Lazy Persistency: a High-Performing and Write-Efficient Software Persistency Technique

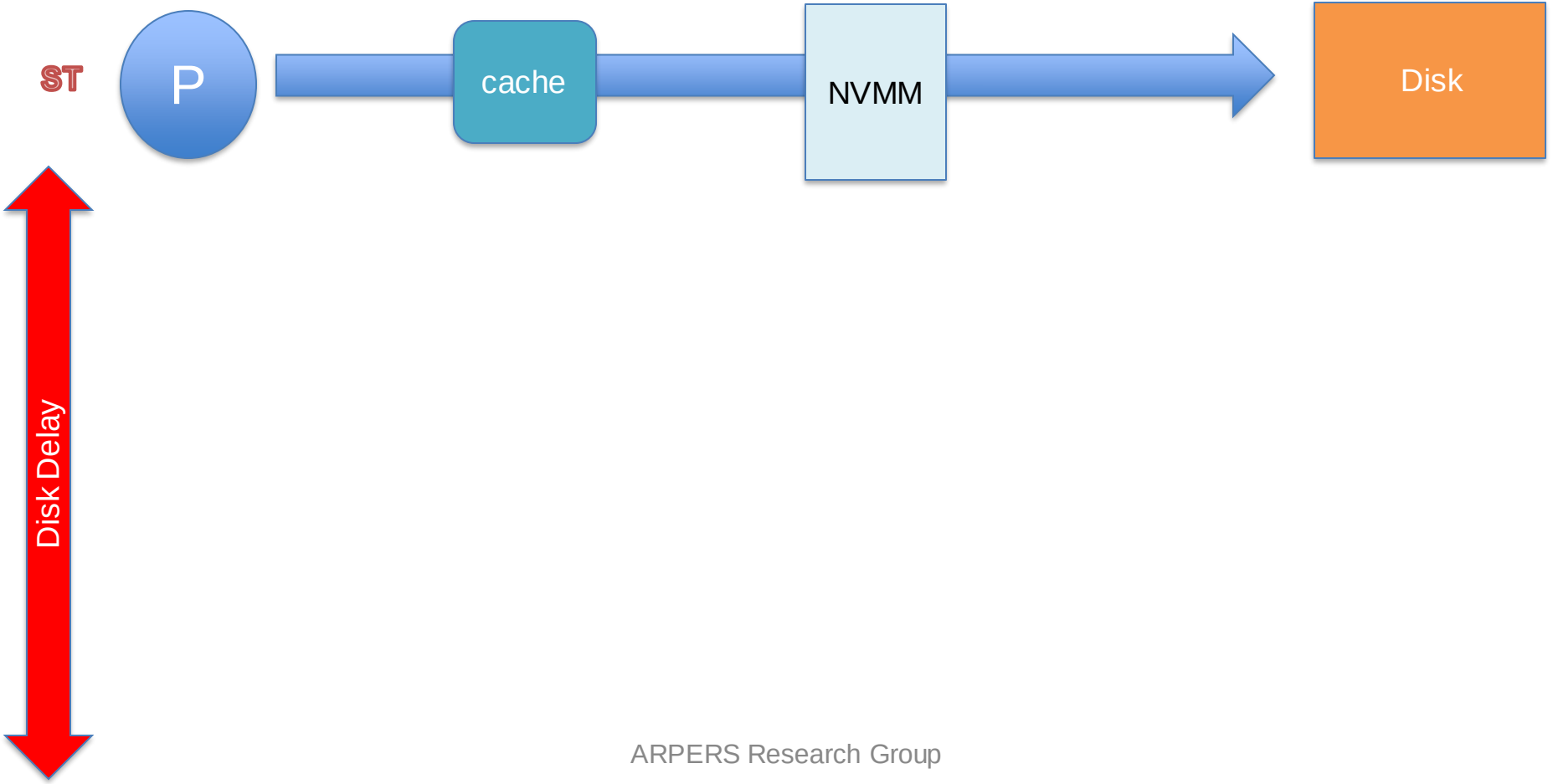
Mohammad Alshboul, James Tuck, and Yan Solihin

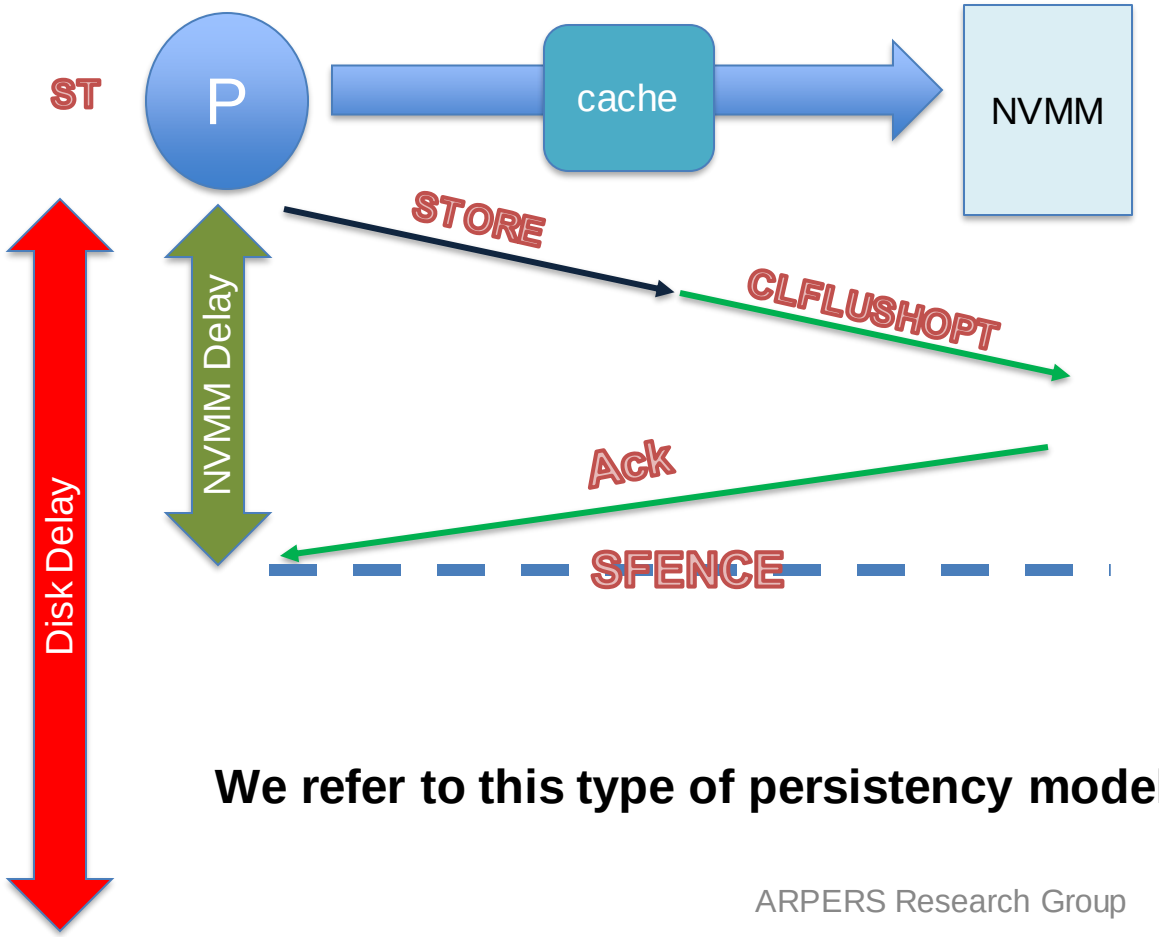
Email: *maalshbo@ncsu.edu*

ARPERS Research Group

Introduction

- Systems will include Non-Volatile Main Memory (NVMM)
  - Faster than Disk, and Byte-addressable
  - Denser than DRAM, and Non-volatile
- NVMM can host data persistently across crashes and reboots
- Data structure to be written in a crash consistent way
- Persistency models define when stores reach NVMM (i.e. become durable)
  - E.g. Intel PMEM: CLFLUSH, CLFLUSHOPT, CLWB, SFENCE


ST
P
cache
NVMM
Disk
Disk Delay



- CLFLUSHOPT flushes a cache block to NVMM
- SFENCE orders CLFLUSHOPT with other stores

We refer to this type of persistency models as Eager Persistency

Our Solution: Lazy Persistency

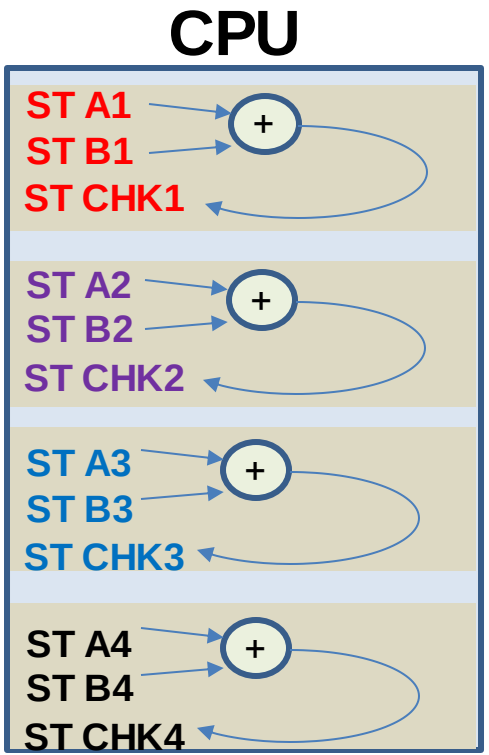
- Software technique
- Principle: Make the Common Case Fast
- Code is broken into Lazy Persistency (LP) regions
  - Each LP region protected by a checksum
  - Checksum enables persistency failure detection after a crash
  - On recovery, failed regions are re-executed

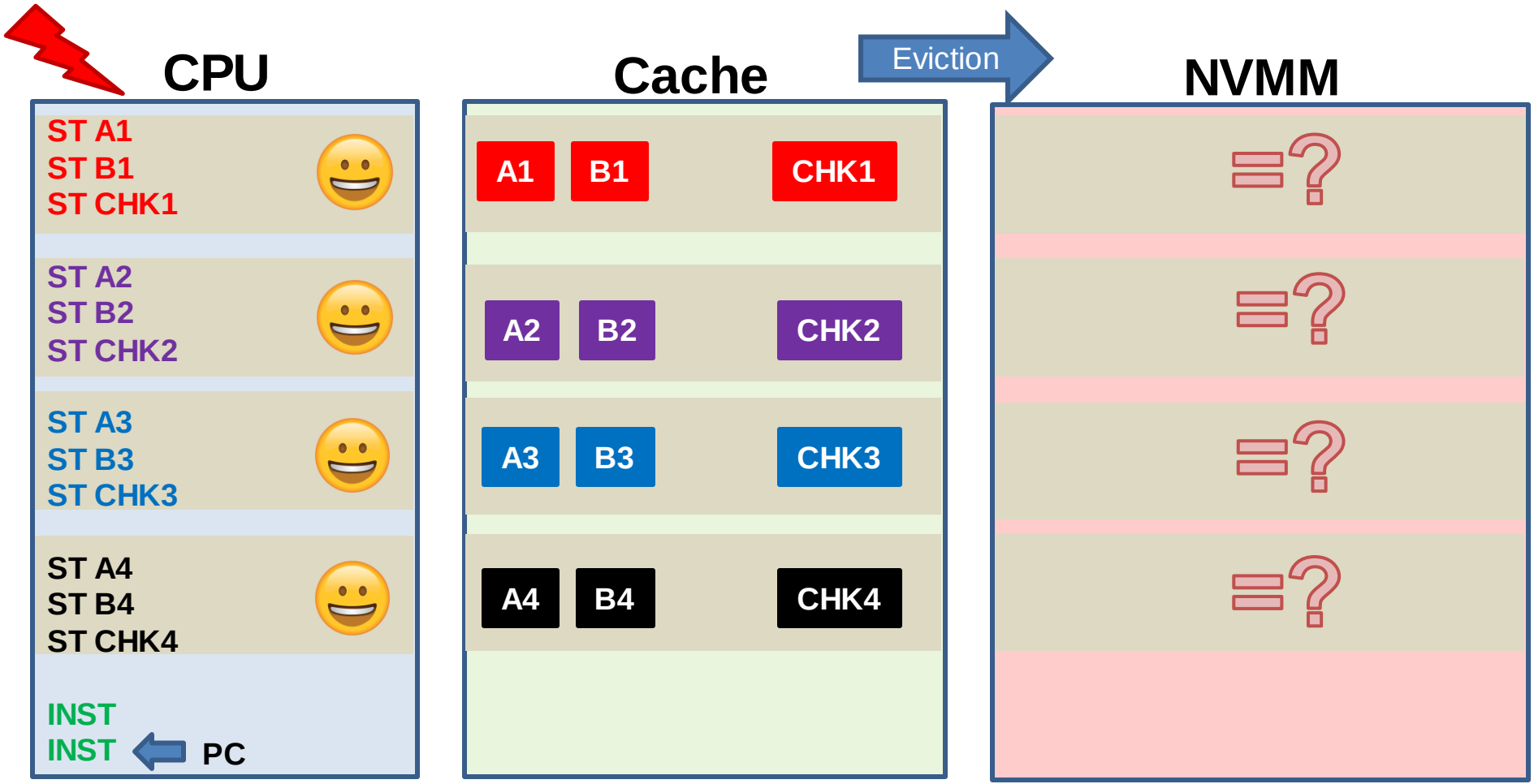
- **Lazily relies on natural cache evictions**

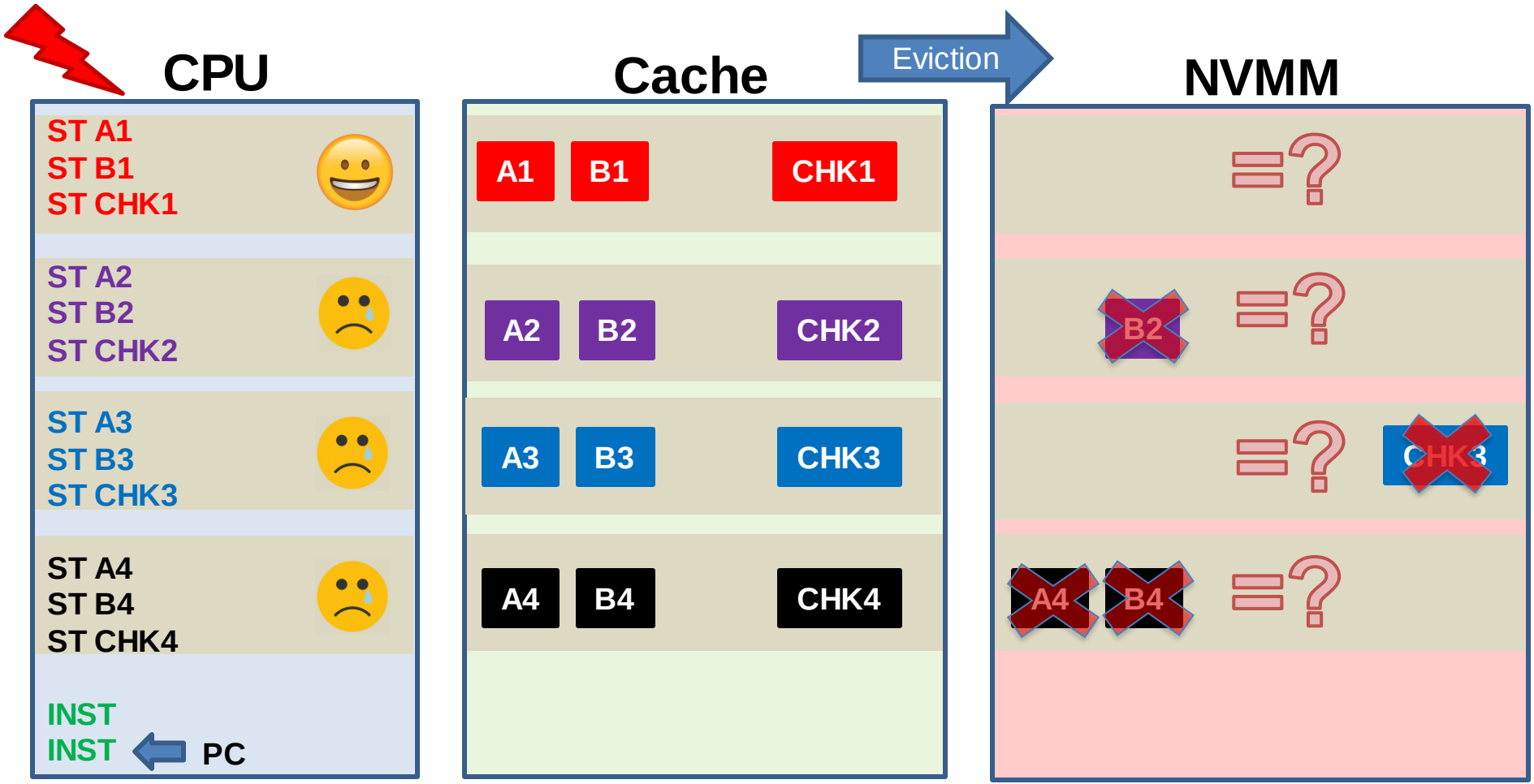
  No persist barriers (CLFLUSHOPT, SFENCE) needed

Lazy Persistency Details

- Programs are divided into associative LP regions
- Programmers choose LP region granularity
- A checksum covers updates in an LP region
  - Stored at the end of the LP region

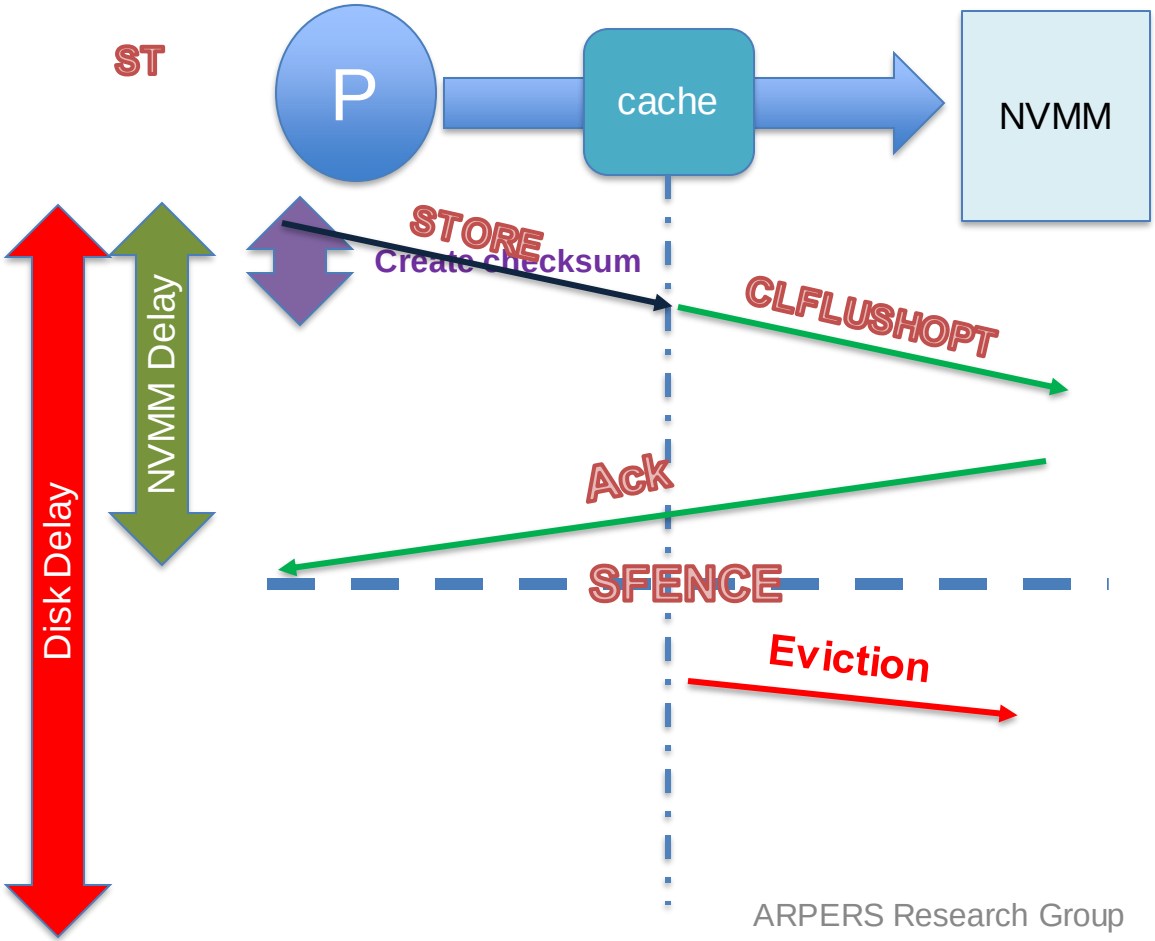



CPU
Cache
Eviction
NVMM
ST A1
ST B1
ST CHK1
ST A2
ST B2
ST CHK2
ST A3
ST B3
ST CHK3
ST A4
ST B4
ST CHK4
INST
INST
PC
A1
B1
CHK1
A2
B2
CHK2
A3
B3
CHK3
A4
B4
CHK4
=?
=?
=?
=?


CPU
Cache
Eviction
NVMM
ST A1
ST B1
ST CHK1
ST A2
ST B2
ST CHK2
ST A3
ST B3
ST CHK3
ST A4
ST B4
ST CHK4
INST
INST
PC
A1
B1
CHK1
A2
B2
CHK2
A3
B3
CHK3
A4
B4
CHK4
B2
A4
B4

Recovering From a Crash

- On a crash, checksums are validated to detect regions that were not persisted
- Failed regions are recomputed
- Finally, program resumes execution in normal mode


ST
P
cache
NVMM
Disk
STORE
Create checksum
CLFLUSHOPT
NVMM Delay
Disk Delay
Ack
SFENCE
Eviction

Limitations of Lazy Persistency

- LP regions need to be associative, i.e. (R1, R2), R3 = R1, (R2, R3)
  - Most HPC kernels contain loop iterations that satisfy this requirement
  - Can be relaxed in some situations (see the paper)

- Recovery code needed for LP regions
  - Solution: Recompute Scheme [TACO'19]

- Amount of recovery may be unbounded (e.g. due to hot blocks)
  - Solution: Periodic Flushes (Next Slide)

Bounding the Amount of Recovery

- Cache blocks may stay in the cache for a long time (e.g. hot blocks)
  - Getting worse the larger the cache
- Regions with such blocks may fail to persist
- Upper-bound is needed for the time a block might remain dirty in the cache
- This is needed to guarantee forward progress

Solution: Periodic Flushes

- A simple hardware support
- All dirty blocks in the cache are written back periodically, in the background
- Modest increase in the number of writes (more details soon)
- The periodic flush interval puts an upper bound for recovery work

Evaluation

Methodology

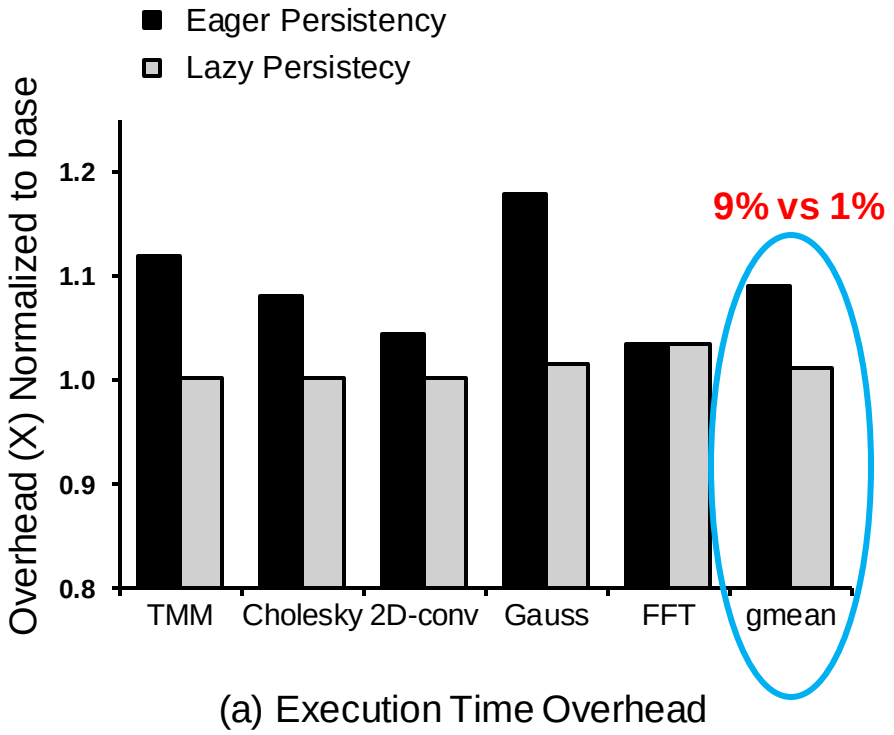
- Simulations on a modified version of gem5. Supports most Intel PMEM instructions (e.g. CLFLUSHOPT)
- Detailed out-of-order CPU model. Ruby memory system. 8 threads is the default for all experiments
- Evaluation was also done on 32-core DRAM-based real hardware machine

Evaluation

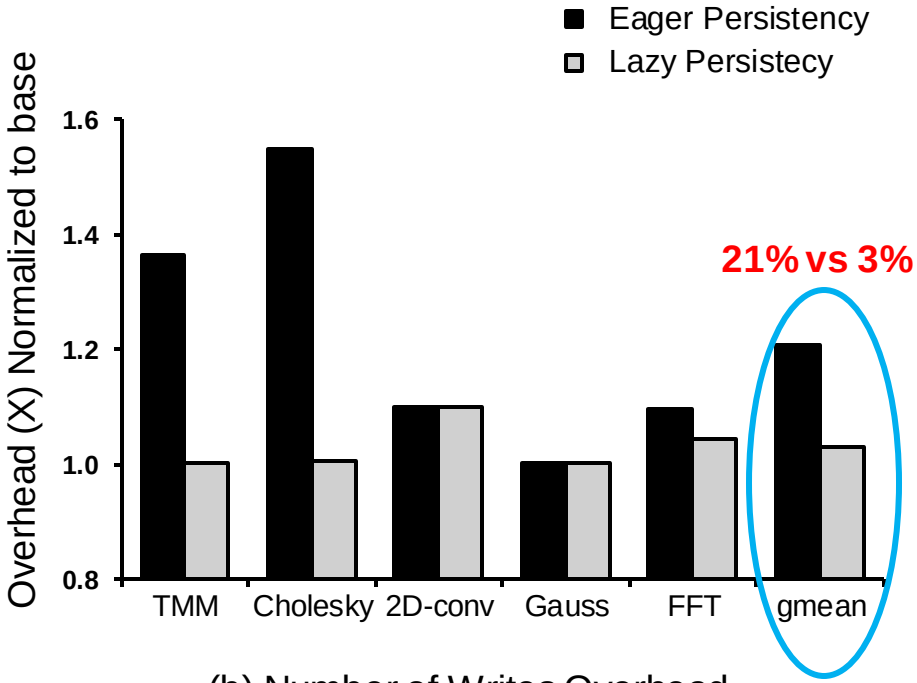
Multi-Threaded Benchmarks

- Tiled Matrix Multiplication
- Cholesky Factorization
- 2D convolution
- Fast Fourier Transform
- Gauss Elimination

Evaluation: All Benchmarks



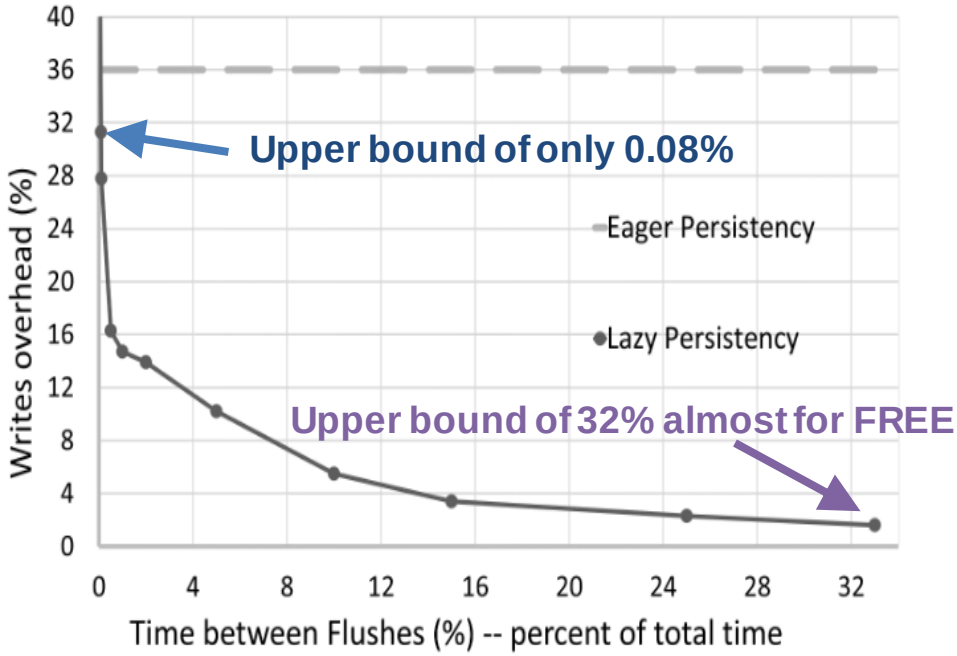
(a) Execution Time Overhead



(b) Number of Writes Overhead

Periodic Flushes (Cont.)

- The figure shows the number of writes overhead associated with each upper-bound
- These flushes are not necessary to be done at once
- Scheduling the writes allowing the process to happen in the background and have negligible impact on the execution time



More Evaluations

We performed other interesting evaluations that can be found in the paper:

- Sensitivity study with varying the read/write latency for NVMM
- Sensitivity study with varying the number of threads
- Evaluating the execution time for all the 5 benchmark on real hardware
- Sensitivity study with varying the Last Level Cache size
- Analysis for the Number of Writes of Periodic Flushes hardware support
- Evaluating the execution time overhead when trying different error detection mechanisms

Summary

- Lazy Persistency is a software persistency technique that relies on natural cache evictions (No stalls on SFENCE)
- It reduces the execution time and write amplification overheads, from 9% and 21%, to only 1% and 3%, respectively.
- A simple hardware support can provide an upper-bound on the recovery work

Questions?

Backup Slides

Methodology and Evaluation

Component	Configuration
Processor	2-17 cores (default 9), each OoO, 2GHz, 4-wide issue/retire ROB: 196, fetchQ/issueQ/LSQ: 48/48/48
L1I and L1D	64KB, 8-way, 64B block, 2 cycles
L2	512KB, 8-way, 64B block, 11 cycles
MC	ReadQ/WriteQ: 32/64, ADR
NVMM	Latencies: 60-150ns read (default 150ns), 150-300ns write (default 300ns)

Benchmark	Description
TMM	1k-square input matrix multiplication
Cholesky	1k-square input matrix cholesky factorization
2D-conv	1k-square input matrix 2D convolution
Gauss	4k-square input matrix gauss elimination
FFT	100k nodes vector FFT

Lazy Persistency Details

```
1  for (kk=starting_kk; kk<n; kk+=bsize) {
2      for ( ii = starting_ii ; ii <n; ii +=bsize) {
3          ResetCheckSum();
4          for ( jj =0; jj <n; jj +=bsize) {
5              for ( i=ii ; i<(ii+bsize); i++) {
6                  for (j=jj ; j<(jj+bsize); j++) {
7                      sum = c[i ][ j ];
8                      for (k=kk; k<(kk+bsize); k++)
9                          sum += a[i][k]*b[k ][ j ];
10                     c[ i ][ j ] = sum;
11                     UpdateCheckSum(c[i][j]);
12                 } // end of for j
13             } // end of for i
14         } // end of for jj
15         hashIndex = GetHashIndex(ii,kk);
16         HashTable[hashIndex] = GetCheckSum();
17     } // end of for ii
18 } // end of for kk
```

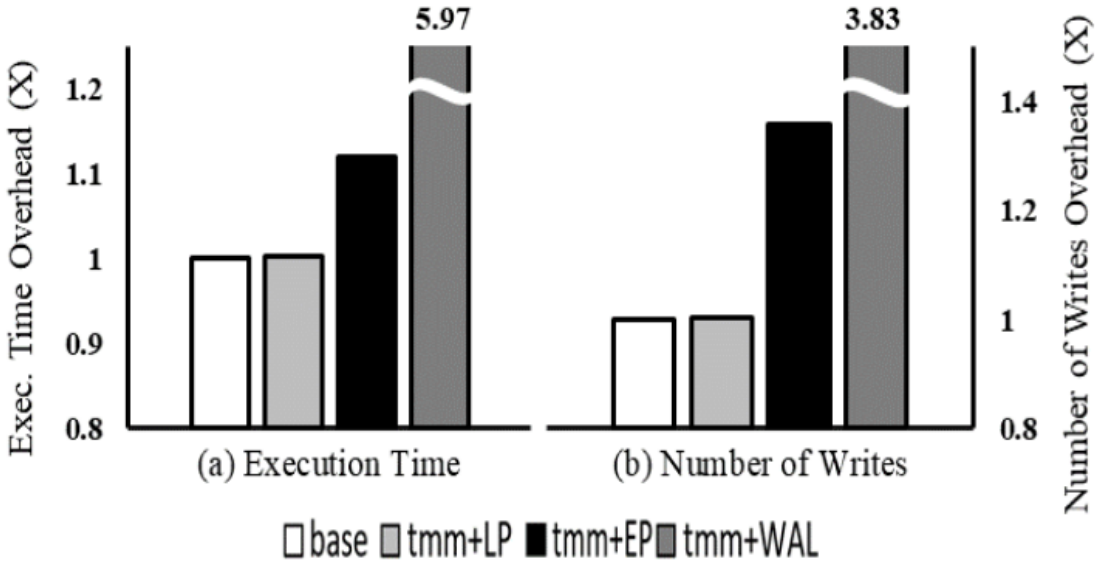
➔ Initialize at the beginning of the region

➔ Update during each iteration in the region

➔ Store the checksum to the corresponding location

LP Region

Evaluation: Main Figure



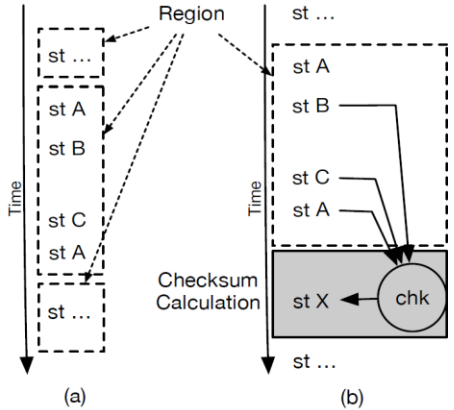
	base	tmm+LP	tmm+EP	tmm+WAL
Exec. Time (X)	1.00	1.002	1.12	5.97
# of writes (X)	1.00	1.003	1.36	3.83

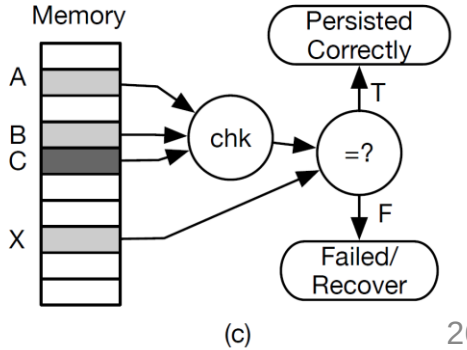
Introduction

- Making data become durable is very important, especially with the complexity of today's systems
- With systems with execution time much longer that MTTF, probably dealing with failures is very important
- Because DRAM is volatile, systems usually rely on the Disk to provide durability

Solution: Lazy Persistency

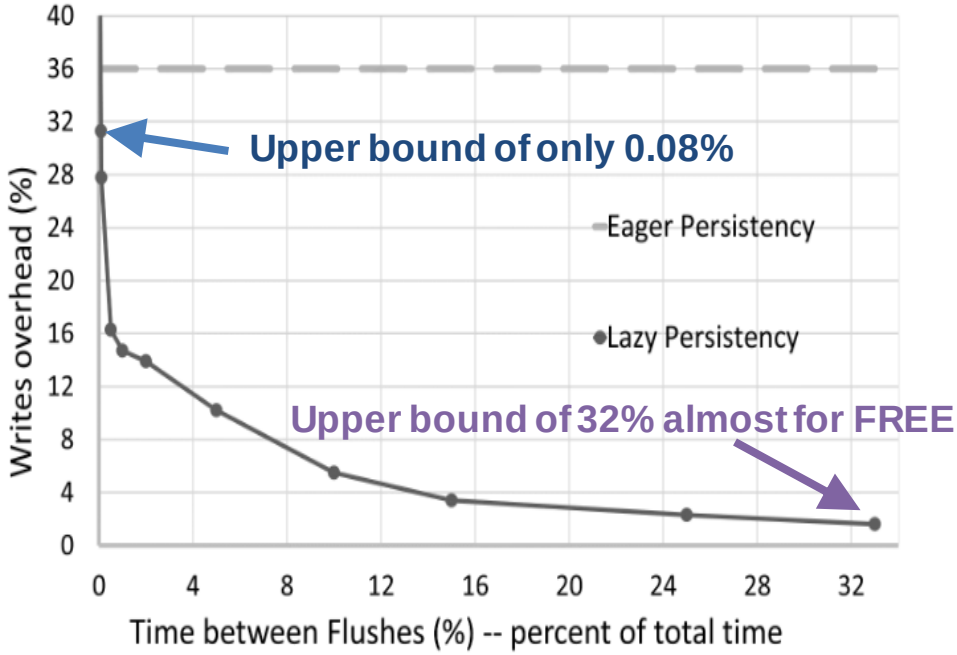
- Lazy Persistency is a software technique that *lazily* persist stores by relying on natural LLC evictions
- We add a **checksum** to each code region that can detect which regions did not fully persist
- Upon a crash, checksums are validated, and recovery is triggered upon the detection of persistency failure (*which is the rare case*)





Periodic Flushes (Cont.)

- The figure shows the number of writes overhead associated with each upper-bound
- These flushes are not necessary to be done at once
- Scheduling the writes allowing the process to happen in the background and have negligible impact on the execution time



Cases After Crash

- Only If the checksum and all the computed results are persisted, the region will be considered persisted and no fixing is needed
- If the checksum or at least one of the data computed results wasn’t persisted, the whole region is considered not persisted
- For any region that wasn’t fully persisted, it needs to be fixed and recomputed during the recovery phase.