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

Mohammad Alshboul, James Tuck, and Yan Solihin

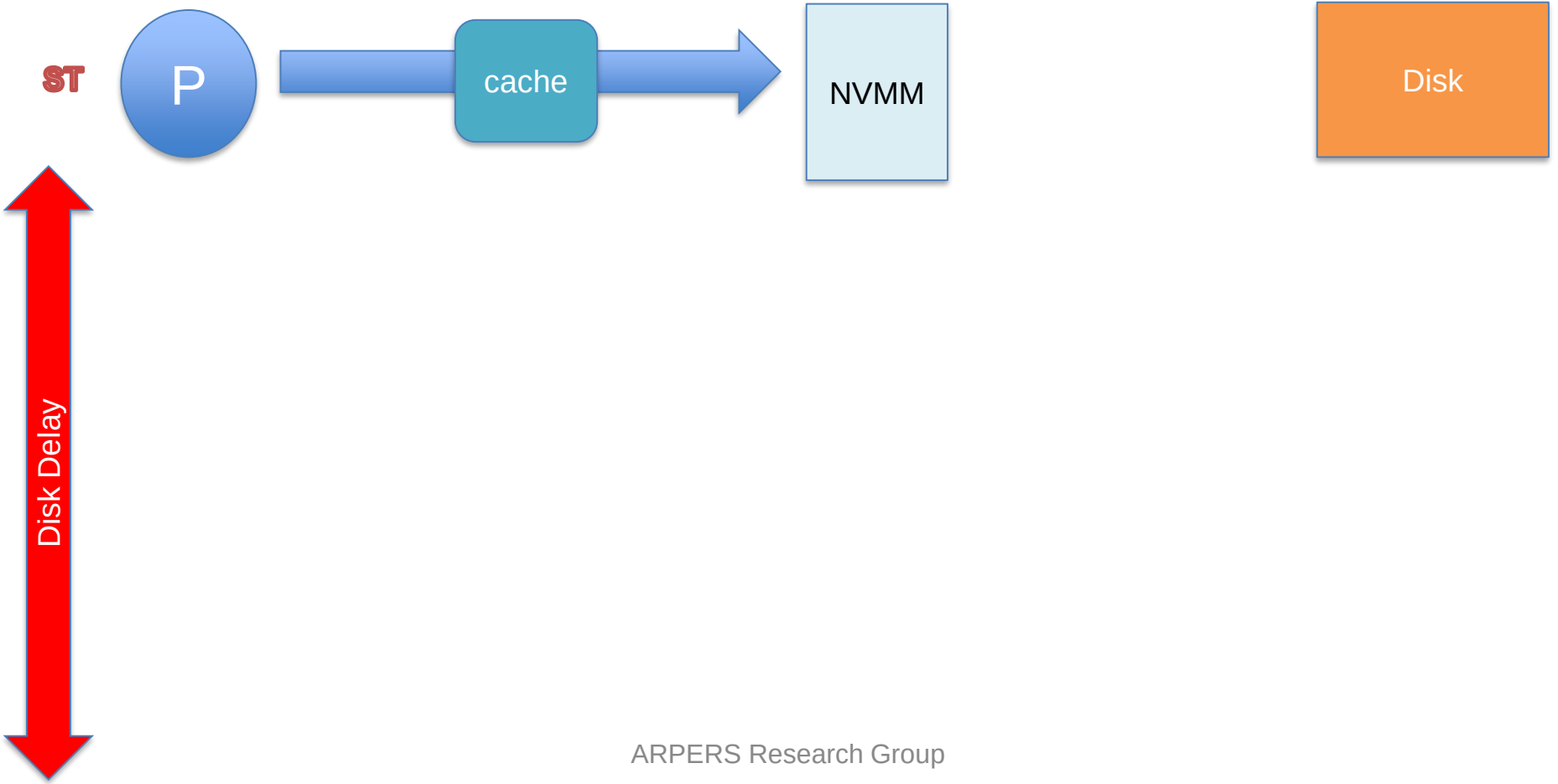
Email: *maalshbo@ncsu.edu*

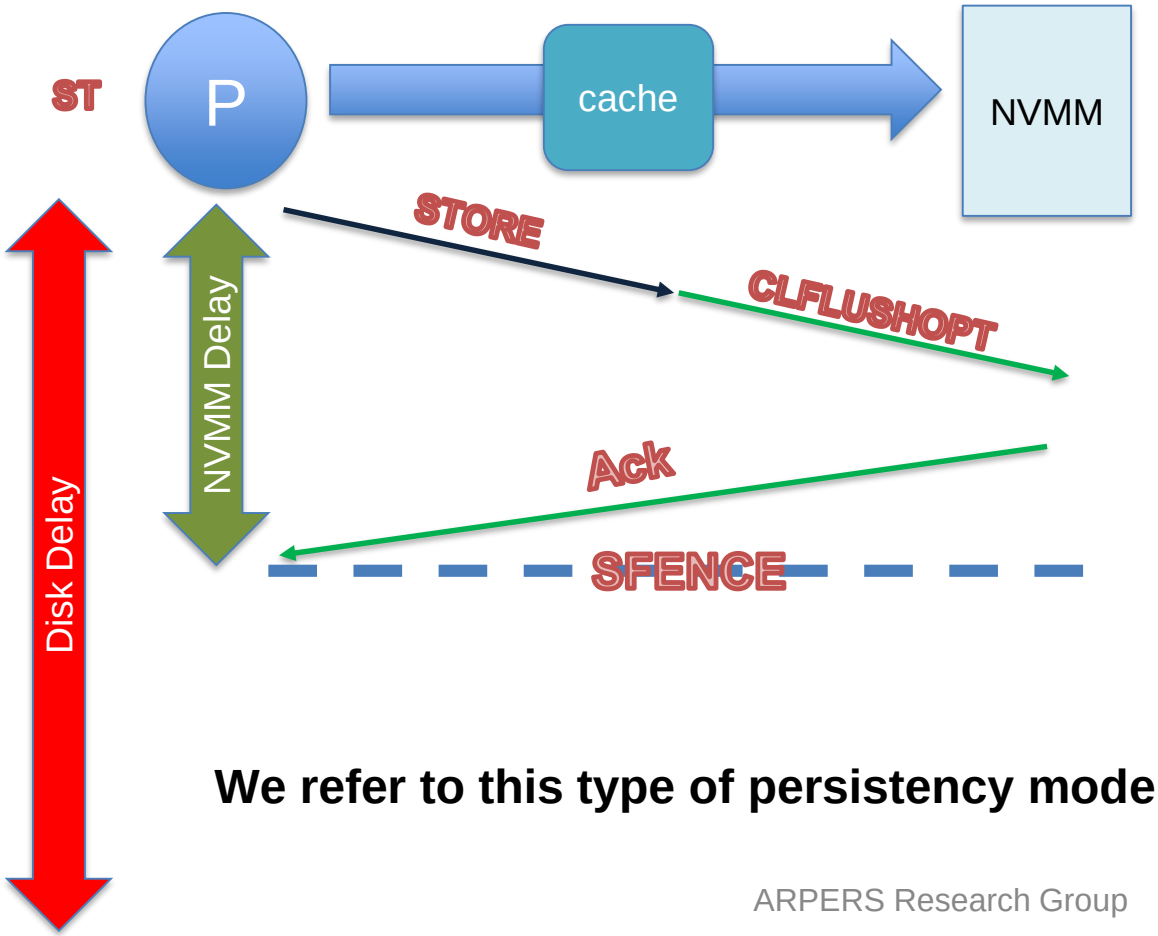
ARPERS Research Group

Introduction

- Future systems will likely include Non-Volatile Main Memory (NVMM)
- NVMM can host data persistently across crashes and reboots
- Crash consistent data requires persistency models, which define when stores reach NVMM (i.e. become durable)
  - E.g. Intel PMEM: CLFLUSH, CLFLUSHOPT, CLWB, SFENCE


ST
P
Disk
Disk Delay


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

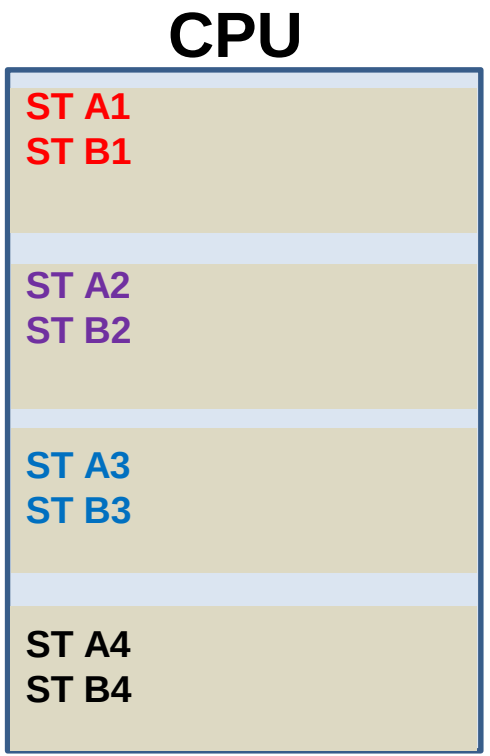
- Principle: Make the Common Case Fast
- Software technique
- Code is broken into Lazy Persistency (LP) regions
  - Each LP region protected by a checksum
  - Checksum enables persistency failure detection after a crash
  - On recovery, failed regions are re-executed

- **Lazily relies on natural cache evictions**

  No persist barriers (CLFLUSHOPT, SFENCE) needed

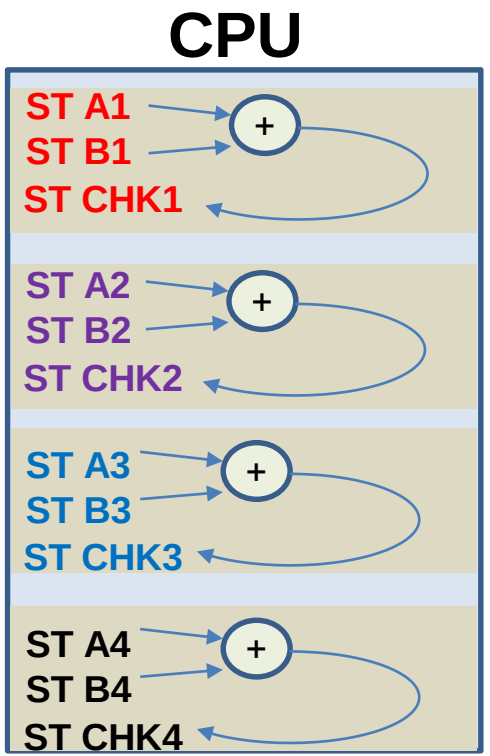
Lazy Persistency Details

- 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



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



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
```

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
```

LP Region

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
```

LP Region

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

LP Region

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

LP Region

Lazy Persistency Details

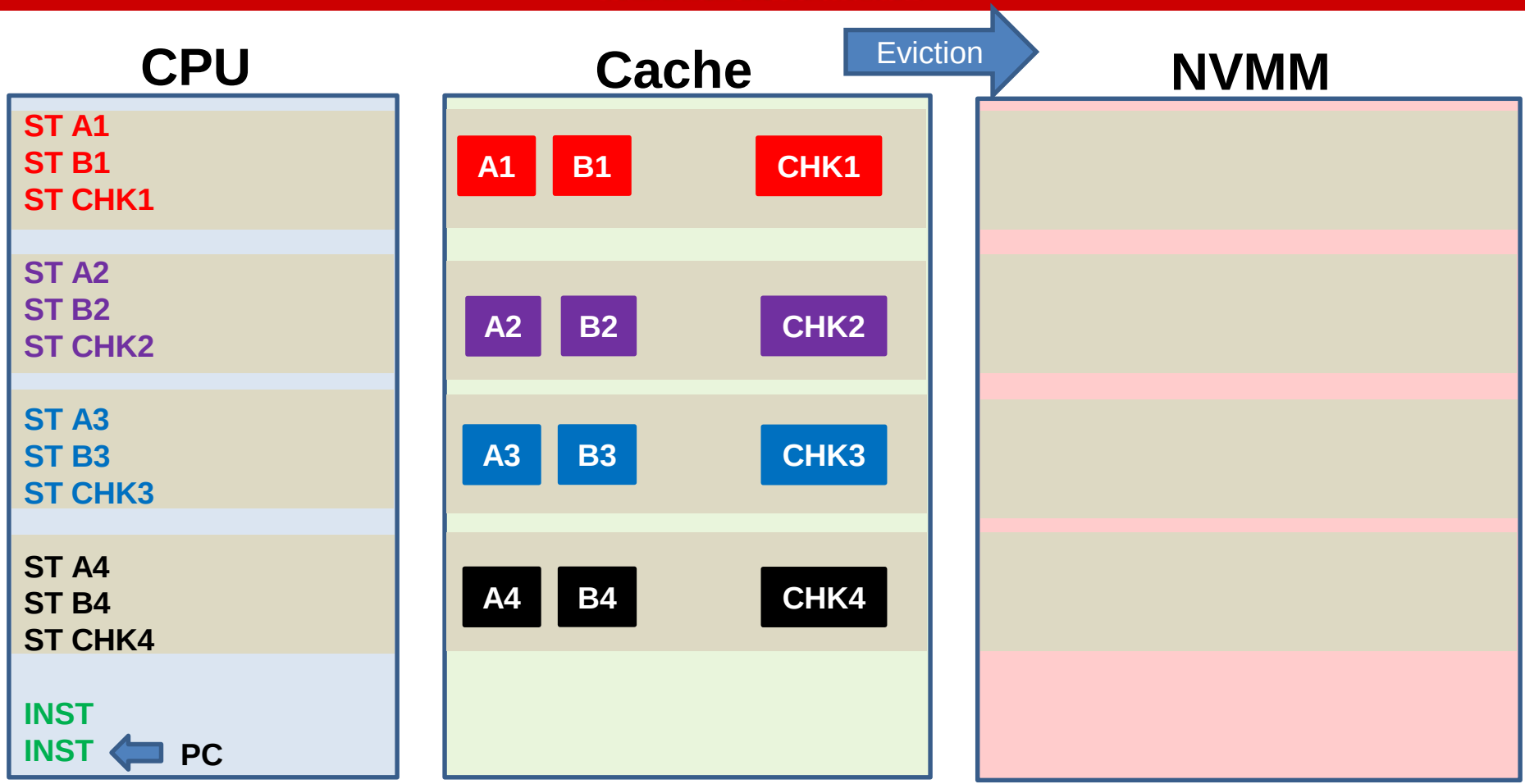
```
1  for (kk=starting_kk; kk<n; kk+=bsize) {
2      for ( ii = starting_ii ; ii <n; ii +=bsize) {
3          ResetCheckSum();
4          for ( jj =0; jj <n; jj +=bsize) {
5              for (i=ii ; i<(ii+bsize); i++) {
6                  for(j=jj ; j<(jj+bsize); j++) {
7                      sum = 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
```

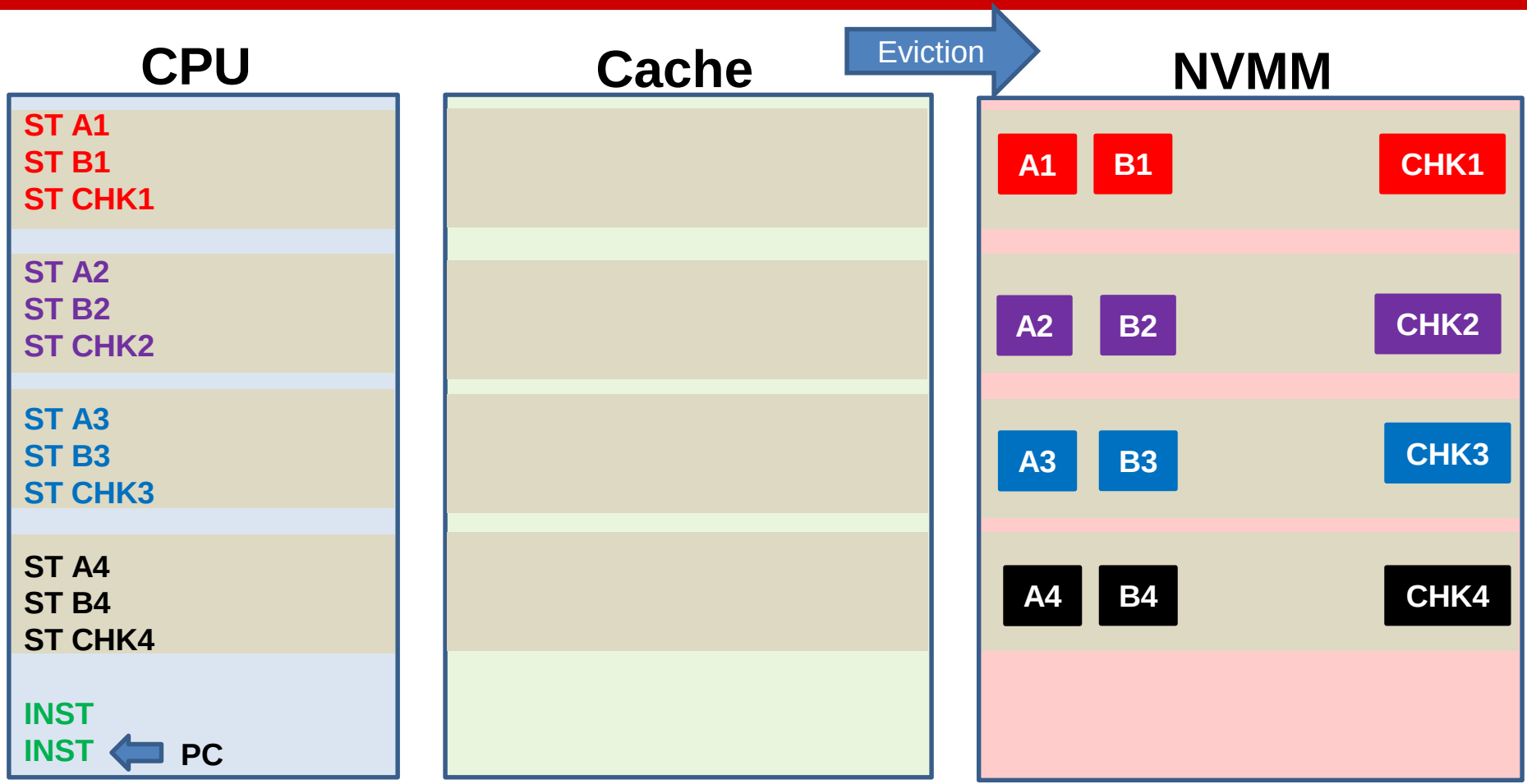
➔ Initialize at the beginning of the region

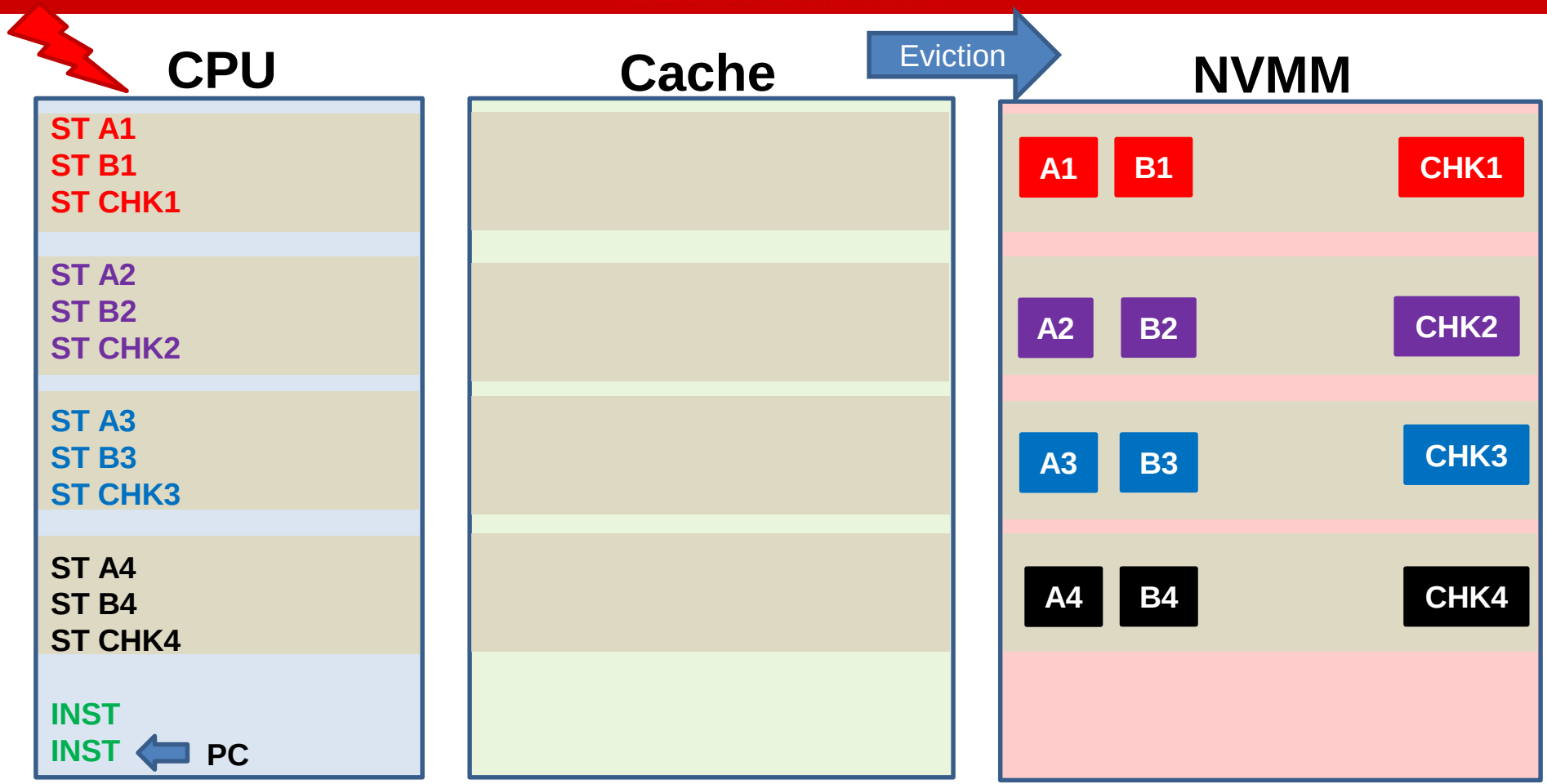
➔ Update during each iteration in the region

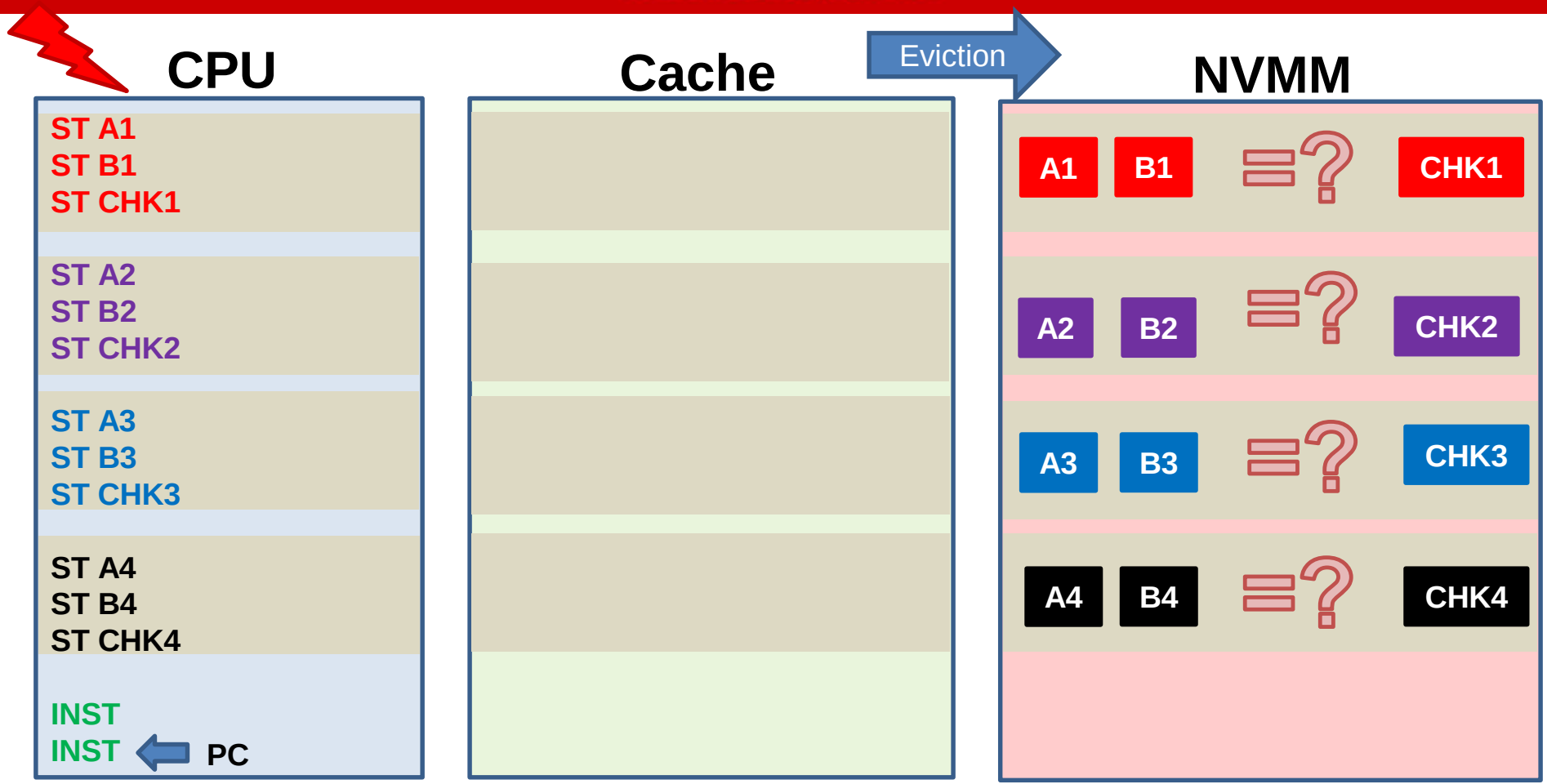
➔ Store the checksum to the corresponding location

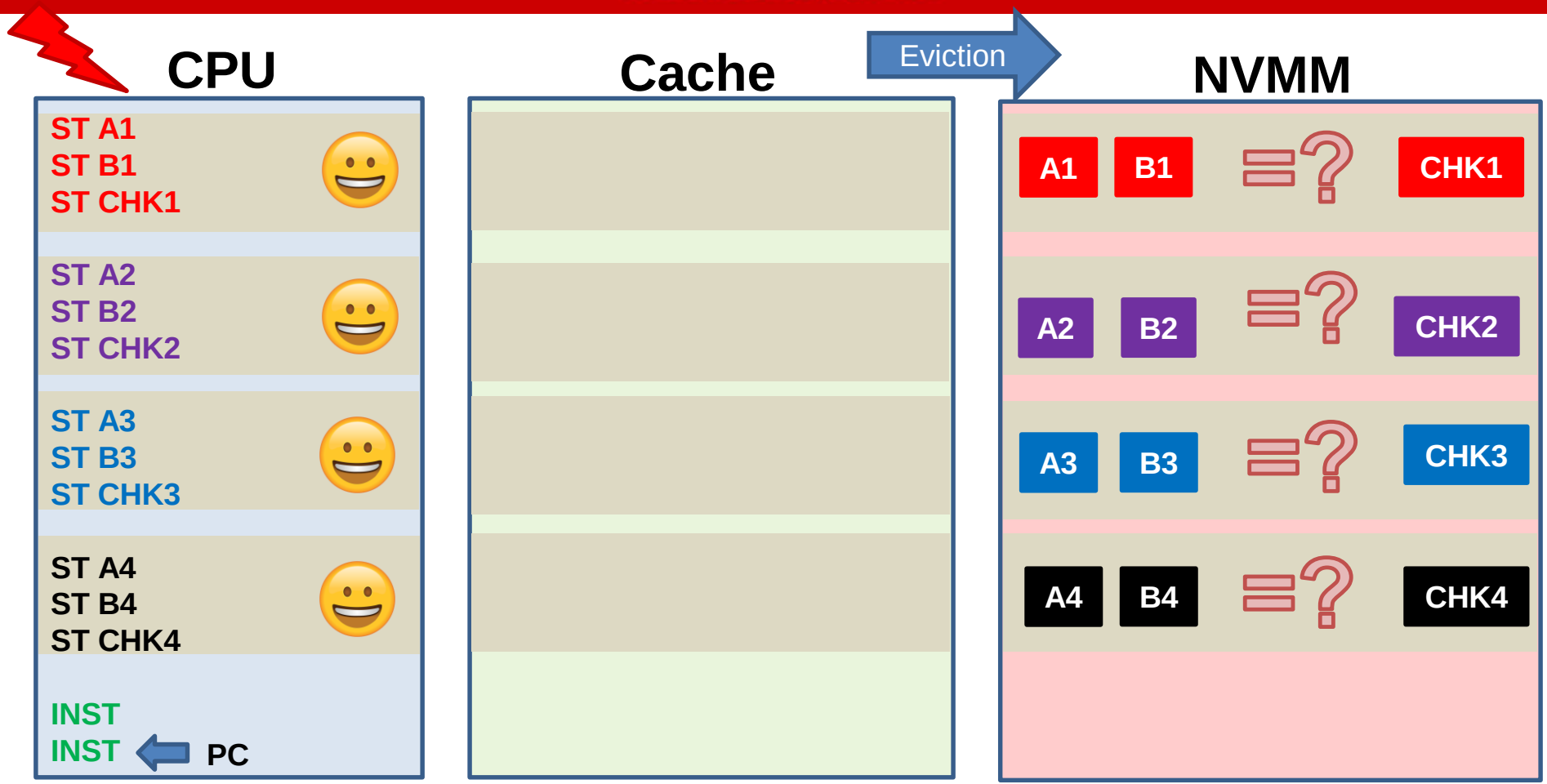
LP Region

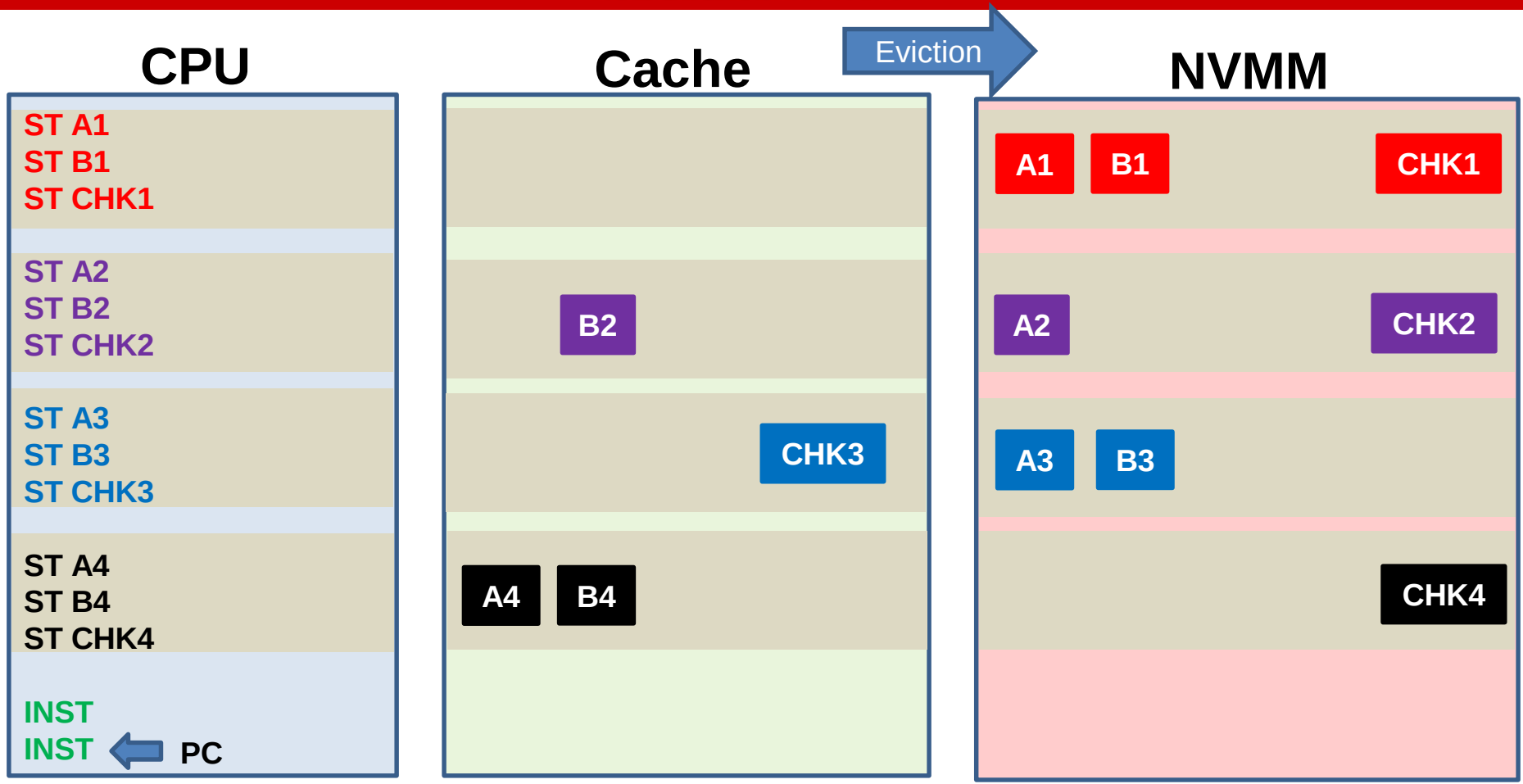

CPU
Cache
Eviction
NVMM
ST A1
ST B1
ST CHK1
ST A2
ST B2
ST CHK2
ST A3
ST B3
ST CHK3
ST A4
ST B4
ST CHK4
INST
INST
PC
A1
B1
CHK1
A2
B2
CHK2
A3
B3
CHK3
A4
B4
CHK4

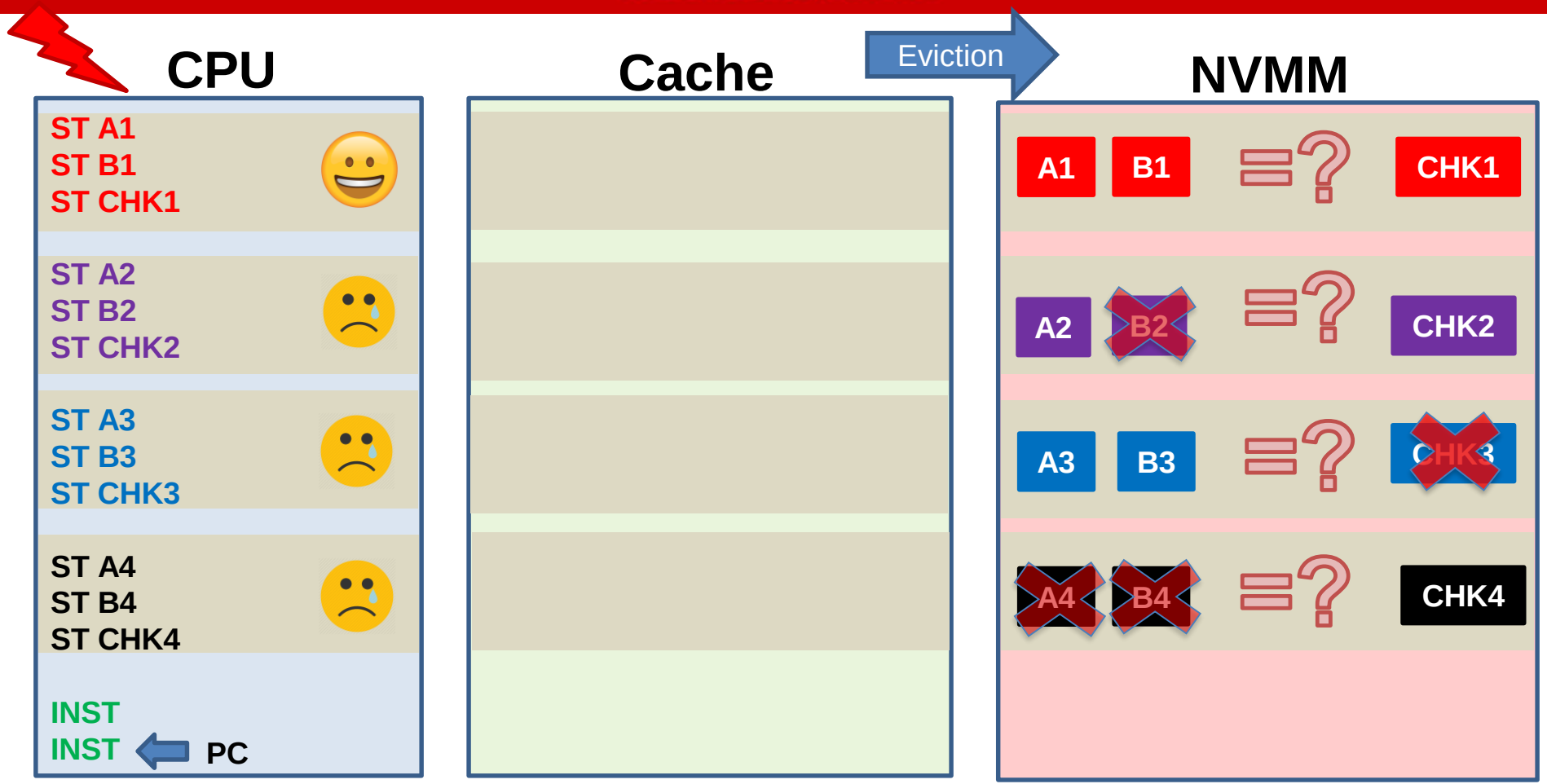

CPU
Cache
Eviction
NVMM
ST A1
ST B1
ST CHK1
ST A2
ST B2
ST CHK2
ST A3
ST B3
ST CHK3
ST A4
ST B4
ST CHK4
INST
INST
PC
A1
B1
CHK1
A2
B2
CHK2
A3
B3
CHK3
A4
B4
CHK4


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

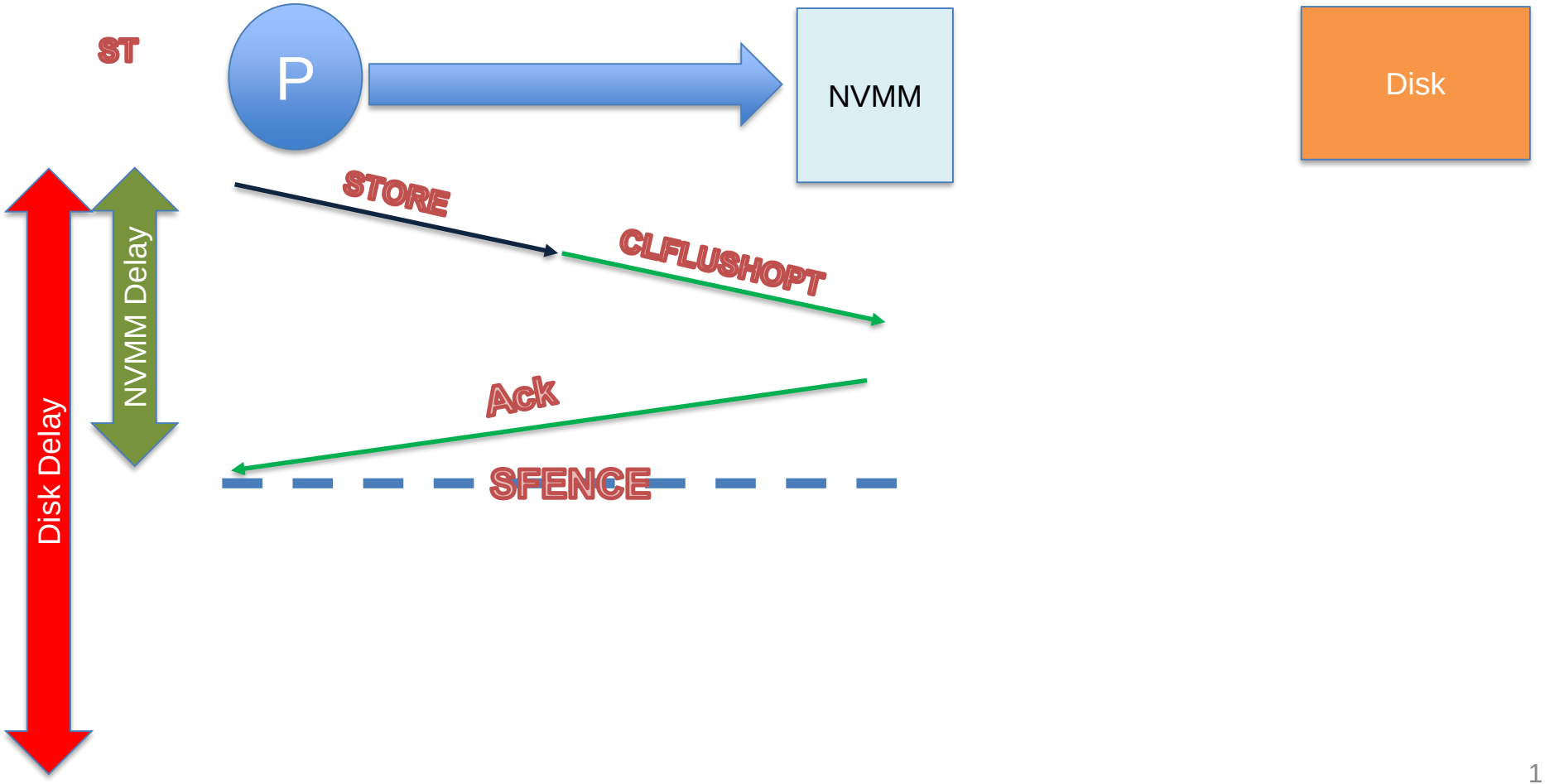

CPU
Cache
Eviction
NVMM
ST A1
ST B1
ST CHK1
ST A2
ST B2
ST CHK2
ST A3
ST B3
ST CHK3
ST A4
ST B4
ST CHK4
INST
INST
PC
A1
B1
CHK1
A2
B2
CHK2
A3
B3
CHK3
A4
B4
CHK4

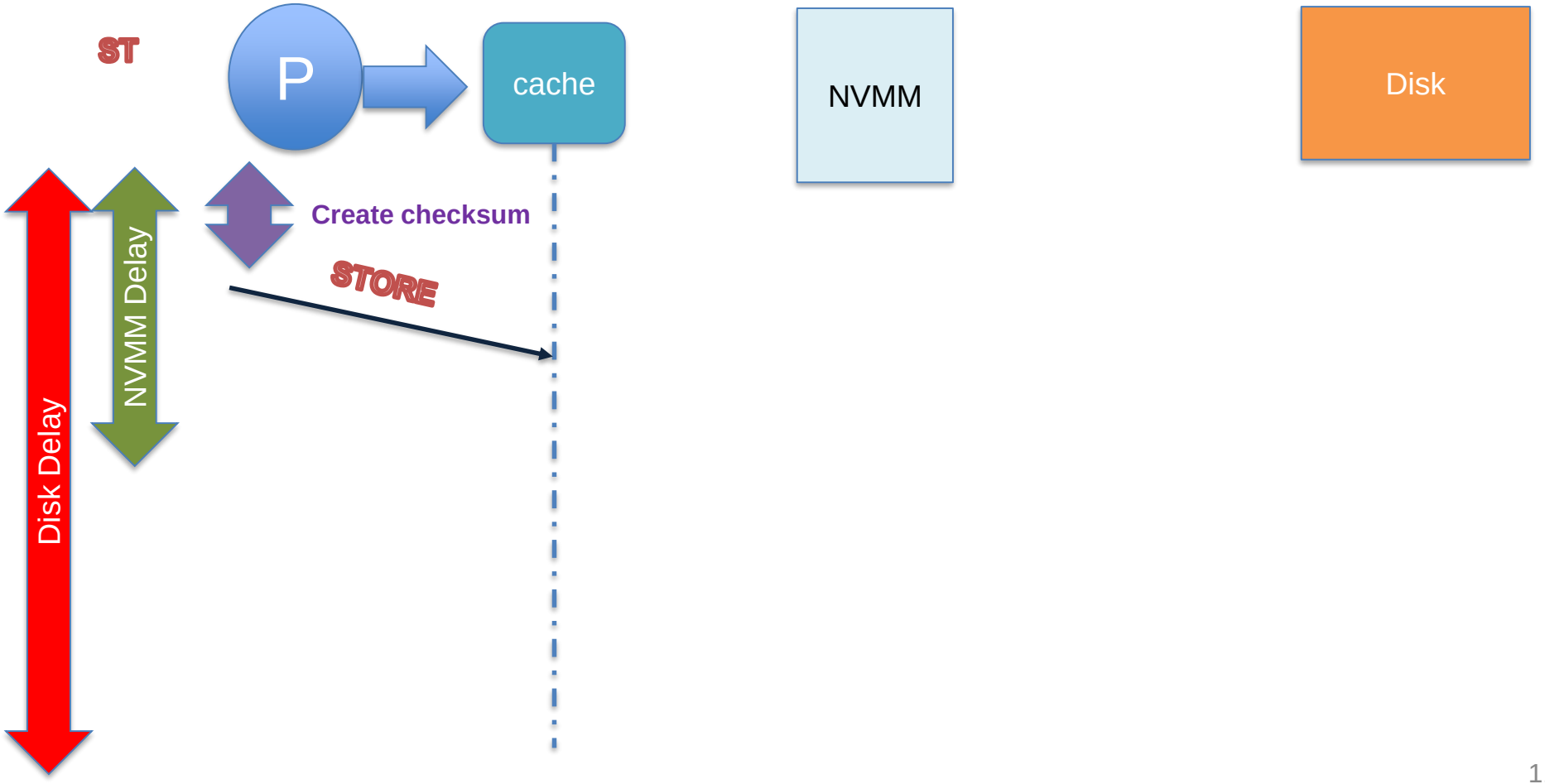

CPU
Cache
Eviction
NVMM
ST A1
ST B1
ST CHK1
ST A2
ST B2
ST CHK2
ST A3
ST B3
ST CHK3
ST A4
ST B4
ST CHK4
INST
INST
PC
B2
CHK3
A4
B4
A1
B1
CHK1
A2
CHK2
A3
B3
CHK4

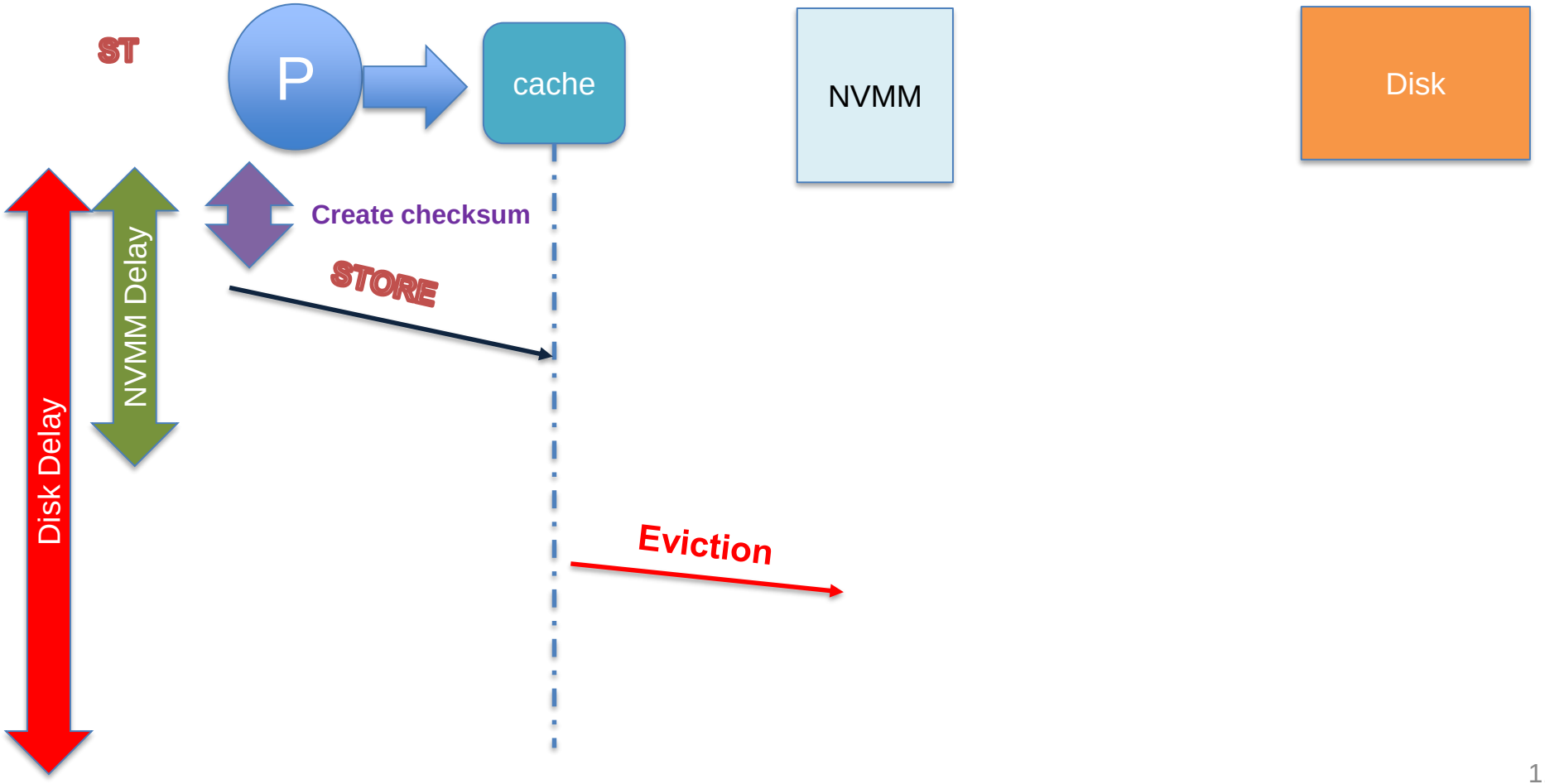

CPU
Cache
Eviction
NVMM
ST A1
ST B1
ST CHK1
ST A2
ST B2
ST CHK2
ST A3
ST B3
ST CHK3
ST A4
ST B4
ST CHK4
INST
INST
PC
A1
B1
CHK1
A2
B2
CHK2
A3
B3
CHK3
A4
B4
CHK4

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
NVMM
Disk
STORE
CLFLUSHOPT
Ack
SFENCE
NVMM Delay
Disk Delay


ST
P
cache
NVMM
Disk
Create checksum
STORE
NVMM Delay
Disk Delay


ST
P
cache
NVMM
Disk
Create checksum
STORE
NVMM Delay
Disk Delay
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: Prior work can be exploited [PACT'17]
- 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 (see paper for details)
- The periodic flush interval puts an upper bound for recovery work

Evaluation

Methodology

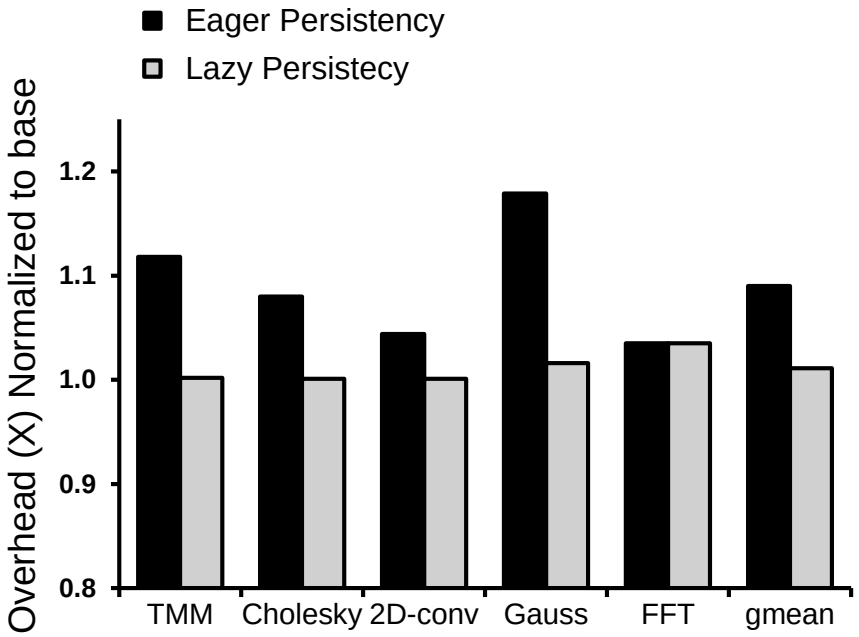
- Simulations on a modified version of gem5. Supports most Intel PMEM instructions (e.g. CLFLUSHOPT)
- Detailed out-of-order CPU model. Ruby memory system. 8 threads is the default for all experiments
- Evaluation was also done on 32-core DRAM-based real hardware machine

Evaluation

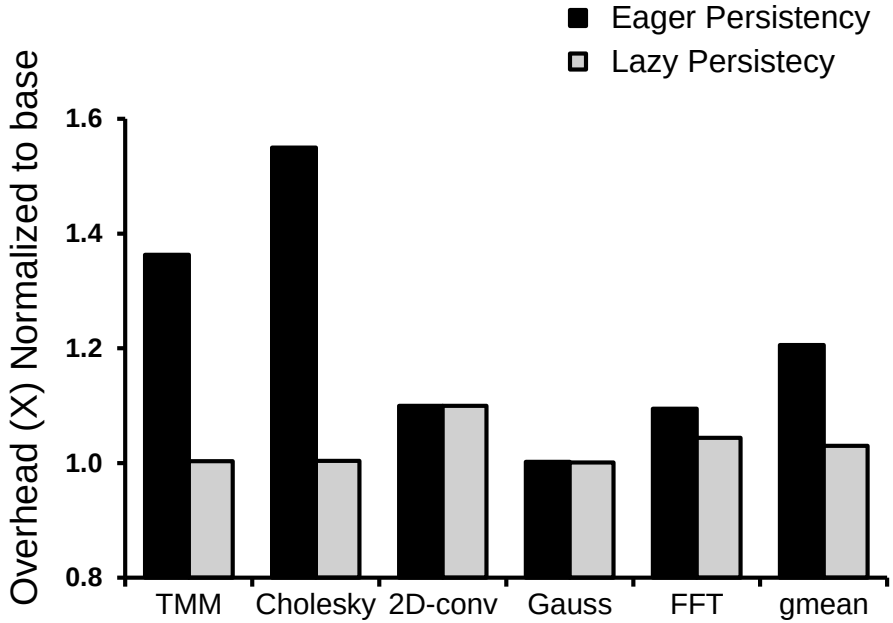
Multi-Threaded Benchmarks

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

Evaluation: All Benchmarks

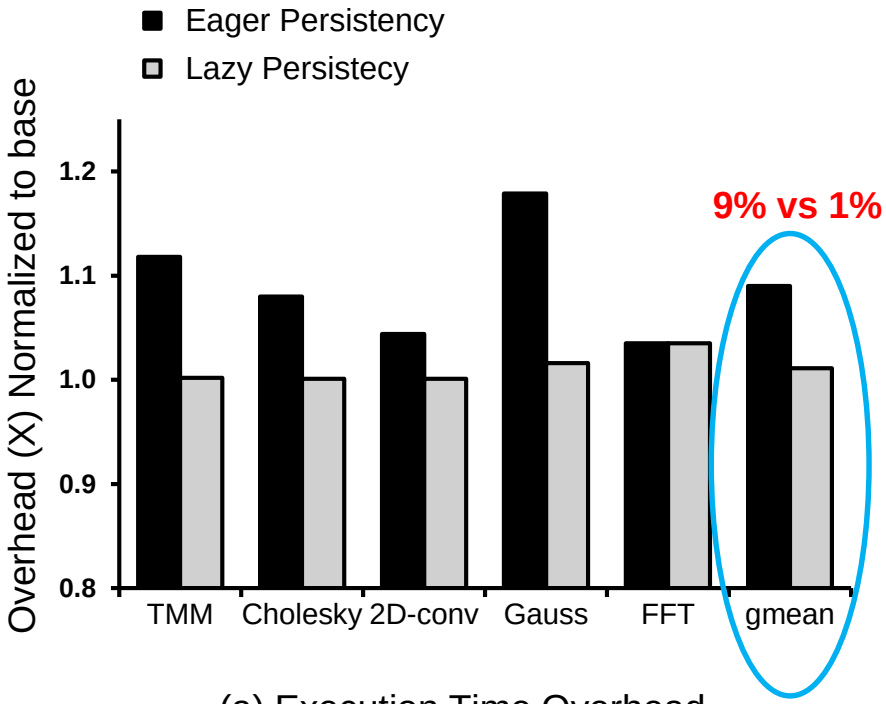


(a) Execution Time Overhead

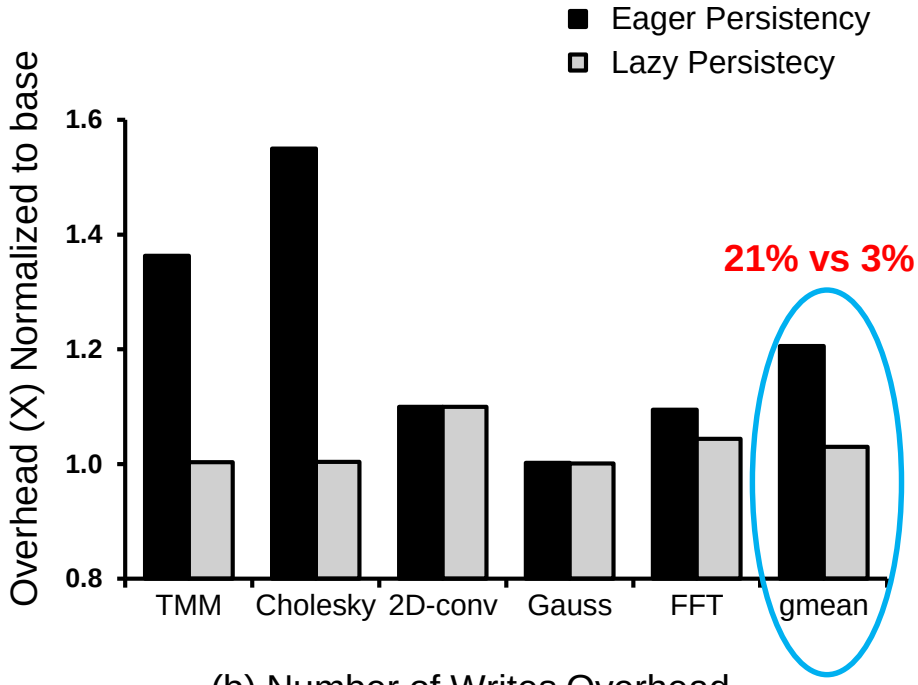


(b) Number of Writes Overhead

Evaluation: All Benchmarks



(a) Execution Time Overhead



(b) Number of Writes Overhead

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?