P-INSPECT: Architectural Support for Programmable Non-Volatile Memory Frameworks

Apostolos Kokolis, Thomas Shull, Jian Huang and Josep Torrellas
University of Illinois at Urbana-Champaign

International Symposium on Microarchitecture (MICRO) 2020

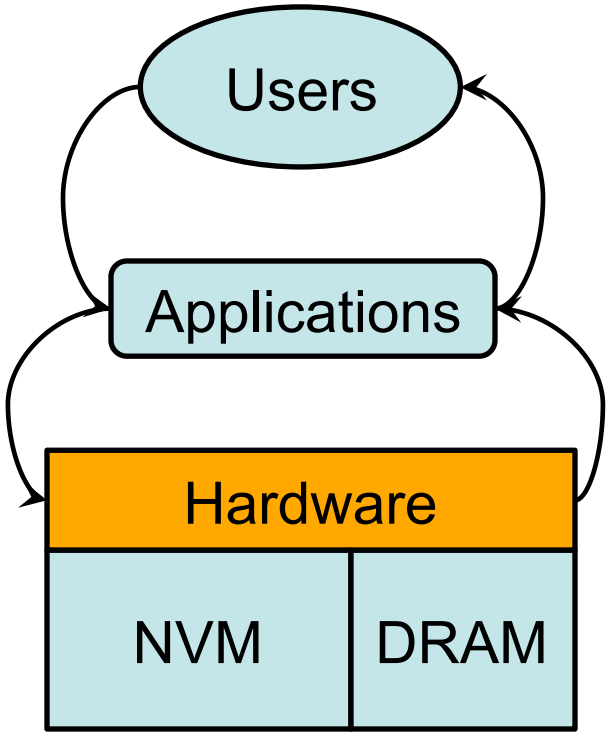
ILLINOIS

Non-Volatile Memory (NVM)

- NVM offers appealing characteristics
  - Capacity
  - Performance
  - Persistence
- Applications have direct access to NVM at byte-granularity
- NVM is already available

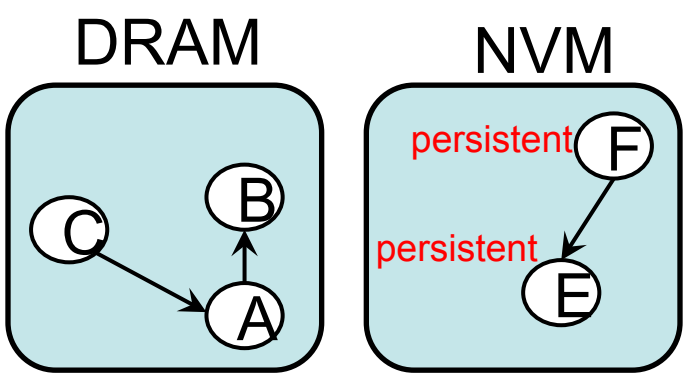
Programming NVM

- NVM success depends on its programming frameworks
- Frameworks should be **efficient** and **easy to use**
- NVM programming challenges
  - API presented to the programmer
  - Identifying persistent objects
  - Performing updates to NVM



Most NVM Frameworks are Hard to Use

- Rely on user involvement
  - Identify persistent data structures
  - Add *clwb* and *sfence*
  - Add *Logging* for transactions
  - Use libraries with NVM support

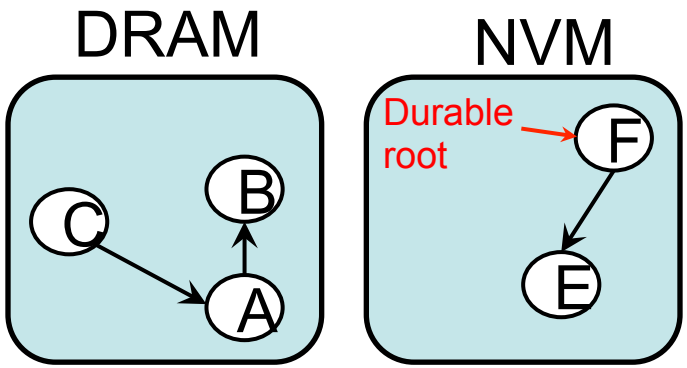


- Efficient ☺
- Increases programming effort ☹
- Code is prone to bugs and inefficiencies ☹
- Hard to apply to legacy code ☹

```
function foo() {
   f = persistent F()
   f.element = value
   clwb f
   sfence
}
```

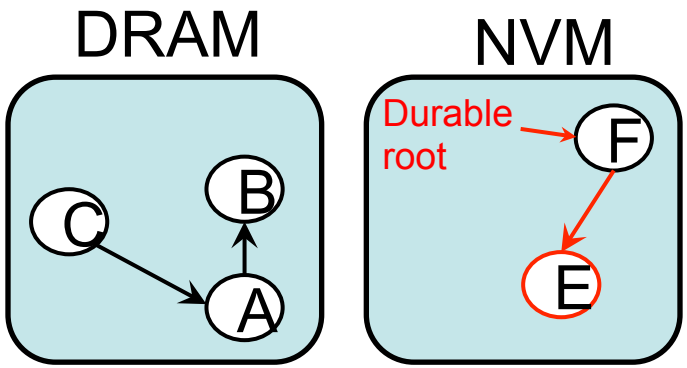
Programmable Framework: Persistence by Reachability

- Programmer only identifies entry points to persistent structures (durable roots)
- The compiler/runtime are responsible to persist necessary objects dynamically



Programmable Framework: Persistence by Reachability

- Programmer only identifies entry points to persistent structures (durable roots)
- The compiler/runtime are responsible to persist necessary objects dynamically



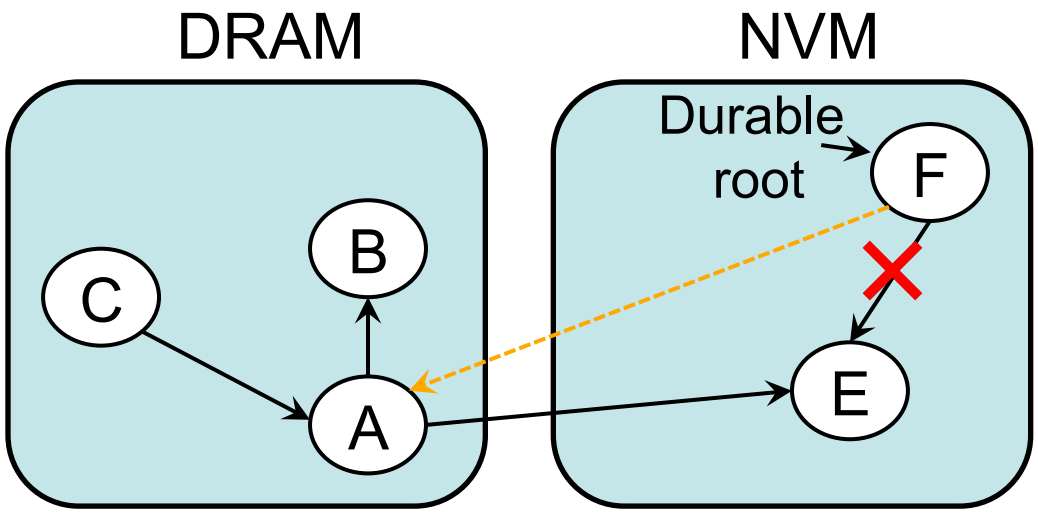
- Simplify programming ☺
- Compatible with legacy code ☺
- Only move data to NVM when needed ☺

The runtime introduces overheads to check object state during execution ☹

Contribution: P-INSPECT

- Transparently **accelerates** Persistence by Reachability (*P-by-R*) frameworks using simple hardware
- Introduces hardware support to minimize software checks and to speed up persistent writes
- In a state-of-the-art *P-by-R* framework, P-INSPECT decreases the executed instructions by 26% and the execution time by 16%

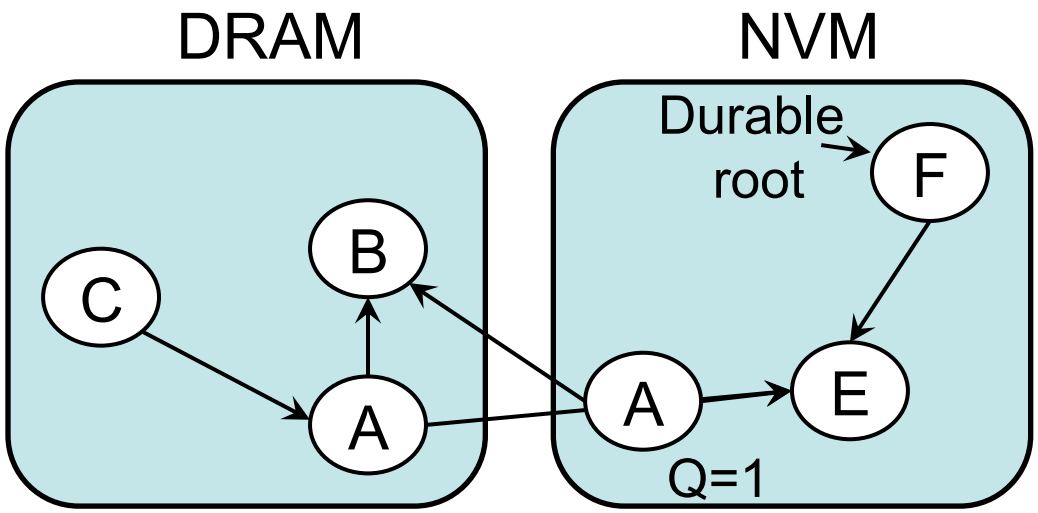
AutoPersist: A Persistence by Reachability Framework[1]



- *F* is a durable root
- *F* wants to point to *A*

[1]: Shull et al., "AutoPersist: An Easy-To-Use Java NVM Framework based on Reachability", PLDI'19

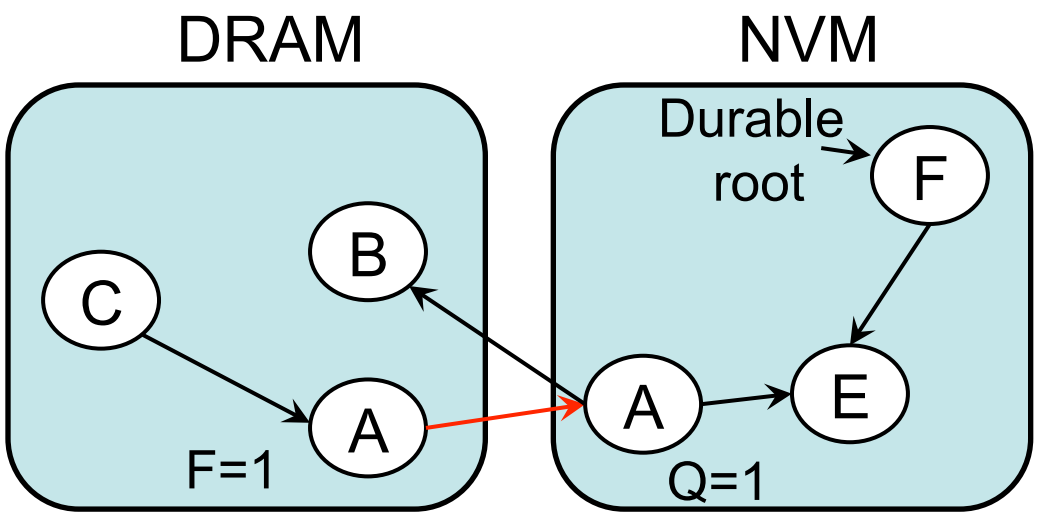
AutoPersist: A Persistence by Reachability Framework



- **Queued bit:** object & its *transitive closure* are being processed

- *F* is a durable root
- *F* wants to point to *A*
- *A* and its transitive closure must become persistent
- Runtime:
  - Creates NVM copy of *A*
  - Sets its Queued bit

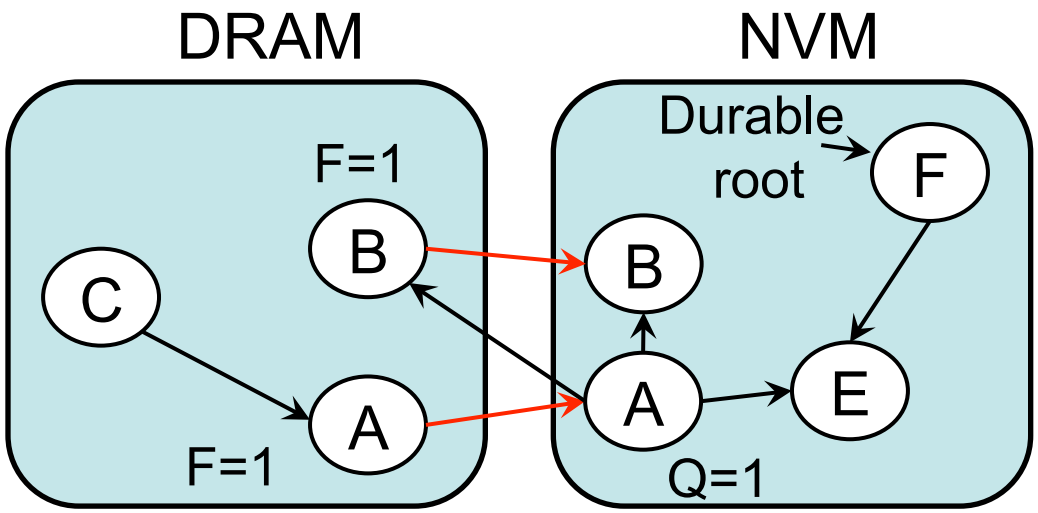
AutoPersist: A Persistence by Reachability Framework



- **Queued bit:** object & its *transitive closure* are being processed
- **Forwarding bit:** object points to its NVM copy

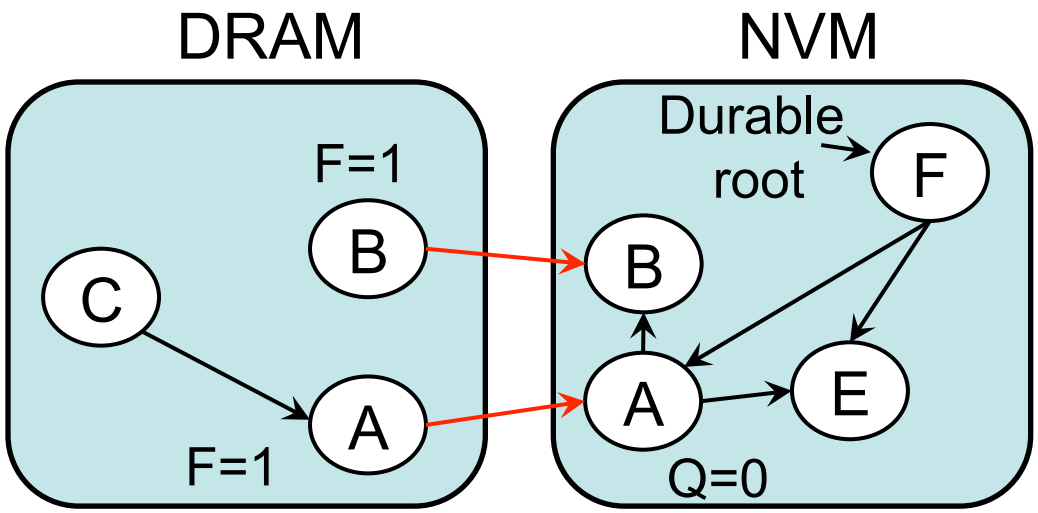
- *F* is a durable root
- *F* wants to point to *A*
- *A* and its transitive closure must become persistent
- Runtime:
  - Creates NVM copy of *A*
  - Sets its Queued bit
  - Sets Forwarding bit of DRAM obj. *A*
  - Creates forwarding pointer for DRAM *A*

AutoPersist: A Persistence by Reachability Framework



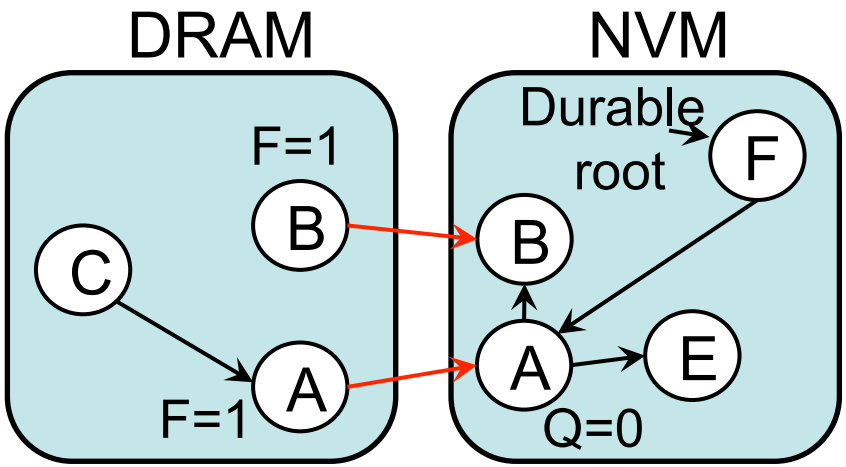
- *F* is a durable root
- *F* wants to point to *A*
- *A* and its transitive closure must become persistent
- Runtime:
  - Creates NVM copy of *A*
  - Sets its Queued bit
  - Sets Forwarding bit of DRAM obj. *A*
  - Creates forwarding pointer for DRAM *A*
  - Repeats for transitive closure of *A*

AutoPersist: A Persistence by Reachability Framework



- *F* is a durable root
- *F* wants to point to *A*
- *A* and its transitive closure must become persistent
- Runtime:
  - Creates NVM copy of *A*
  - Sets its Queued bit
  - Sets Forwarding bit of DRAM obj. *A*
  - Creates forwarding pointer for DRAM *A*
  - Repeats for transitive closure of *A*

Required Runtime Software Checks in AutoPersist



Load: reg = $holder_{obj}$

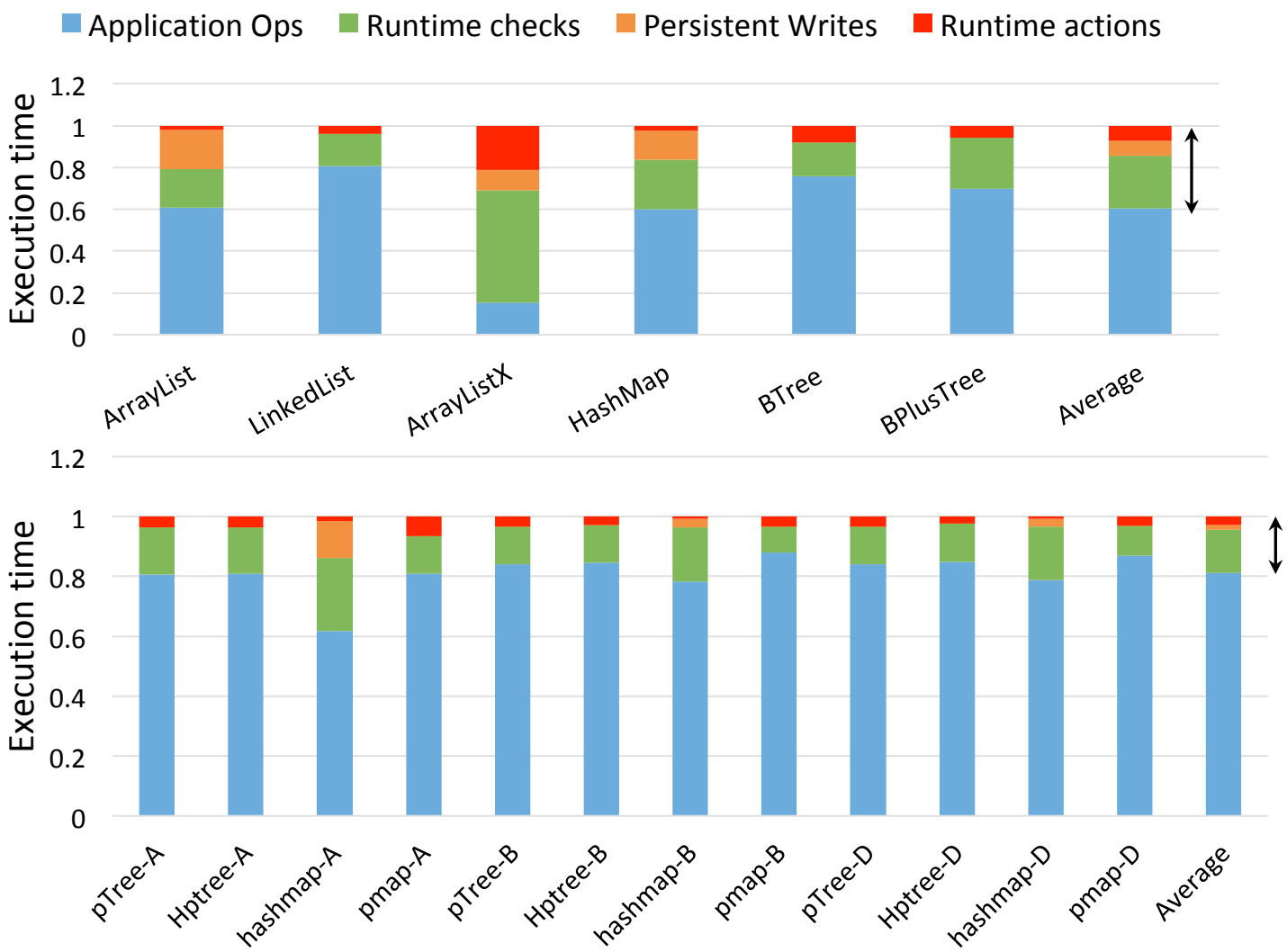
Store: $holder_{obj}.f = value_{obj}$

Runtime must determine if:

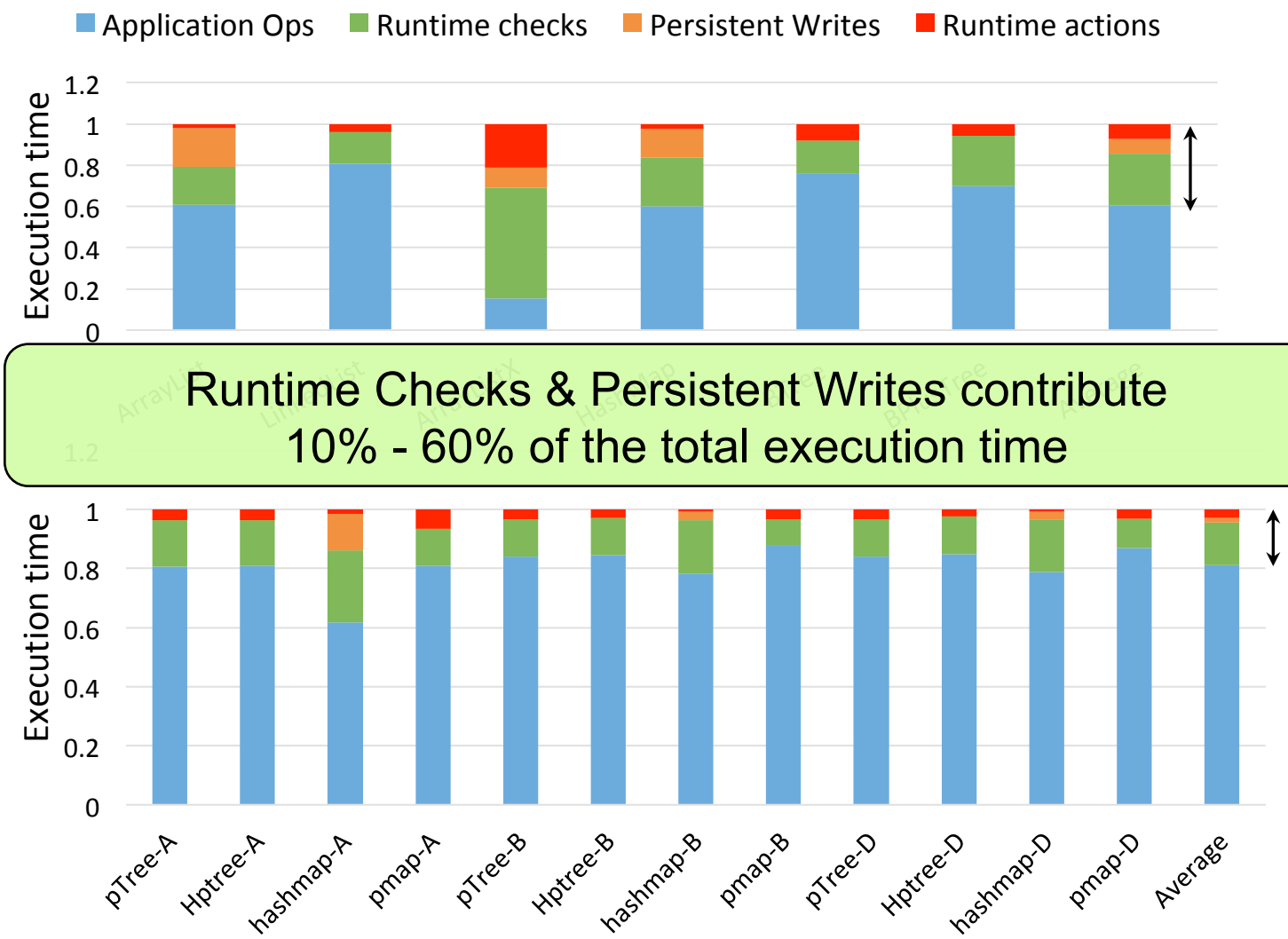
- Object is or needs to be persistent
- Object is forwarding
- Object's trans. closure is processing
- Logging is required

Runtime Checks
Holder's & value's memory locations
Holder's & value's forwarding bits
Value's queued bit
If execution is inside a Xaction

Motivation: AutoPersist has High Overhead



Motivation: AutoPersist has High Overhead



Runtime Checks & Persistent Writes contribute 10% - 60% of the total execution time

P-INSPECT: Design

- **Idea:** Introduce hardware to perform the checks

Before: SW reads object's header to find if object is in NVM

Now: HW checks the Virtual Address of the object

Runtime Checks	Hardware Mechanism
Holder's & value's memory location	Virtual Address
Holder's & value's forwarding bits	
Value's queued bit	
Execution inside a Xaction	

P-INSPECT: Design

- **Idea:** Introduce hardware to perform the checks

Bloom filter (BF) is a cheap HW structure to check for membership
Objects with *Forwarding* bit set are placed in the *Forwarding* BF
Objects with *Queued* bit set are placed in the *Trans. Clos.* BF
On an object access, HW checks for BF membership

Runtime Checks	Hardware Mechanism
Holder's & value's memory location	Virtual Address
Holder's & value's forwarding bits	*Forwarding* bloom filter
Value's queued bit	*Transitive Closure* bloom filter
Execution inside a Xaction	

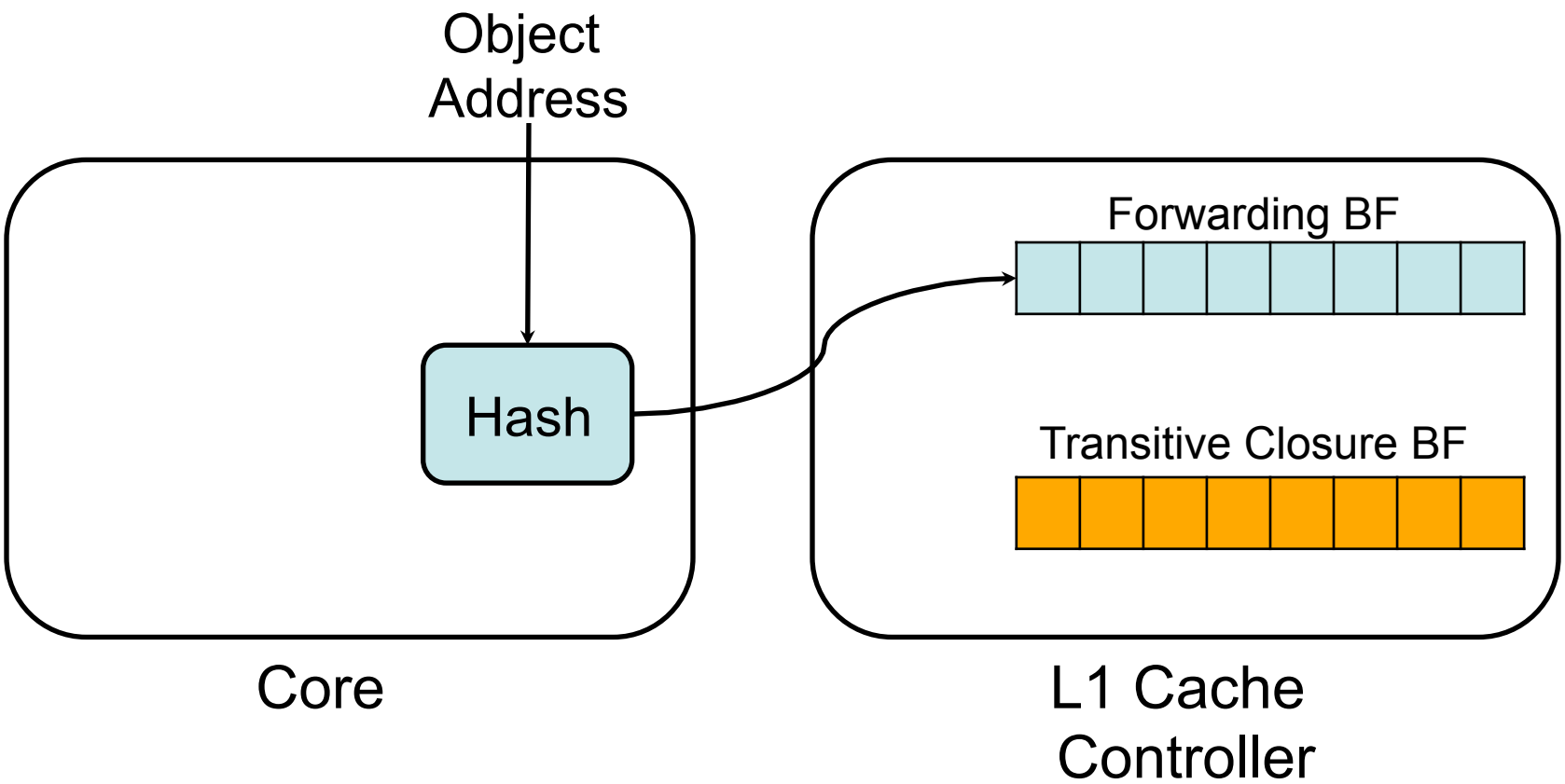
P-INSPECT: Design

- **Idea:** Introduce hardware to perform the checks

Runtime Checks	Hardware Mechanism
Holder's & value's memory location	Virtual Address
Holder's & value's forwarding bits	*Forwarding* bloom filter
Value's queued bit	*Transitive Closure* bloom filter
Execution inside a Xaction	Control register bit

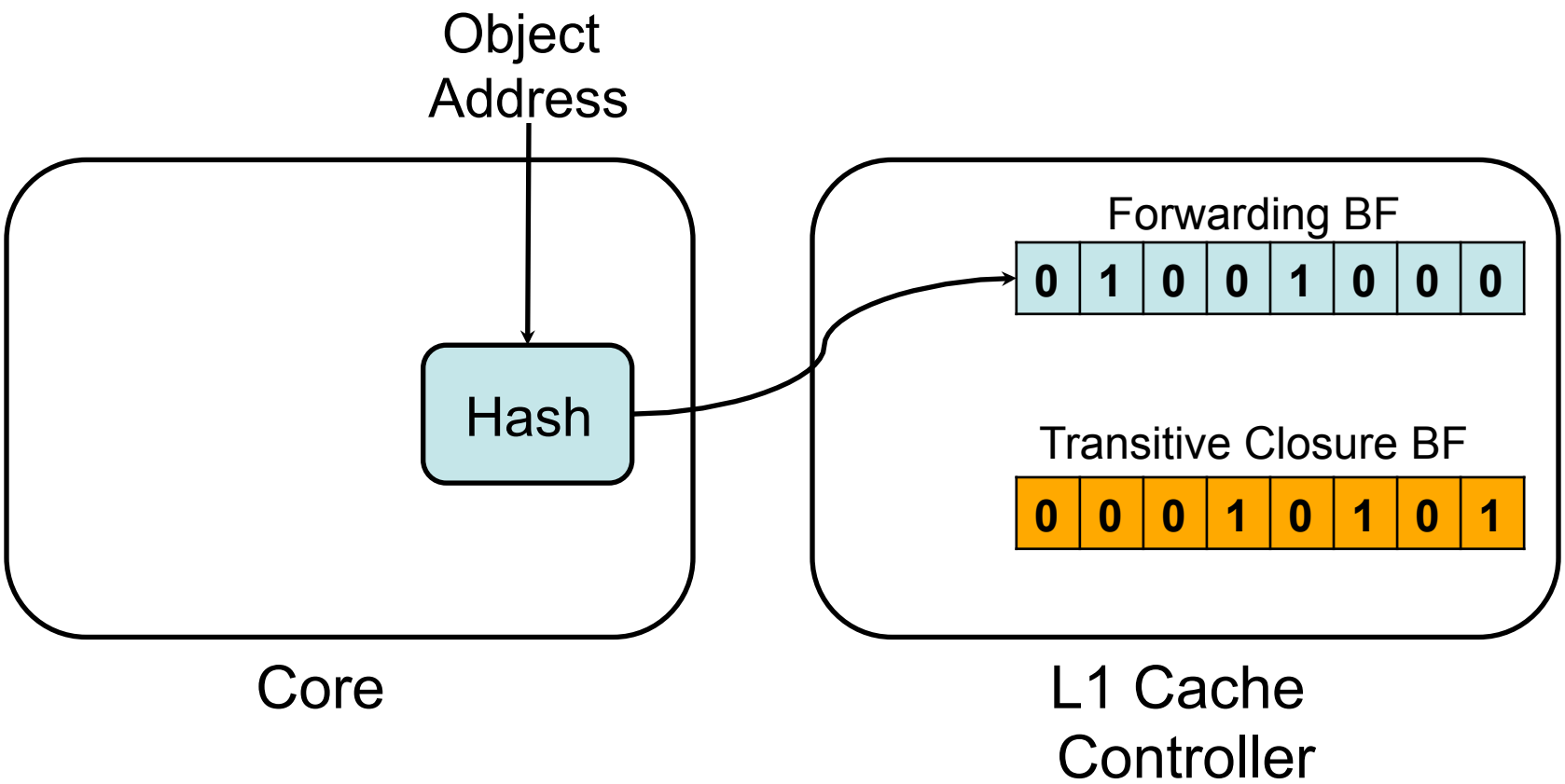
P-INSPECT: Operation of the Bloom filters

- Software adds addresses of objects to the BFs



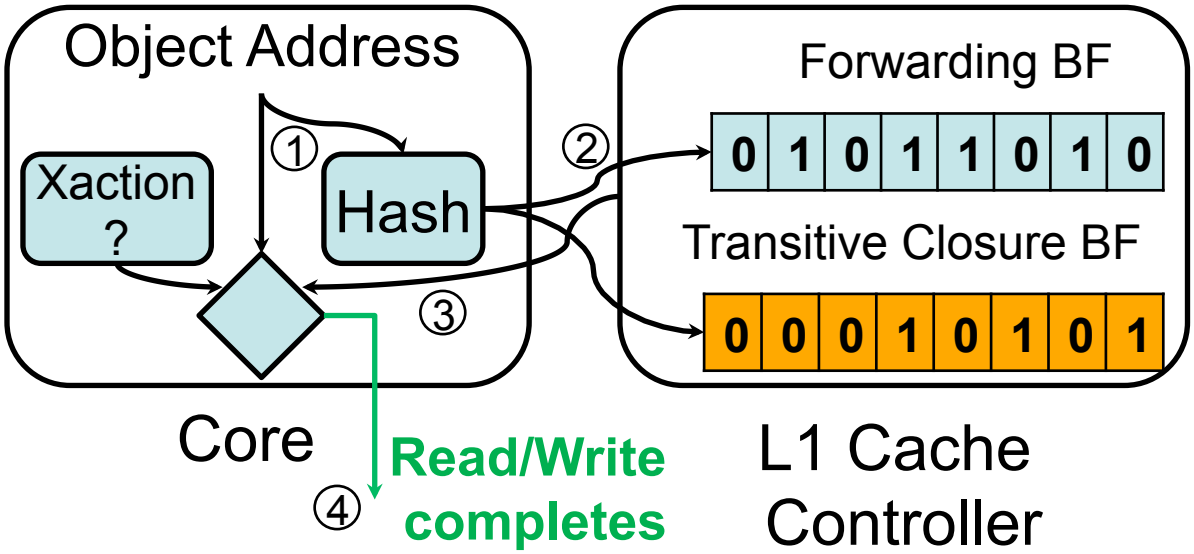
P-INSPECT: Operation of the Bloom filters

- Software adds addresses of objects to the BFs



P-INSPECT: Operation of the Bloom filters

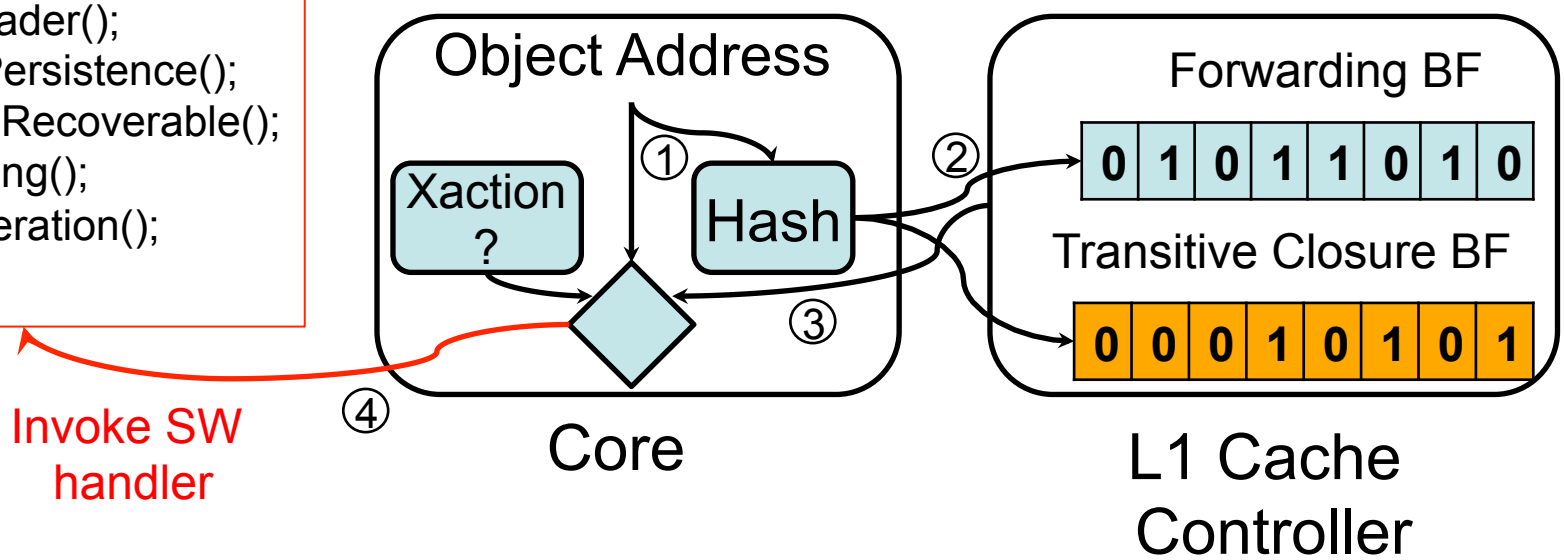
- On a read/write access to an object, HW checks for BF membership (and other checks)
  - If not present and all other checks pass: Read/write completes



P-INSPECT: Operation of the Bloom filters

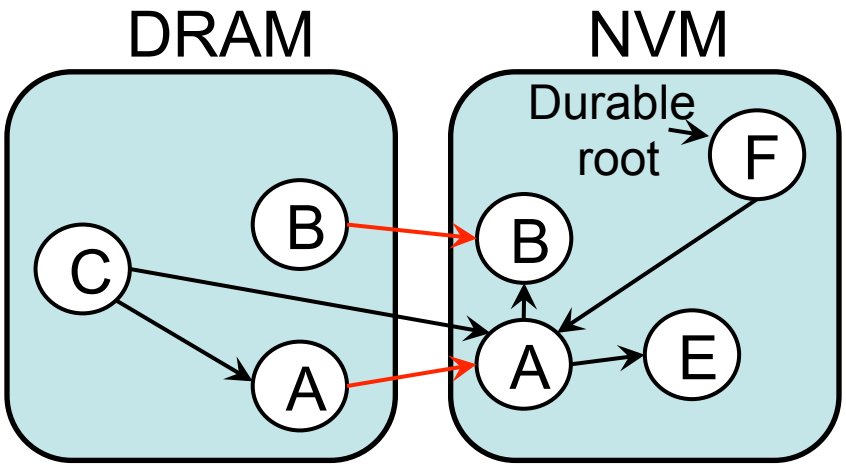
- On a read/write access to an object, HW checks for BF membership (and other checks)
  - If not present and all other checks pass: Read/write completes
  - Else: Invoke SW handler to make the state consistent and read/write

```
function handler(){
  readObjHeader();
  determinePersistence();
  checkMakeRecoverable();
  checkLogging();
  performOperation();
}
```

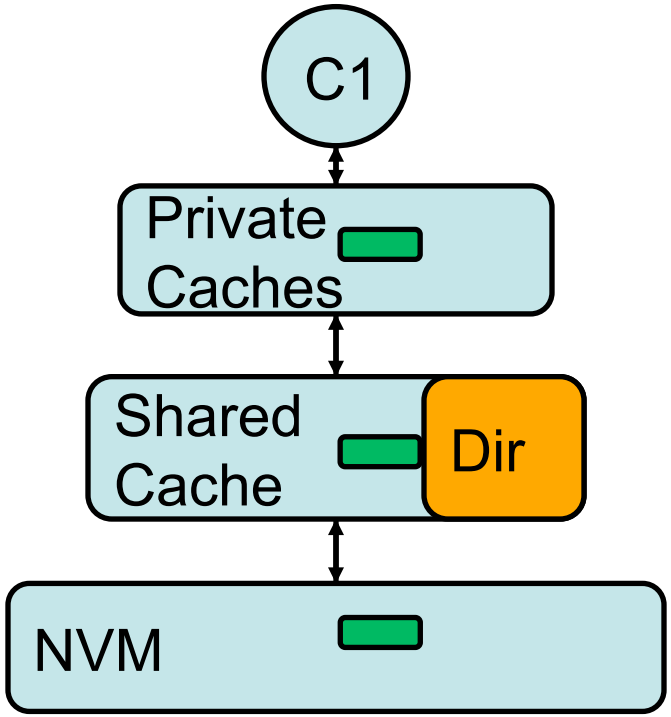


Clearing the *Forwarding* Bloom filter

- A special thread regularly
  - Traverses the volatile heap & fixes the *forwarding* pointers
  - After that it clears the *Forwarding* bloom filter
- At Garbage Collection time: *Forwarding* objects are deleted



Conventional Operation of a Persistent Write



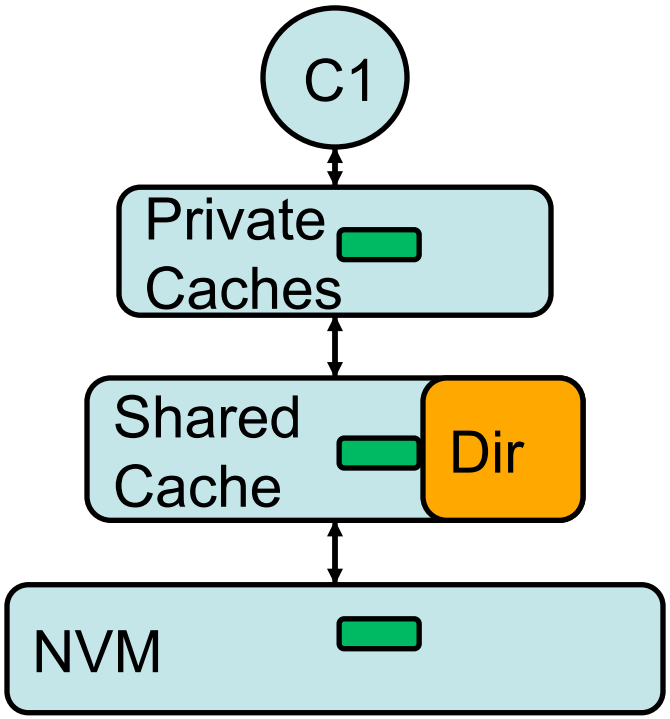
- A persistent write is a combination of
  - store
  - clwb
  - sfence
- Operation
  - The line is first fetched into the private cache
  - The write is performed
  - Line is flushed in NVM
  - Fence is executed

Persistent writes add overhead to application execution

Worst case: Need 2 round trips to NVM

P-INSPECT: Speeds up Persistent Writes

Idea: Combine *write+clwb+sfence* in one transaction



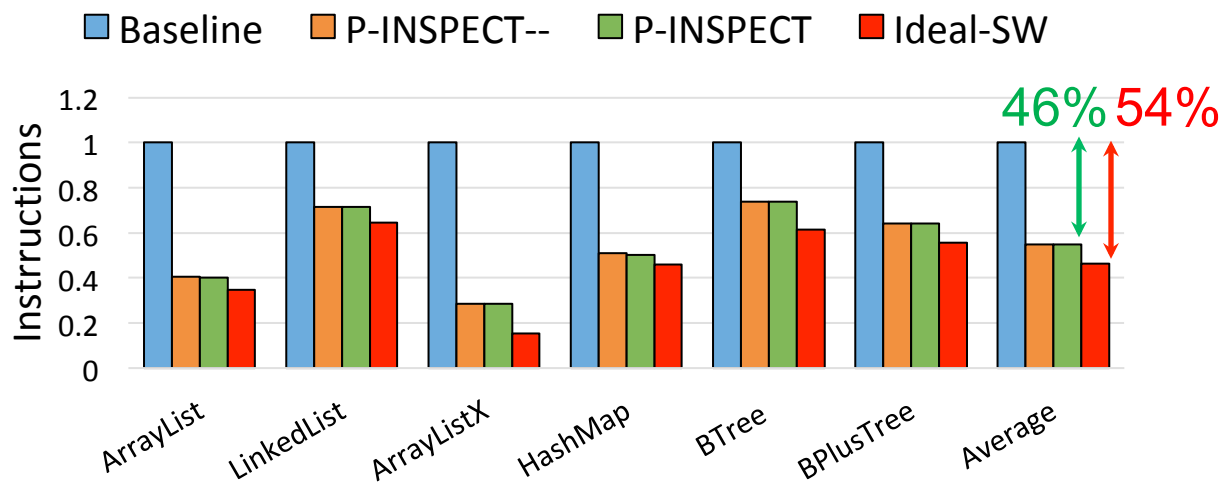
- Operation
  - Write proceeds to NVM
  - NVM is updated
  - Updated cache line is brought into the cache in Exclusive state

Faster execution: Only 1 round trip to NVM

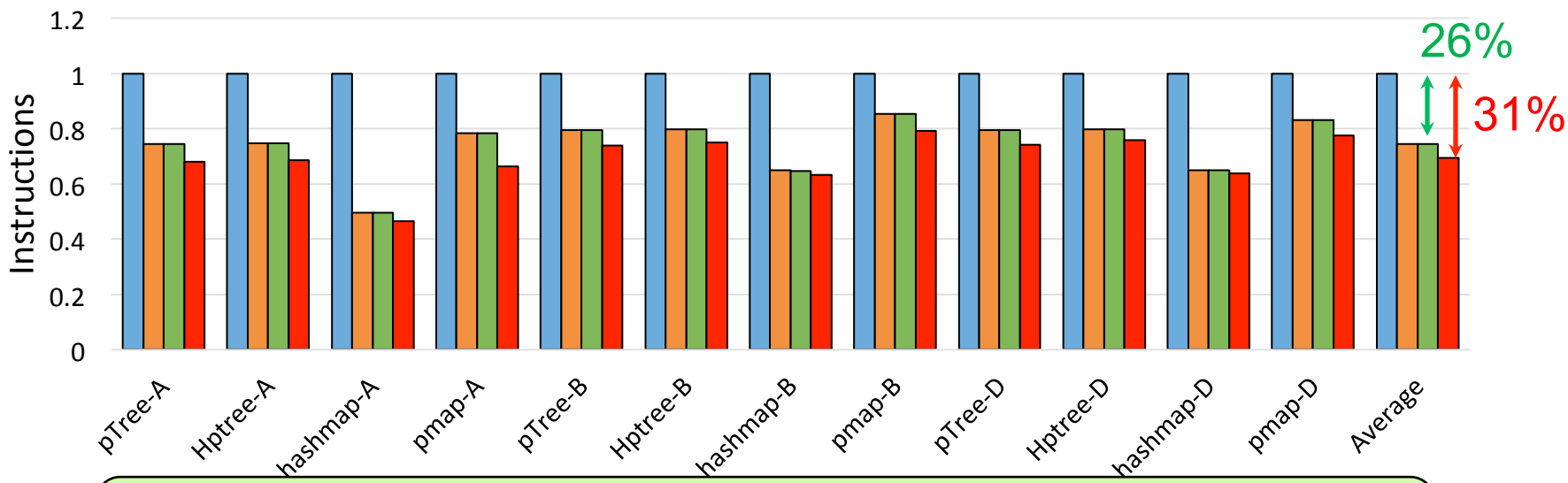
Evaluation Methodology

- Full system simulations with Simics + SST
- Pintool simulations for long execution intervals
- Applications
  - Kernels
  - Key Value Store with many backends running YCSB
- AutoPersist JVM runtime (modified Maxine 2.0.5) to communicate the runtime instructions to the simulators

Reduction in Executed Instructions

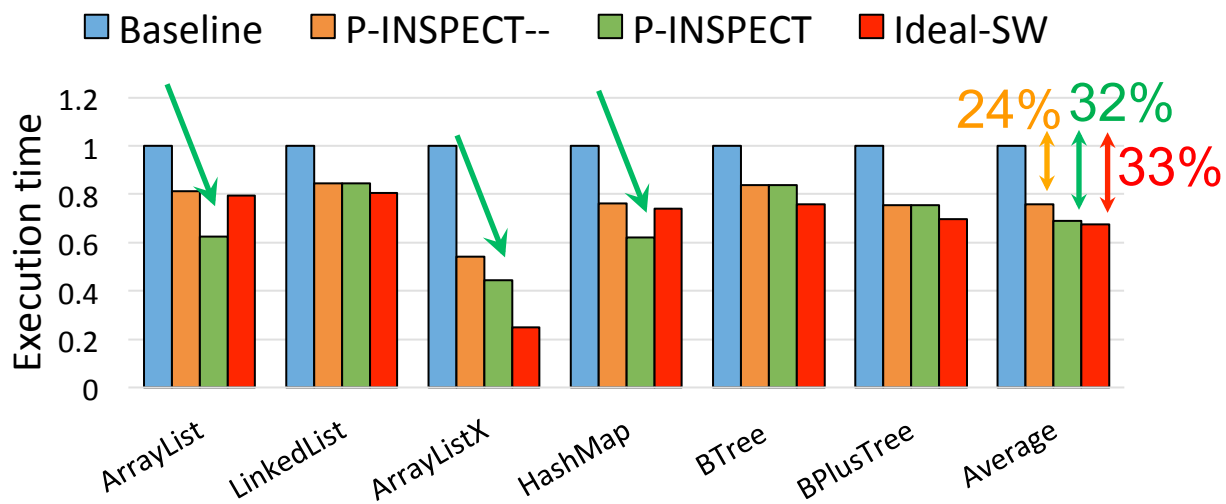


Baseline: AutoPersist
P-INSPECT--: no persistent write optimization
P-INSPECT
Ideal-SW: no P-by-R overheads and no persistent write optimization

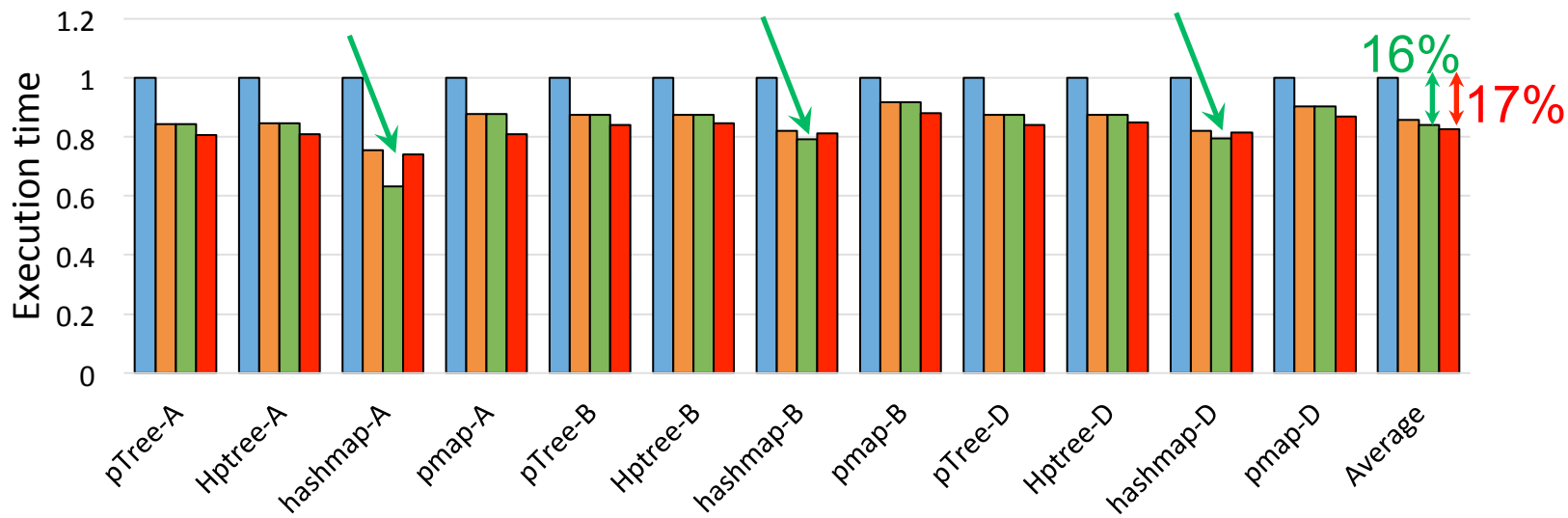


P-INSPECT greatly reduces number of instructions

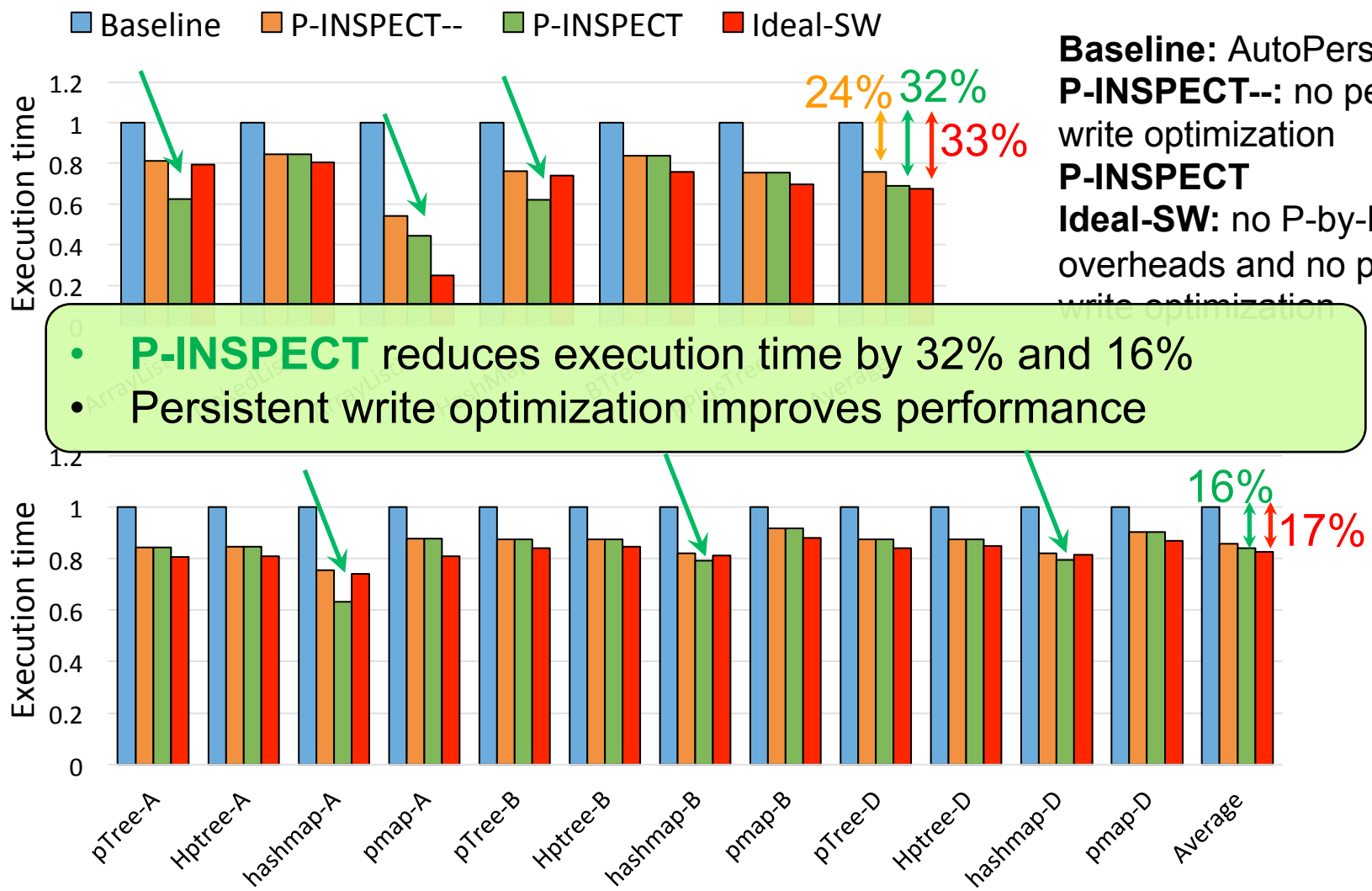
Reduction in Execution Time



Baseline: AutoPersist
P-INSPECT--: no persistent write optimization
P-INSPECT
Ideal-SW: no P-by-R overheads and no persistent write optimization



Reduction in Execution Time



Baseline: AutoPersist
P-INSPECT--: no persistent write optimization
P-INSPECT
Ideal-SW: no P-by-R overheads and no persistent write optimization

- **P-INSPECT** reduces execution time by 32% and 16%
- Persistent write optimization improves performance

Also in the paper…

- More details on P-INSPECT
  - Operation and instructions
  - Software handlers
  - Bloom filter operations

- Deeper Evaluation

Conclusion

P-INSPECT uses hardware to speed up programmable Persistence by Reachability (*P-by-R*) frameworks

- Keeps *P-by-R programming* advantages
- Performs checks with simple hardware and accelerates persistent writes
- Achieves performance similar to an ideal runtime system with no *P-by-R*

P-INSPECT: Architectural Support for Programmable Non-Volatile Memory Frameworks

Apostolos Kokolis, Thomas Shull, Jian Huang and Josep Torrellas
University of Illinois at Urbana-Champaign
{kokolis2, shull1, jianh, torrella}@Illinois.edu
International Symposium on Microarchitecture (MICRO) 2020

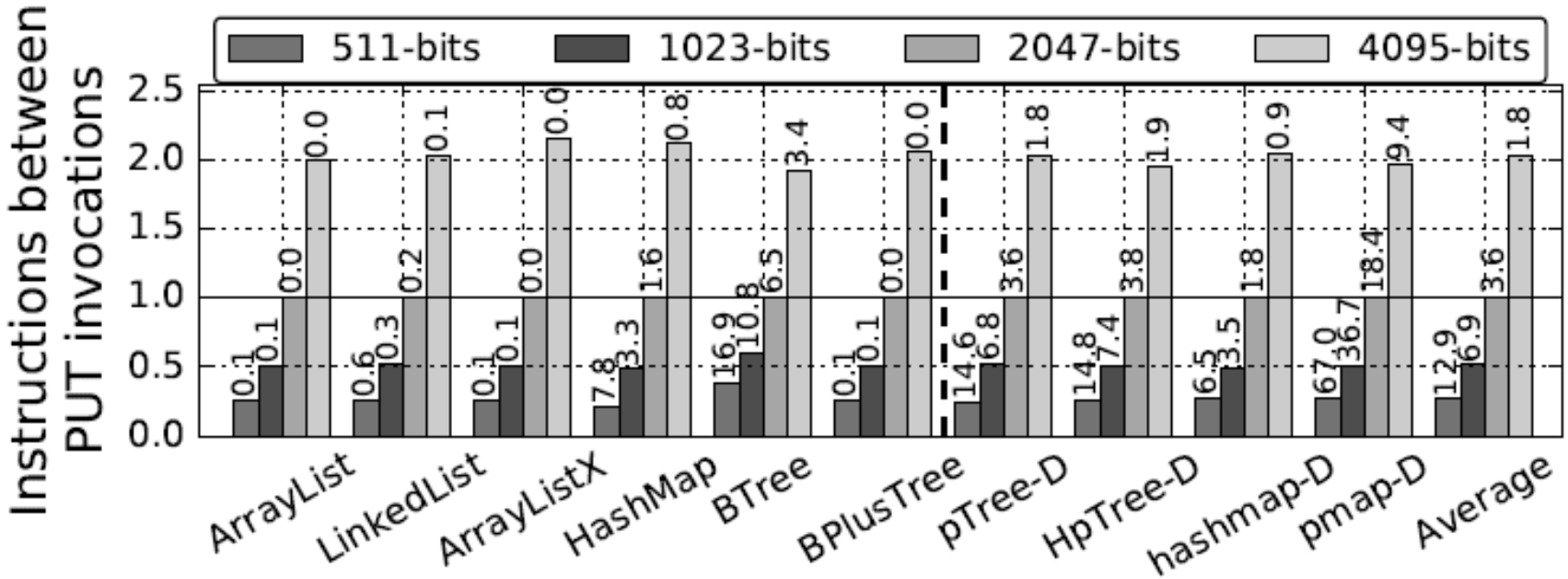
THANK YOU!!

ILLINOIS

Non-Blocking Pointer Update Thread

- The Pointer Update Thread (PUT) traverses the live objects of the volatile heap and updates pointers to FWD objects
- PUT operation is non-blocking
- We keep 2 FWD bloomfilters
  - Active
  - Inactive
- Both are checked on a special load/store
- Insert to active
- Clear inactive when PUT completes

Pointer Update Thread Overhead



Pointer Update Thread Overhead

Applic.	# Inst. between PUT calls (mill.)	# FWD checks per insert (thous.)	Avg. FWD occup.	PUT instr.
ArrayList	26,326	3,006.0	14.5%	0.0%
LinkedList	3,175	163.5	15.9%	0.2%
ArrayListX	43,778	4,937.4	15.8%	0.0%
HashMap	928	134.8	15.9%	1.6%
BTree	237	10.4	15.9%	6.5%
BPlusTree	45,367	3,201.0	15.9%	0.0%
pTree-D	478	22.4	16.0%	3.6%
HpTree-D	426	11.1	15.8%	3.8%
hashmap-D	969	85.2	16.1%	1.8%
pmap-D	92	1.9	15.9%	18.4%
Average	12,177	1,157.4	15.8%	3.6%

Coherent Bloomfilters

- Bloomfilters (BFs) span multiple cache lines
- To read the BFs a core must first fetch the cache lines to the L1 cache controller
- For a write access to the BF one of the lines of the BF acts as the Seed
  - The Seed is requested in Exclusive state
  - Once it attains the Seed locks it and proceeds to obtain the rest of the cache lines
  - When all the lines are present the write operation is performed

Speeding-up persistent writes

- Interaction with coherence protocol
  - If the cache line is in one of the caches it is incorporated in the write
  - Once it reaches the directory, the directory is locked. The directory indicates if the line is in another's core cache
  - The NVM ACKs the directory. Directory marks the line as Exclusive for the originating persistent-write core

New instructions

Name	What it Does
checkStoreBoth [Ha],Va	Performs checks, then Mem[Ha] = Va
checkStoreH [Ha],value	Performs checks, then Mem[Ha] = value
checkLoad [Ha],dest	Performs checks, then dest = Mem[Ha]
$insertBF_{FWD}$ Addr	Inserts Addr in the FWD bloom filter
$insertBF_{TRANS}$ Addr	Inserts Addr in the TRANS bloom filter
$clearBF_{FWD}$	Clears the FWD bloom filter
$clearBF_{TRANS}$	Clears the TRANS bloom filter

A possible implementation in x86 is to use the existing store and load instructions with a prefix

Complete list of hardware checks

Conditions						Action Taken
Where is Base(Ha)?	Base(Ha) in FWD?	Where is Va?	Va in FWD?	Va in TRANS?	In Xaction?	
NVM	-	NVM	-	false	false	HW
DRAM	false	DRAM	false	-	-	HW
DRAM	false	NVM	-	-	-	HW
DRAM	true / -	- / DRAM	- / true	-	-	SW: ①
NVM	-	DRAM / NVM	-	- / true	-	SW: ②
NVM	-	NVM	-	false	true	SW: ③