log Light	No Logging:Logging ratio 90:10

Heavy Logging Performance (X86-64)



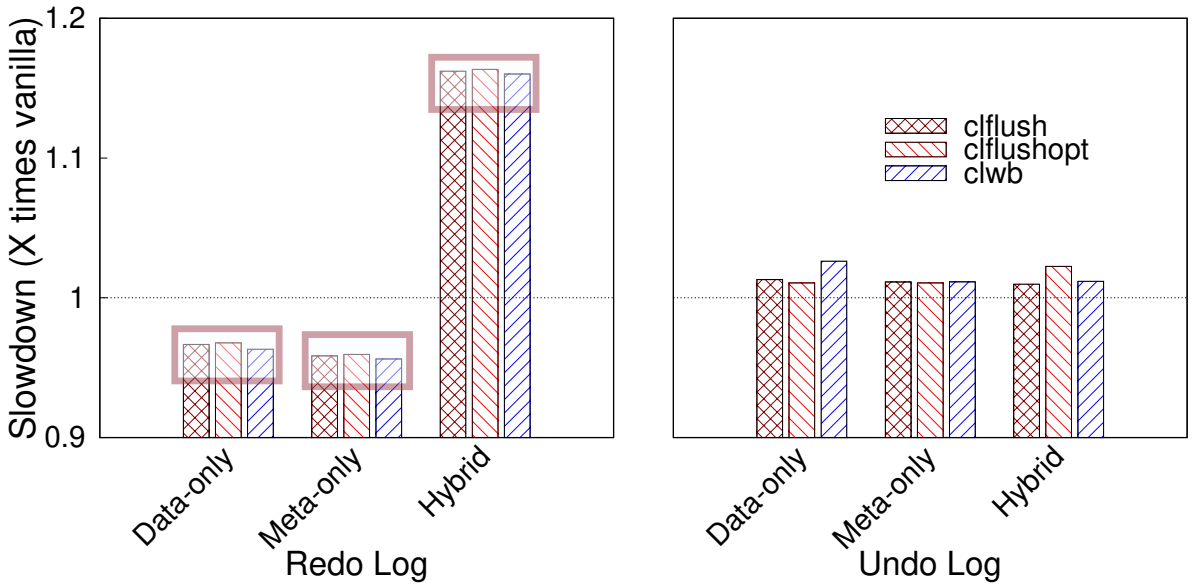
Undo log showed performance overhead for all modifications types.

Heavy Logging Performance (X86-64)



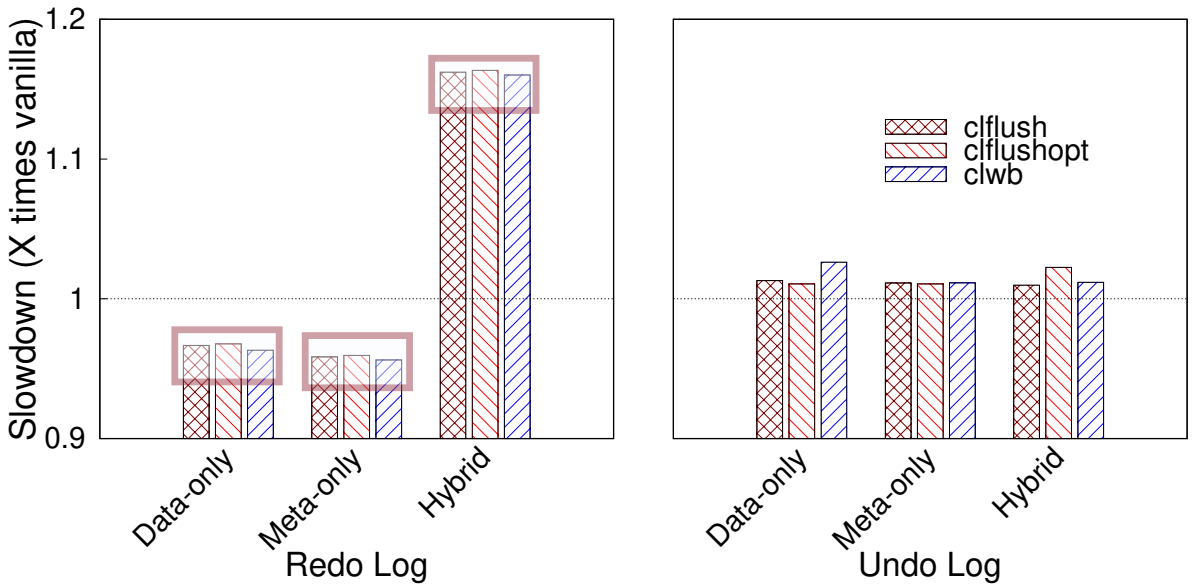
clflush, clflushopt showed better performance than clwb with undo.

Heavy Logging Performance (X86-64)



Redo showed benefit with clwb.

Heavy Logging Performance (X86-64)



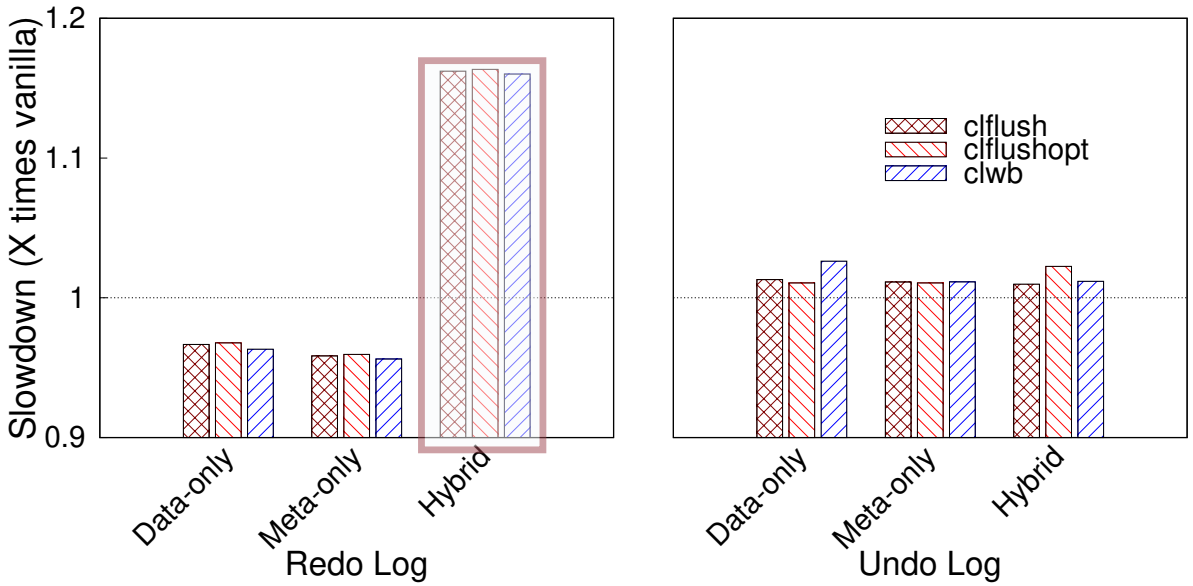
Redo+clwb provided cache benefits.

Heavy Logging Performance (X86-64)

Methods	Data-only	Meta-only	Hybrid
Log Heavy			
vanilla	20.46	20.43	20.27
redo	18.44	18.52	17.76
undo	20.23	20.27	20.17

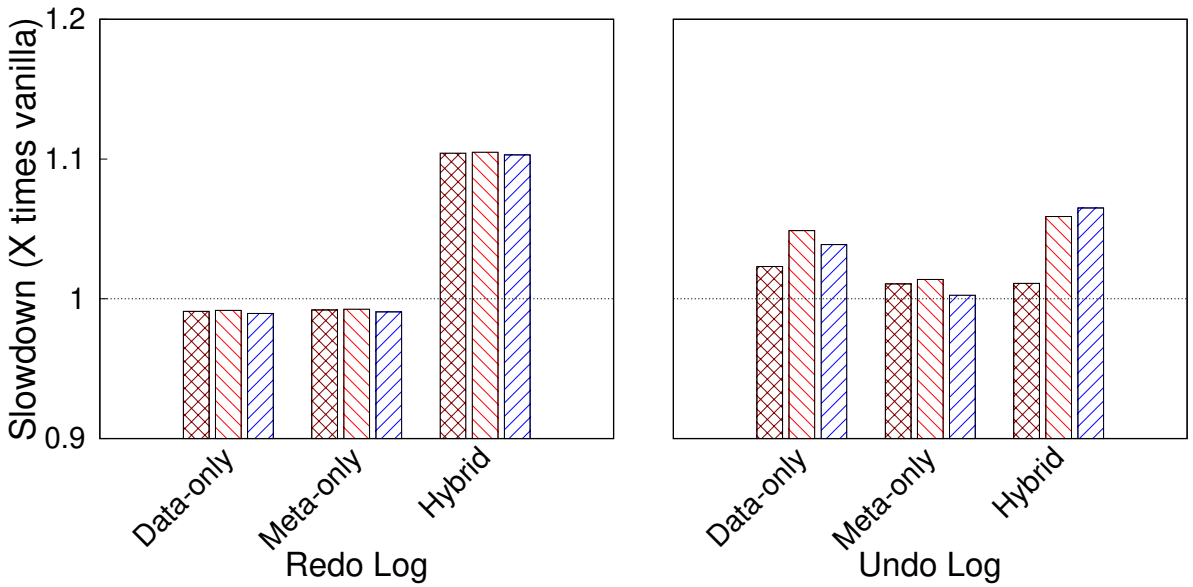
With redo+clwb L1-D miss rates are lower, and also L1-D cache write-backs are reduced between 31% to 5%.

Heavy Logging Performance (X86-64)



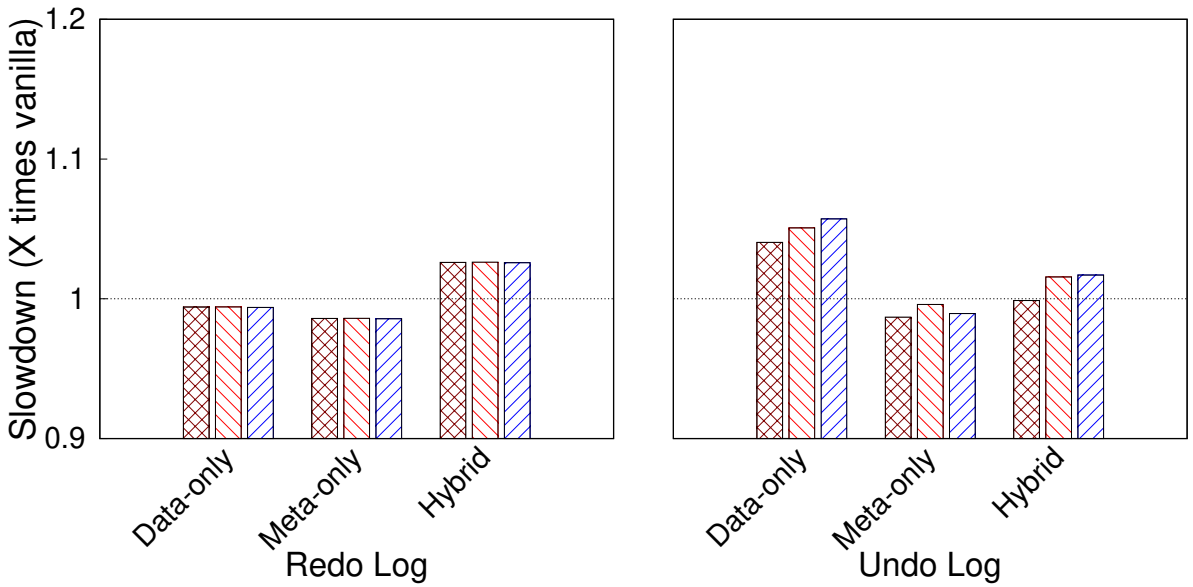
Cache prefetching plays an important role in redo performance, with redo 37% ↑ in issued L1D prefetch requests for Hybrid.

Medium Logging Performance (X86-64)



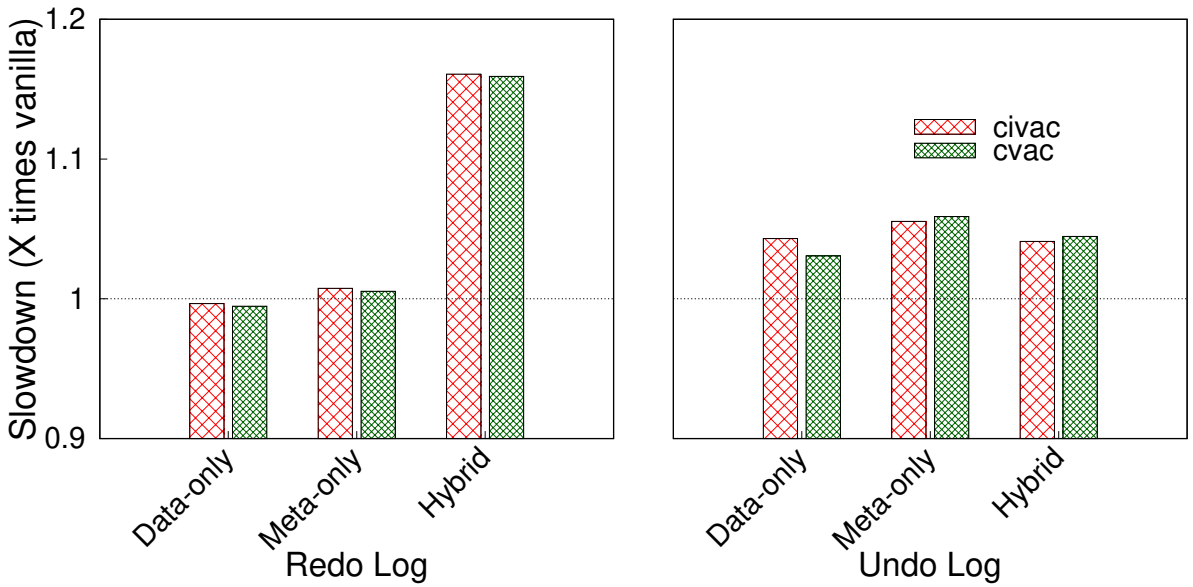
The performance trends are similar to heavy logging.

Light Logging Performance (X86-64)



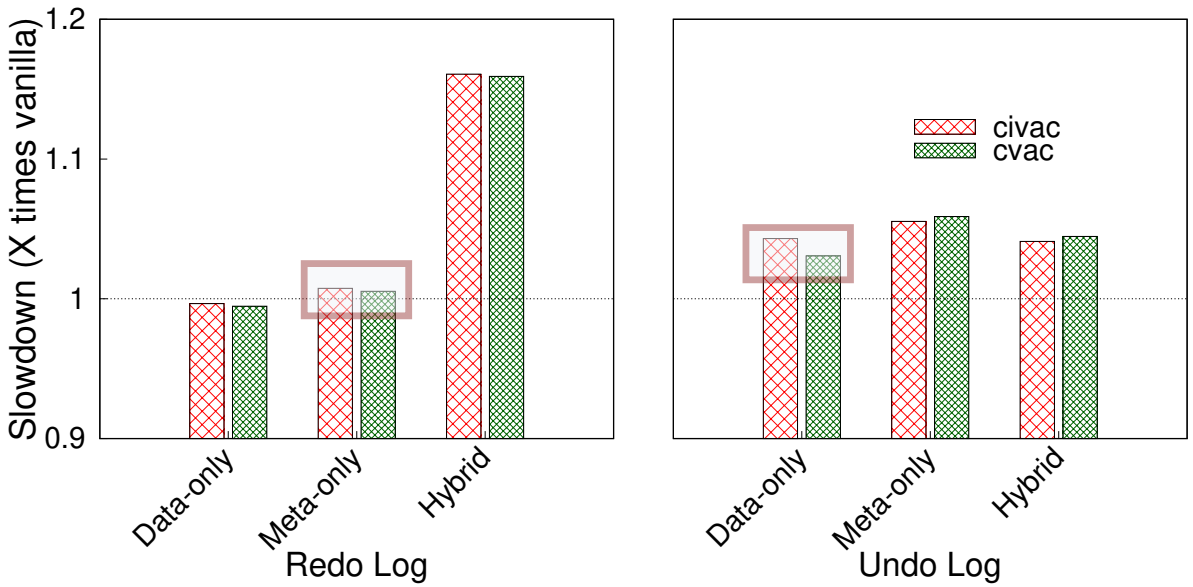
The clflush, clflushopt continued showing better performance than clwb with undo.

Heavy Logging Performance (ARM64)



Performance trends are similar to X86−64

Heavy Logging Performance (ARM64)



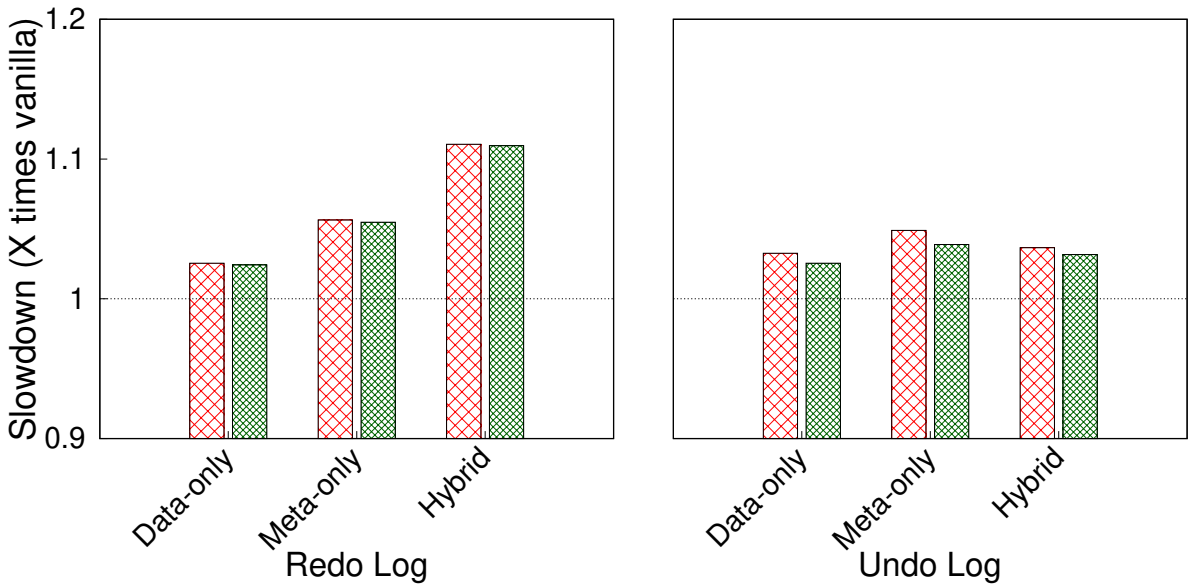
cvac performed better with redo, undo in majority of cases.

Heavy Logging Performance (ARM64)

Methods	Data-only	Meta-only	Hybrid
Log Heavy			
vanilla	21.41	21.47	21.39
redo	19.15	18.99	18.43
undo	21.16	21.22	21.12

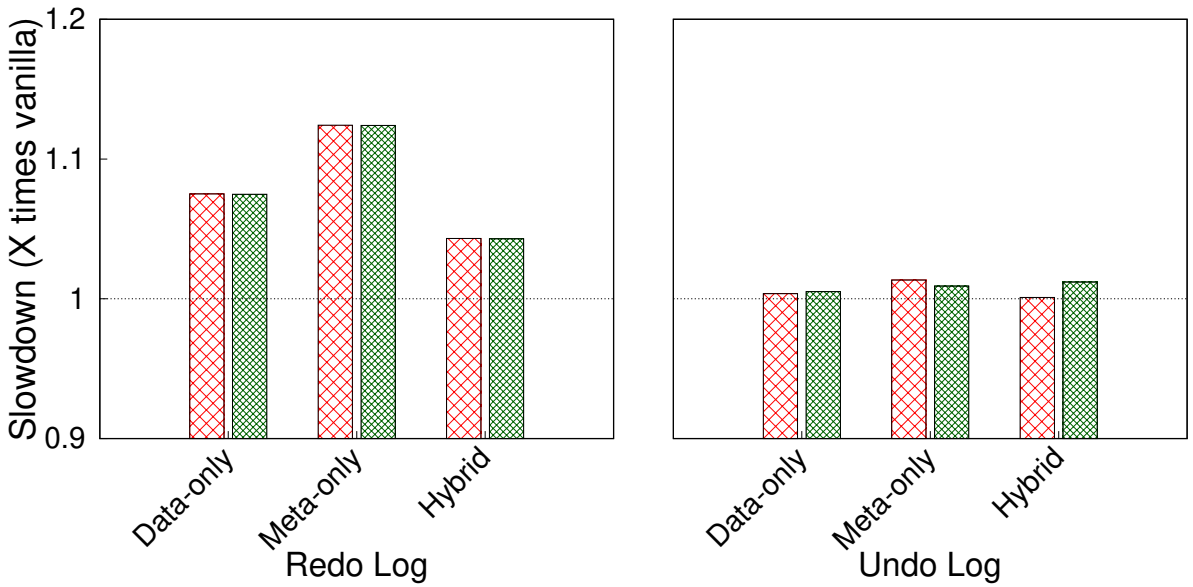
With redo+cvac L1-D miss rates are lower.

Medium Logging Performance (ARM64)



The performance trends are similar to heavy logging.

Light Logging Performance (ARM64)



cvac continued performing better with redo, undo in majority of cases.

Key Takeaways

- `X86` and `ARM64` showed similar data consistency performance overhead trends.

Key Takeaways

- `X86` and `ARM64` showed similar data consistency performance overhead trends.
- clwb, cvac benefited applications with temporal and spatial locality.

Key Takeaways

- `X86` and `ARM64` showed similar data consistency performance overhead trends.
- clwb, cvac benefited applications with temporal and spatial locality.
- Fences contributed significantly to the persistent barrier overhead.

Key Takeaways

- `X86` and `ARM64` showed similar data consistency performance overhead trends.
- clwb, cvac benefited applications with temporal and spatial locality.
- Fences contributed significantly to the persistent barrier overhead.
- Application's memory access pattern decide influence of read % on persistent barrier overhead.

Key Takeaways

- `X86` and `ARM64` showed similar data consistency performance overhead trends.
- clwb, cvac benefited applications with temporal and spatial locality.
- Fences contributed significantly to the persistent barrier overhead.
- Application's memory access pattern decide influence of read % on persistent barrier overhead.
- Nature of logging requirement decide choice of cache-line flush.

Conclusion

We studied the performance overhead of data consistency methods on `X86-64` and `ARM64`. We found that —

- Selection of data consistency methods should be based upon the workload characteristics.
- Serialization operations to enforce order of writes to NVM impact performance.

As a future direction, study the influence of data consistency methods on co-running applications in a memory and cache congested scenario.

For more details contact:

Arun KP
kparun@cse.iitk.ac.in