Mnemosyne
Lightweight Persistent Memory

Haris Volos

Andres Jaan Tack, Michael M. Swift

University of Wisconsin – Madison

Executive Summary

- Storage-Class Memory (SCM) enables memory-like storage
- Persistent Memory is an abstraction that enables direct access to SCM
- Durable memory transactions allow consistent in-place updates

Technology Trends: SCM

- Features
  - Memory-like interface (load/store)
  - Short access time (1000x faster than flash)
  - Non-volatile
- Technologies
  - Phase Change Memory (PCM)
  - Spin Torque Transfer RAM (STT-RAM)
  - Memristors
  - Flash-backed DRAM (+ supercapacitor)

Application Trends: Low-latency storage

- Web applications
  - Amazon
  - Facebook
- Desktop applications
  - Firefox
- Other
  - Distributed agreement protocols
  - High-frequency trading

GOAL: Flexible Low-latency Storage

- Idea 1: File System
  - Layering overheads
- Idea 2: Persistent Object Stores
  - Single data-storage model

- Our idea: Persistent Memory
  - +Flexible and fine-grain
  - +Low-latency

Persistent Memory

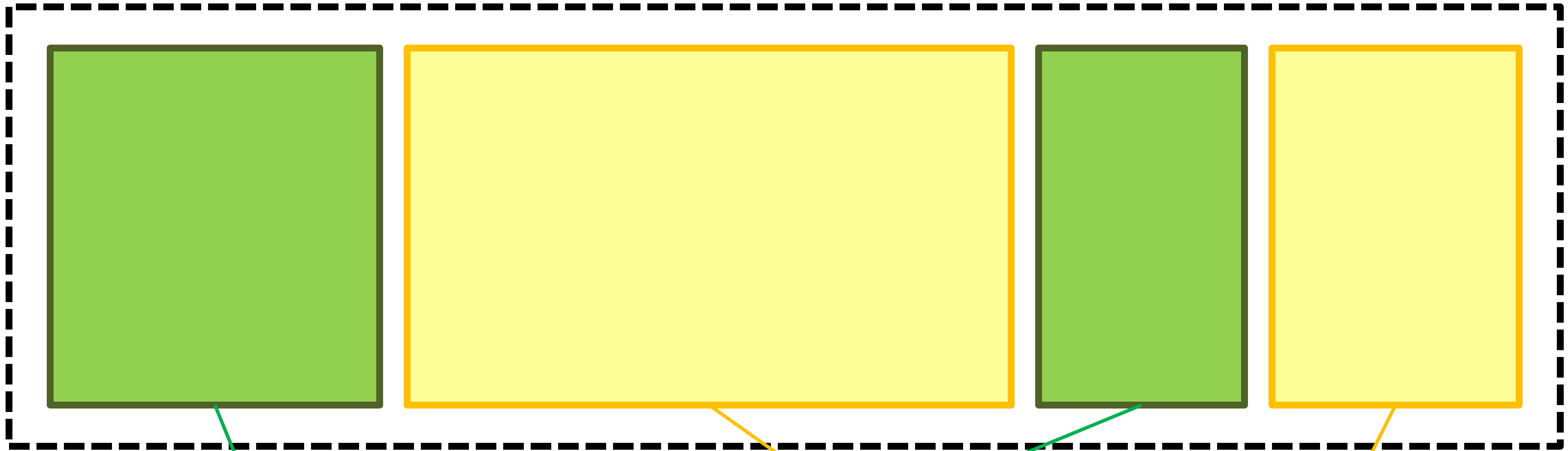
Memory Controller

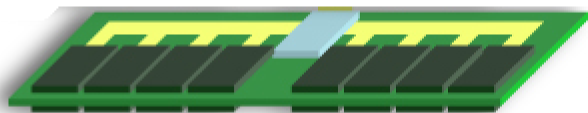
DRAM

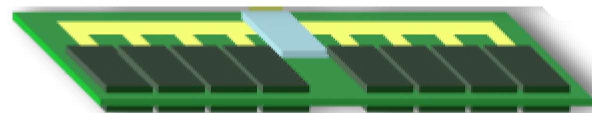
SCM

Persistent Memory

Address Space



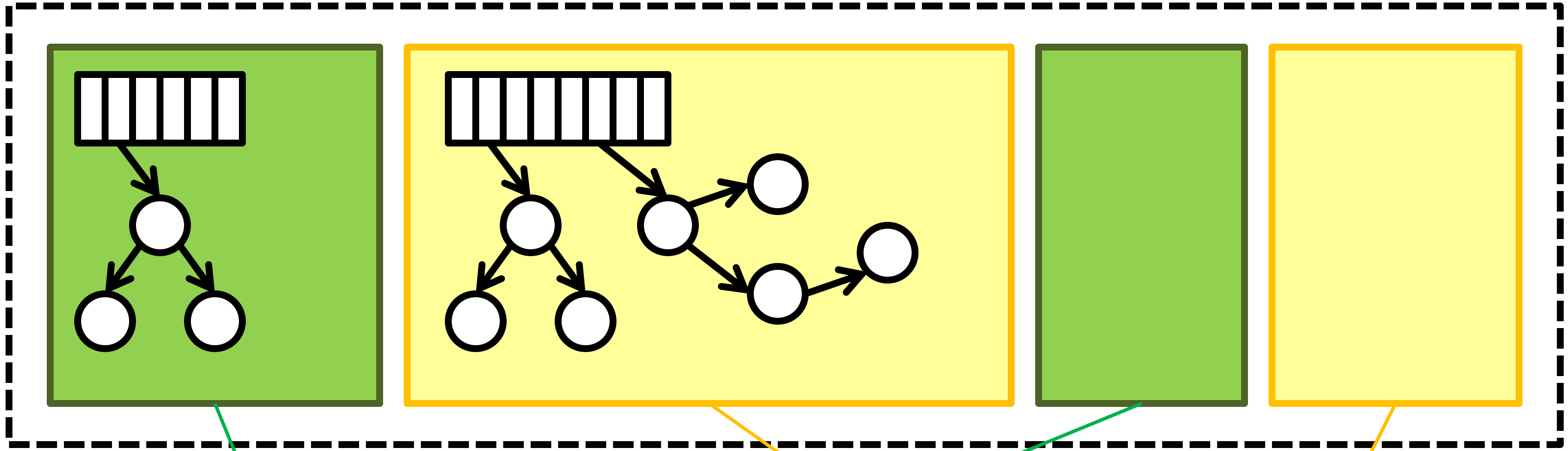


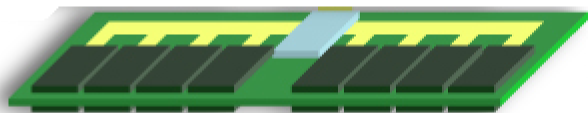


DRAM **SCM**

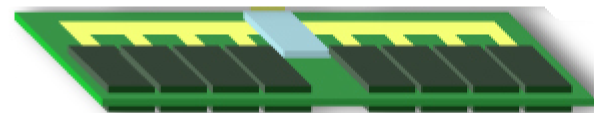
Persistent Memory

Address Space





DRAM

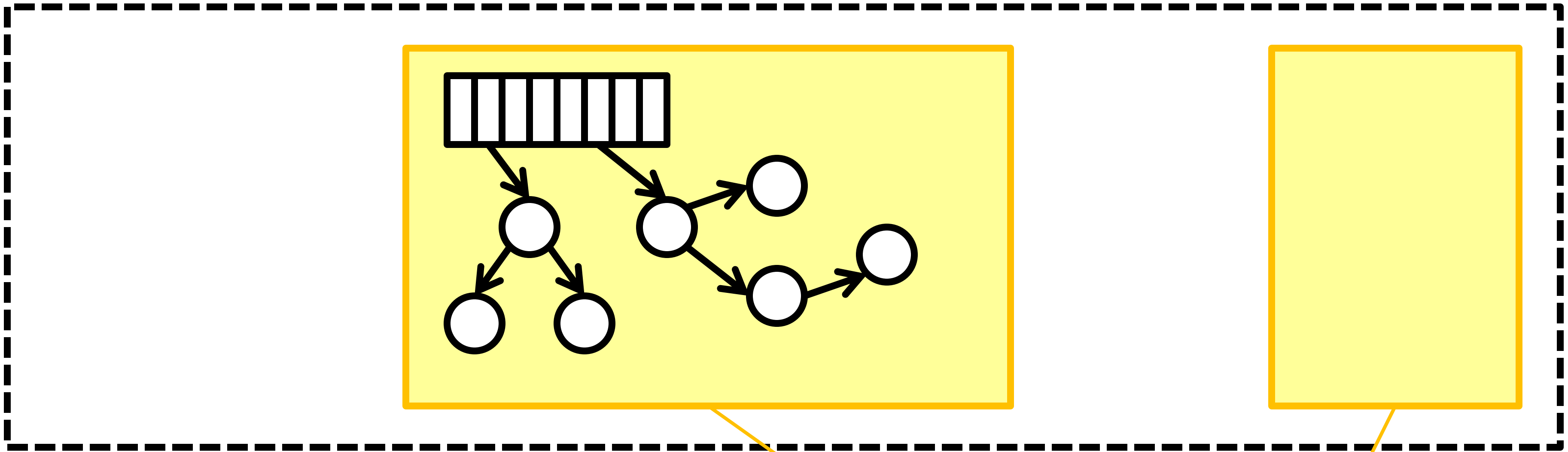


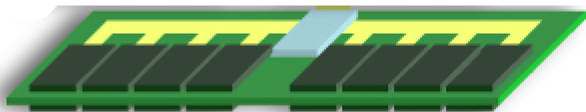
SCM

Persistent Memory

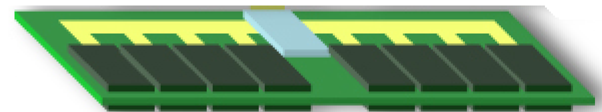
Persistent Memory

Address Space





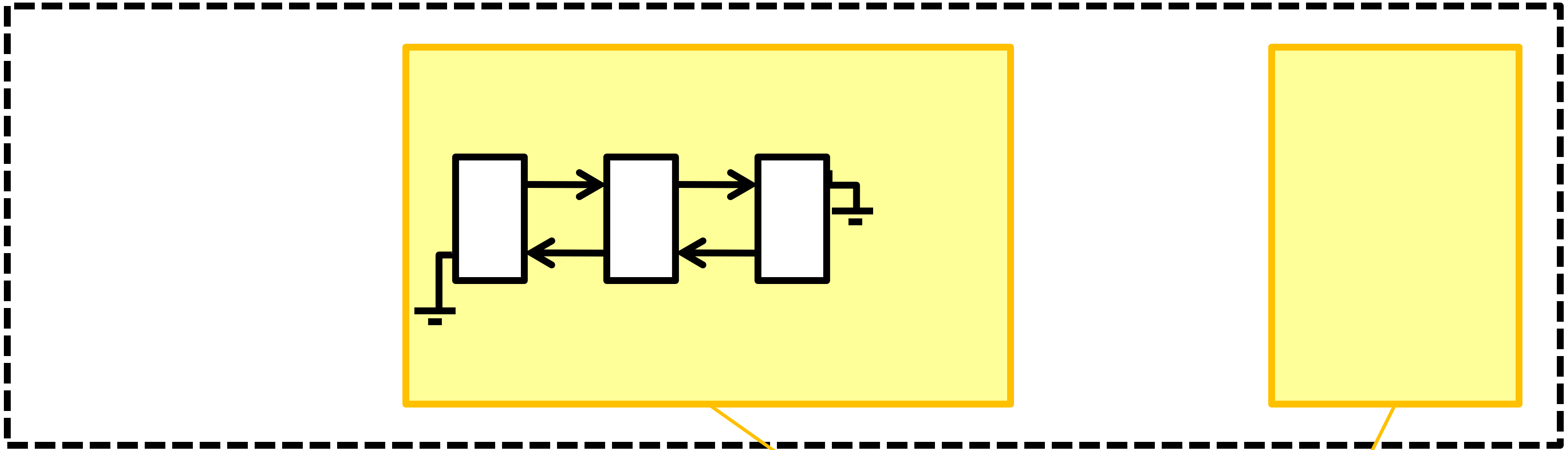
DRAM

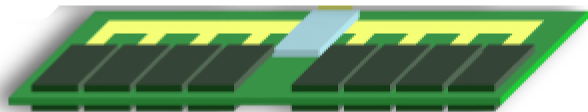


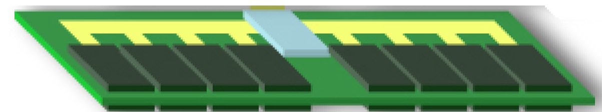
SCM

Persistent Memory

Address Space

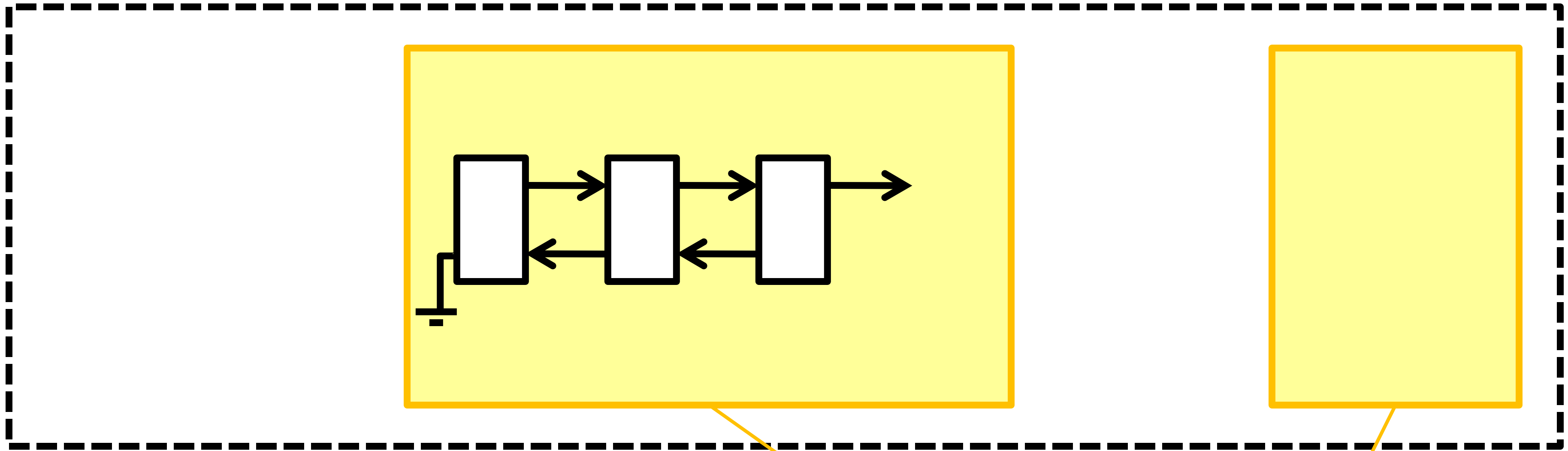


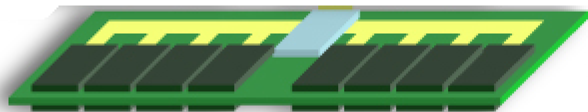




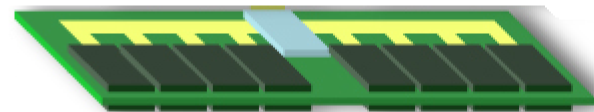
DRAM **SCM**

Persistent Memory



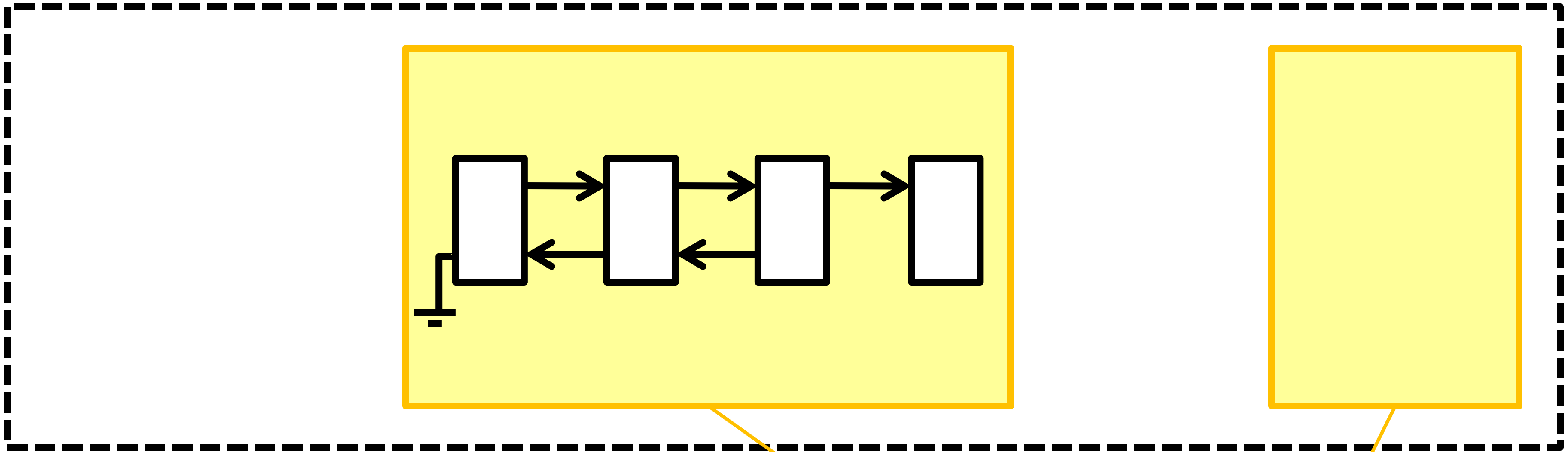


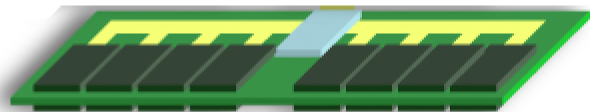
DRAM

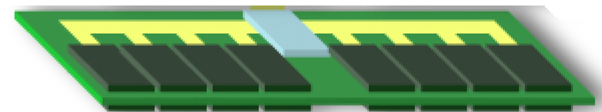


SCM

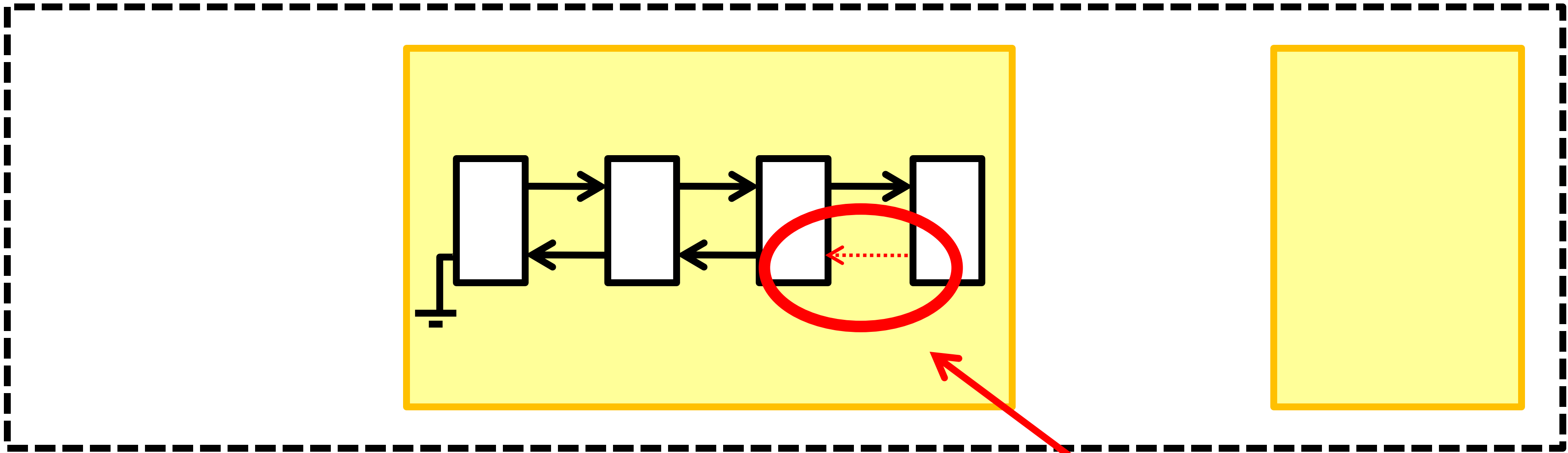
Persistent Memory







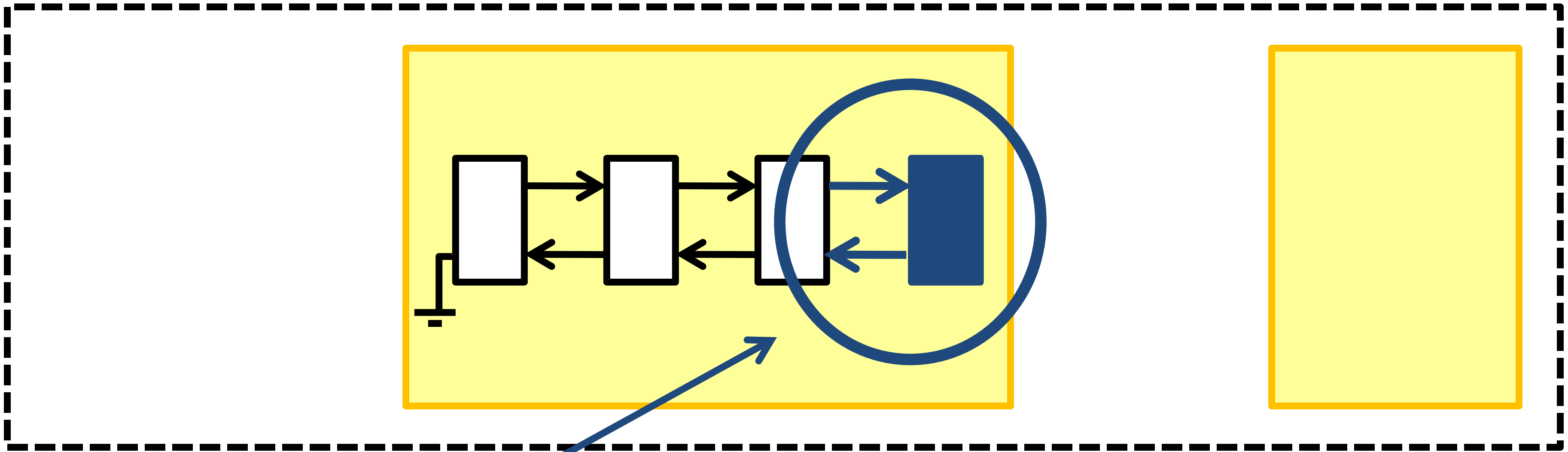
Persistent Memory



Inconsistent state

(missing pointer)

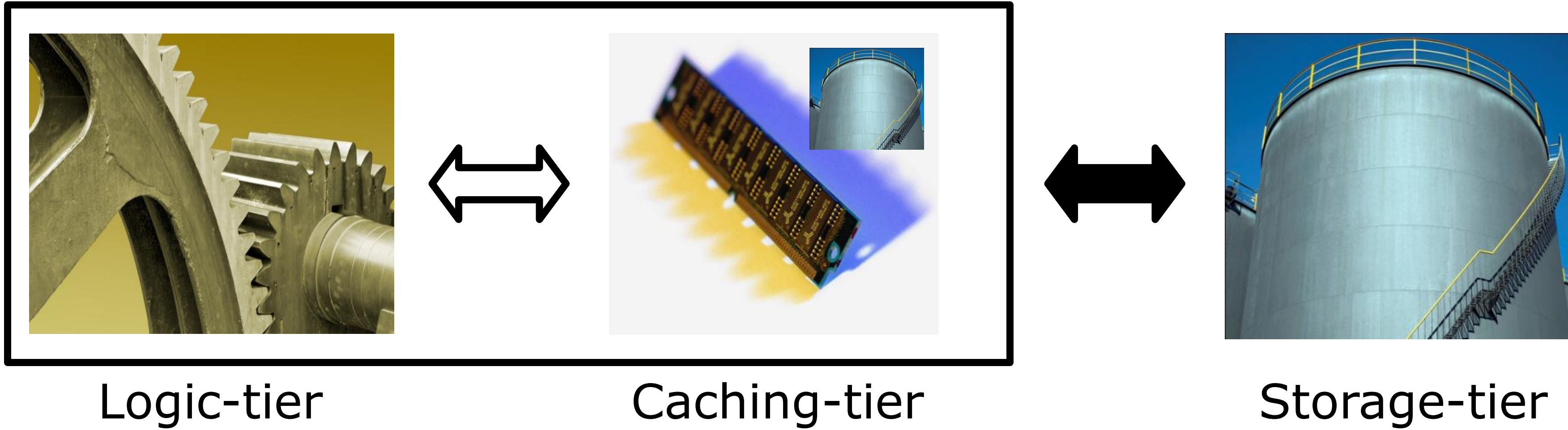
Persistent Memory



Consistent Updates
prevent inconsistent state

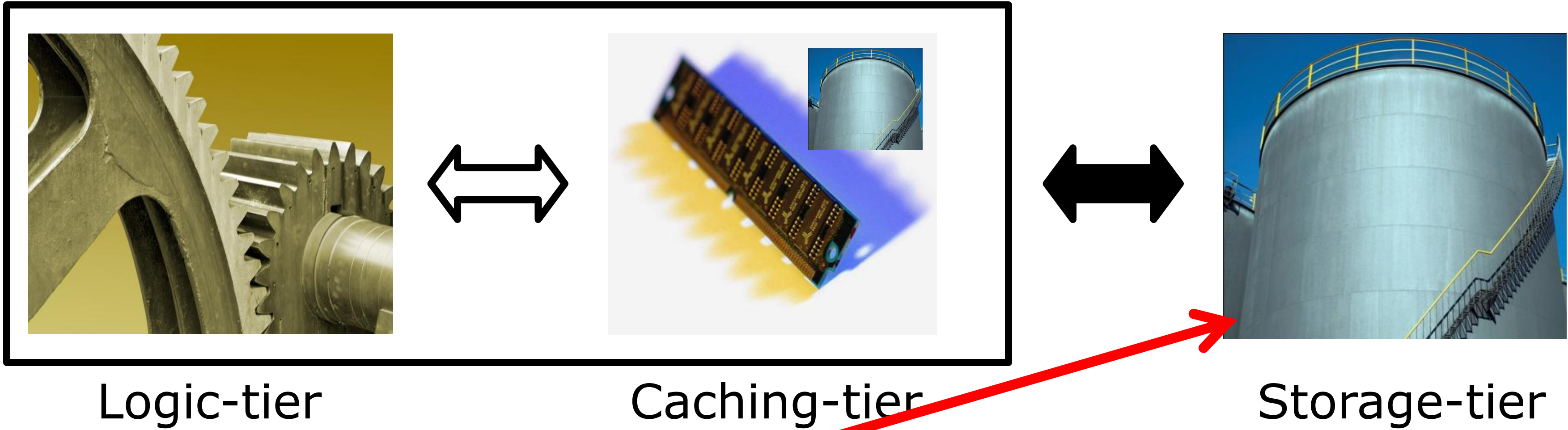
In-vivo Persistence

- In-vitro vs. In-vivo persistence



In-vivo Persistence

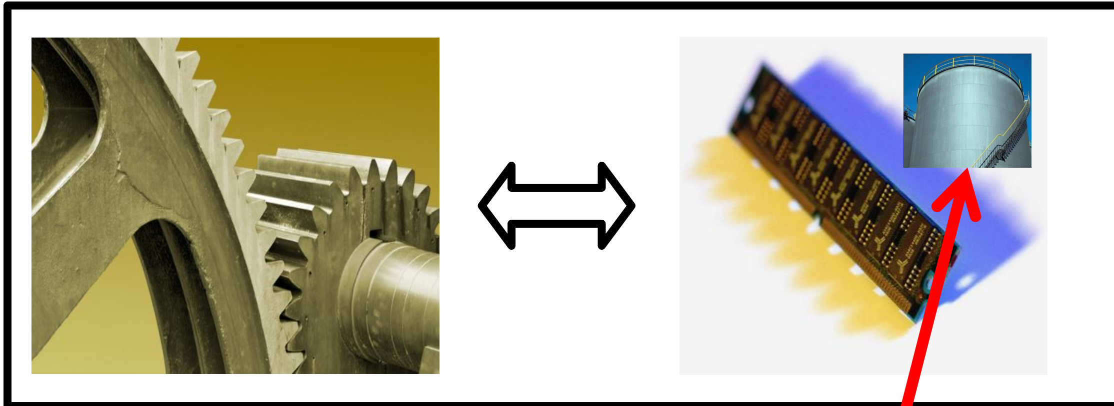
- In-vitro vs. In-vivo persistence



In-vitro persistence

In-vivo Persistence

- In-vitro vs. In-vivo persistence

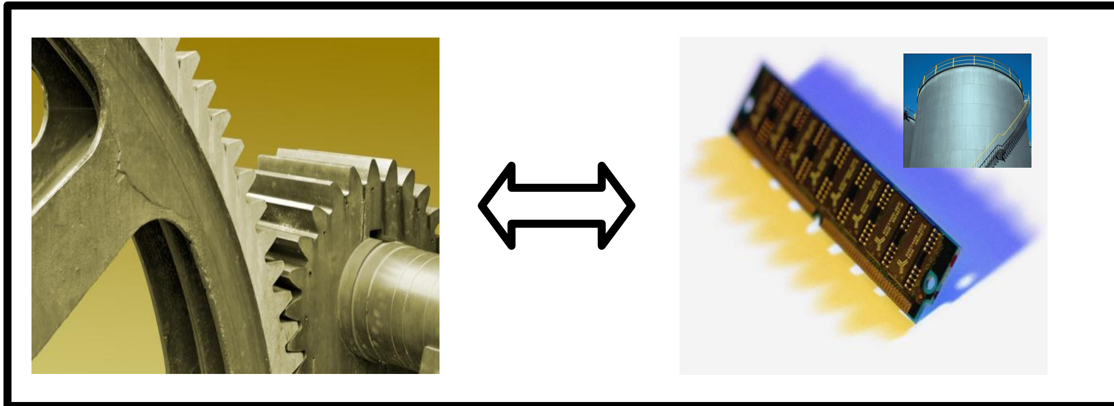


Logic-tier Caching-tier

In-vivo persistence

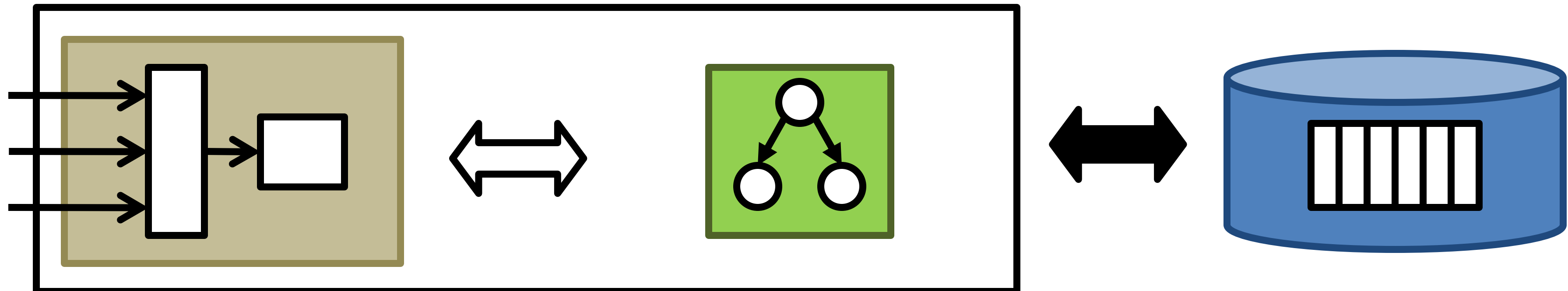
In-vivo Persistence

- In-vitro vs. In-vivo persistence



Logic-tier Caching-tier

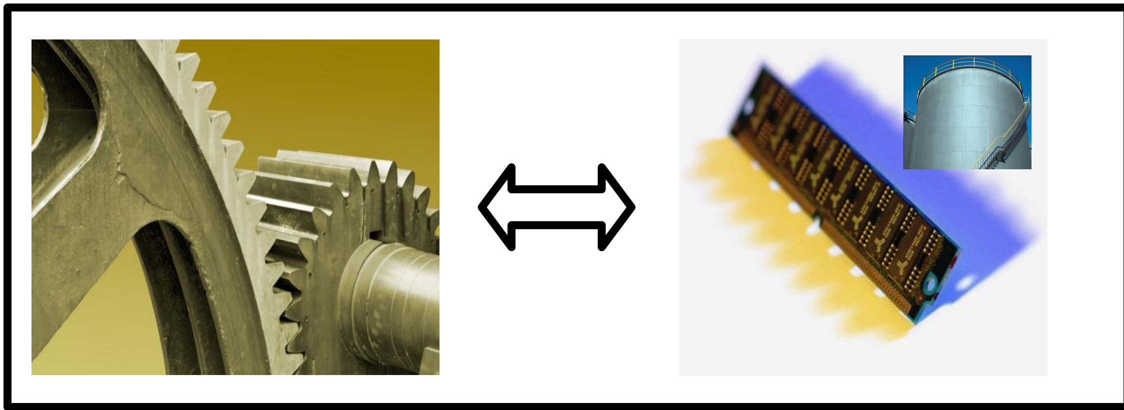
- Example: OpenLDAP



Handle Request In-memory Directory Berkeley DB

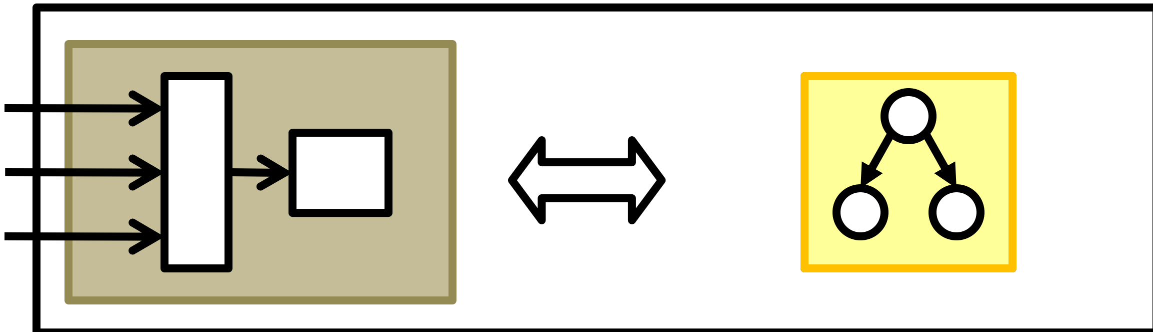
In-vivo Persistence

- In-vitro vs. In-vivo persistence



Logic-tier | Caching-tier

- Example: OpenLDAP



Handle Request | In-memory Directory

Outline

- Motivation
- **Persistent Memory**
  - **Persistent regions**
  - **Consistent updates**
- Durable Memory Transactions
- Evaluation

Persistent Regions

- Virtual memory segments stored in SCM
  - Enable programming flexibility
  - Persist across restarts
- Two region types

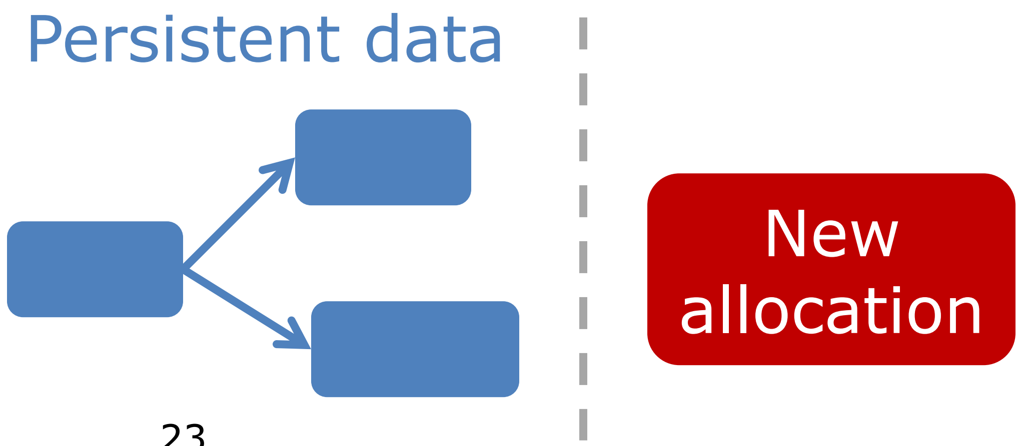
Persistent region type	API
Static	`pstatic` *`var`*
Dynamic (persistent heap)	`pmalloc`

Persistent Regions

- Virtual memory segments stored in SCM
  - Enable programming flexibility
  - Persist across restarts

➢ Two region types

Persistent region type	API
Static	`pstatic` *`var`*
Dynamic (persistent heap)	`pmalloc`



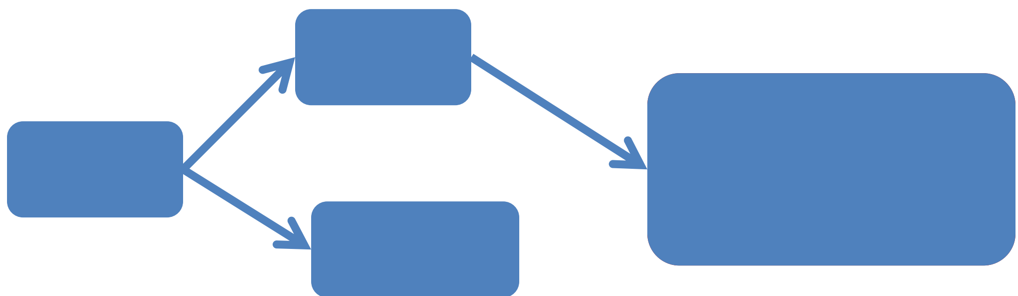
Persistent Regions

- Virtual memory segments stored in SCM
  - Enable programming flexibility
  - Persist across restarts

➢ Two region types

Persistent region type	API
Static	`pstatic` *`var`*
Dynamic (persistent heap)	`pmalloc`

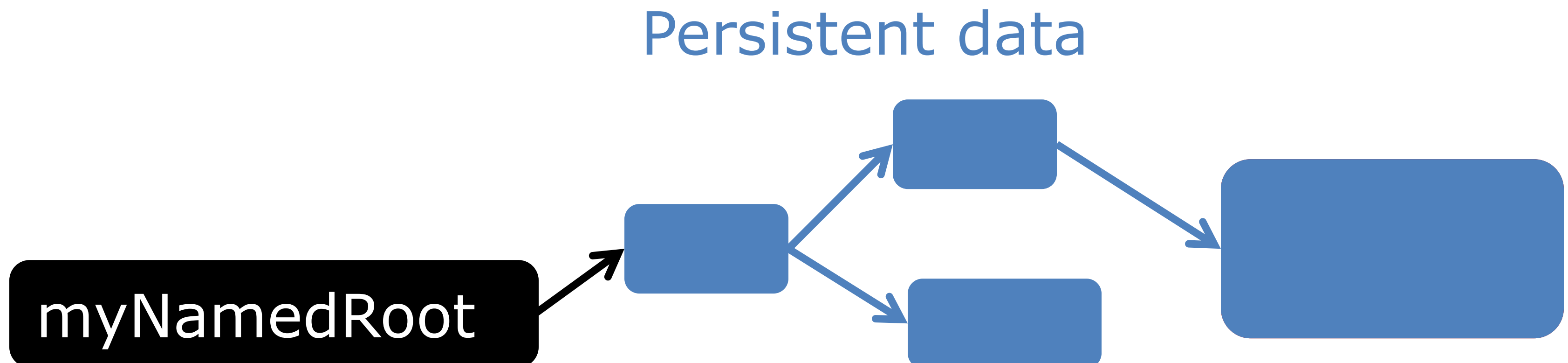
Persistent data



Persistent Regions

- Virtual memory segments stored in SCM
  - Enable programming flexibility
  - Persist across restarts
- Two region types

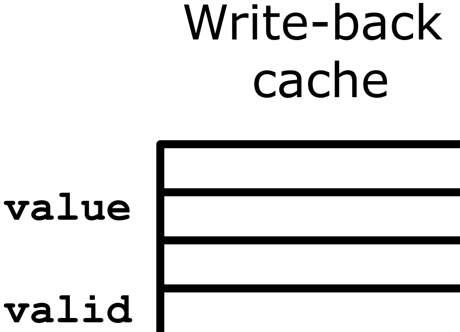
Persistent region type	API
Static	`pstatic` *`var`*
Dynamic (persistent heap)	`pmalloc`

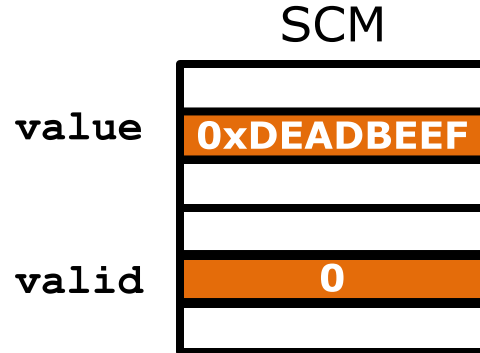


Consistent Updates

- Support updating data without risking correctness after a failure

```
R.value = 0xC0FFEE
R.valid = 1
```

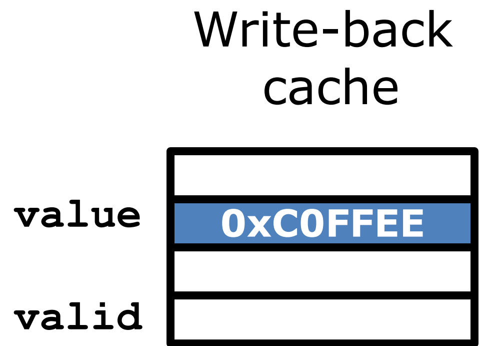


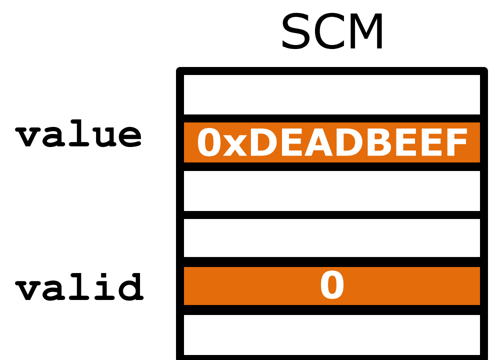


Consistent Updates

- Support updating data without risking correctness after a failure

```
R.value = 0xC0FFEE
R.valid = 1
```

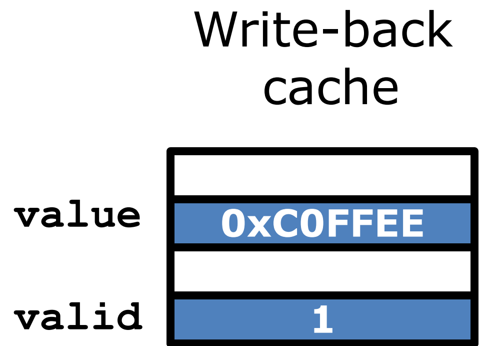


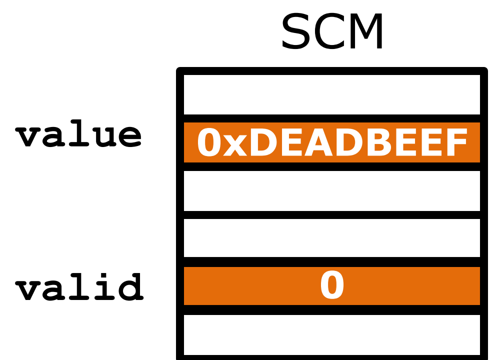


Consistent Updates

- Support updating data without risking correctness after a failure

```
R.value = 0xC0FFEE
R.valid = 1
```

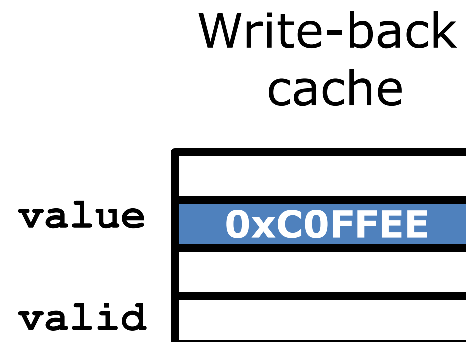


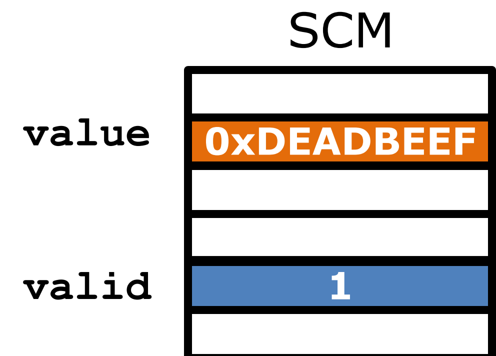


Consistent Updates

- Support updating data without risking correctness after a failure

```
R.value = 0xC0FFEE
R.valid = 1
```

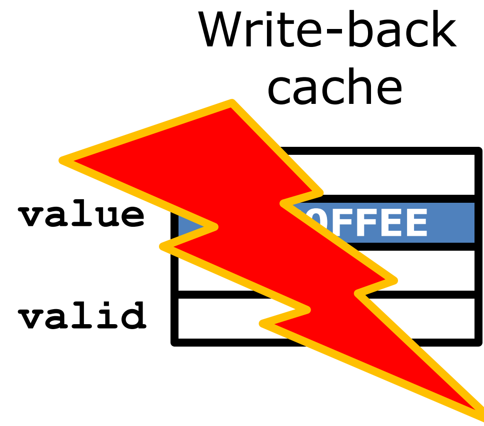


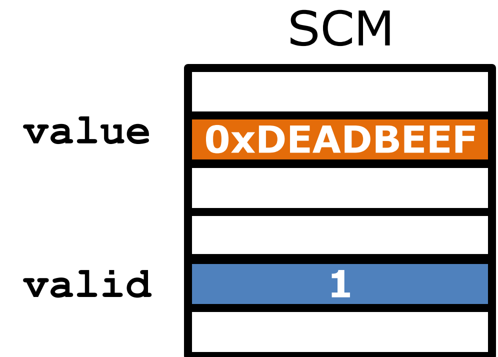


Consistent Updates

- Support updating data without risking correctness after a failure

```
R.value = 0xC0FFEE
R.valid = 1
```

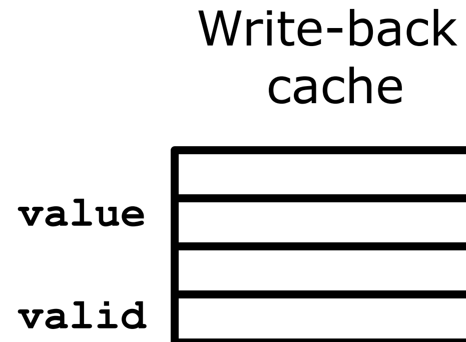


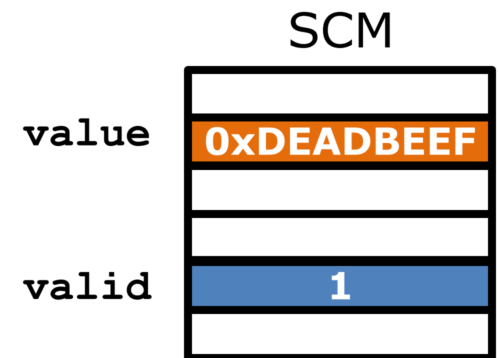


Consistent Updates

- Support updating data without risking correctness after a failure

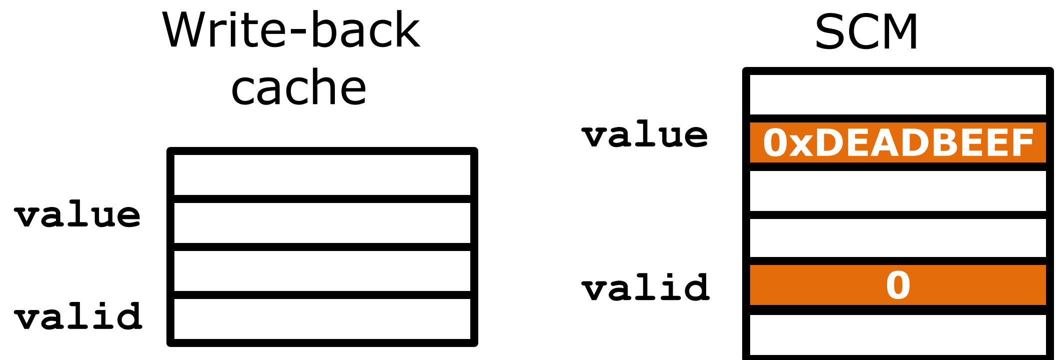
```
R.value = 0xC0FFEE
R.valid = 1
```





Consistent Updates

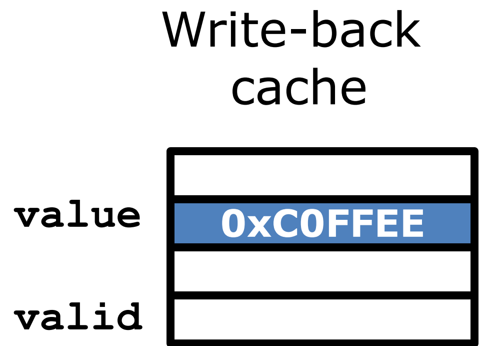
- Support updating data without risking correctness after a failure
- Rely on conventional CPU primitives for ordering
  - Flush
  - Fence

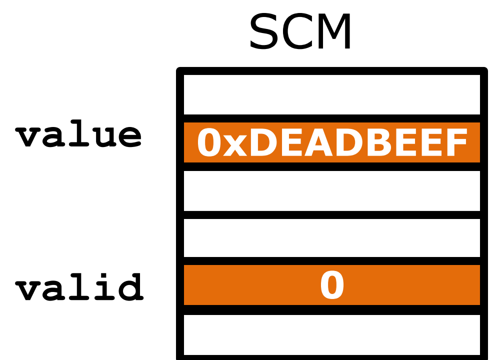


Consistent Updates

- Support updating data without risking correctness after a failure
- Rely on conventional CPU primitives for ordering
  - Flush
  - Fence

`R.value ← 0xC0FFEE`

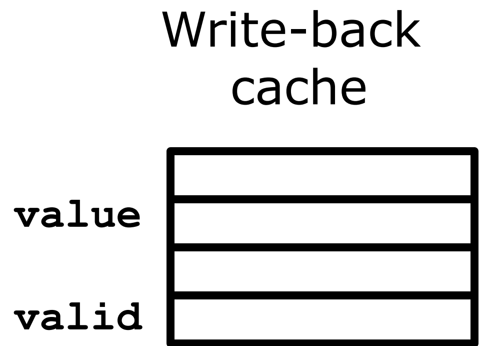


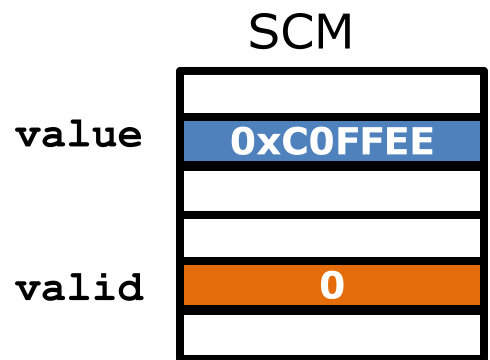


Consistent Updates

- Support updating data without risking correctness after a failure
- Rely on conventional CPU primitives for ordering
  - Flush
  - Fence

```
R.value ← 0xC0FFEE
FLUSH(&R.value)
```

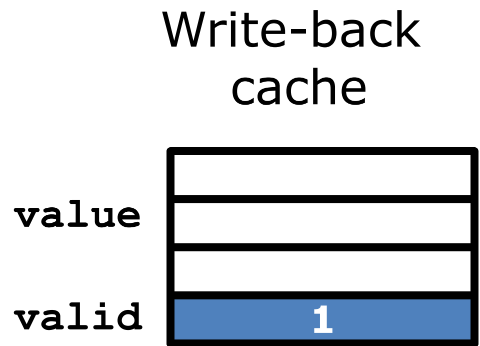


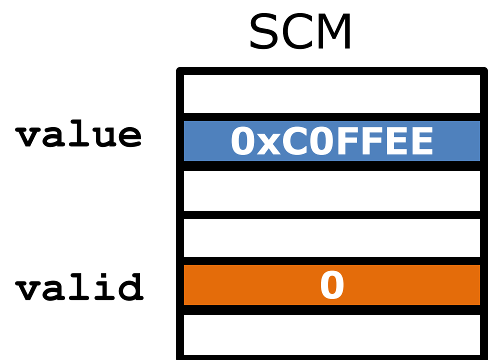


Consistent Updates

- Support updating data without risking correctness after a failure
- Rely on conventional CPU primitives for ordering
  - Flush
  - Fence

```
R.value ← 0xC0FFEE
FLUSH(&R.value)
FENCE
R.valid ← 1
```

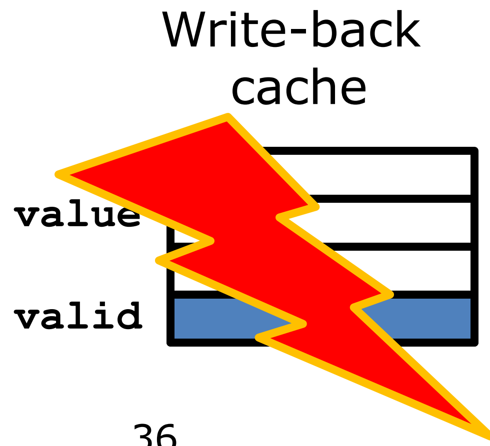


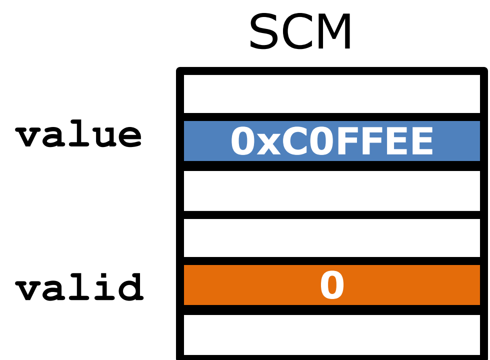


Consistent Updates

- Support updating data without risking correctness after a failure
- Rely on conventional CPU primitives for ordering
  - Flush
  - Fence

```
R.value ← 0xC0FFEE
FLUSH(&R.value)
FENCE
R.valid ← 1
```

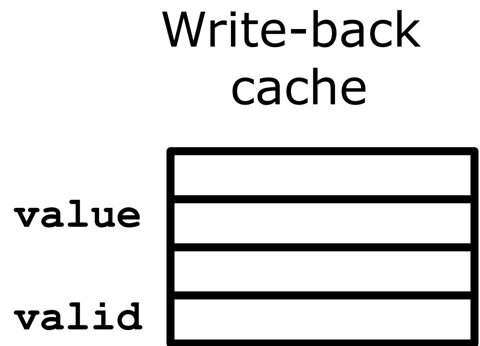


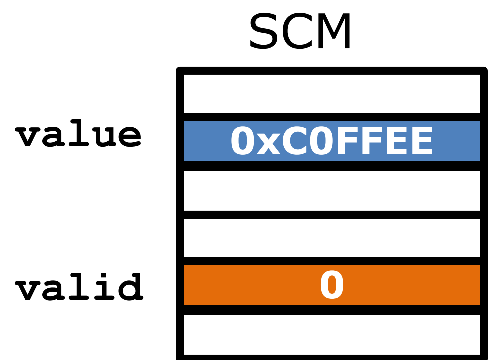


Consistent Updates

- Support updating data without risking correctness after a failure
- Rely on conventional CPU primitives for ordering
  - Flush
  - Fence

```
R.value ← 0xC0FFEE
FLUSH(&R.value)
FENCE
R.valid ← 1
```





Outline

- Motivation
- Persistent Memory
  - Persistent regions
  - Consistent updates
- **Durable Memory Transactions**
- Evaluation

Durable Memory Transactions

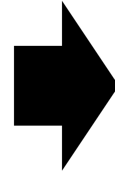
- Compiler instruments **atomic** blocks

```
atomic {
 R.value = 0xC0F;
 R.valid = 1;
}
```

Durable Memory Transactions

- Compiler instruments **atomic** blocks

```
atomic {
 R.value = 0xC0F;
 R.valid = 1;
}
```



```
begin_transaction();


commit_transaction();
```

Durable Memory Transactions

- Compiler instruments **atomic** blocks

```
atomic {
 R.value = 0xC0F;
 R.valid = 1;
}
```

```
begin_transaction();
stm_store(&R.value, 0xC0F);
stm_store(&R.valid, 1);
commit_transaction();
```

Durable Memory Transactions

- Compiler instruments **atomic** blocks

```
atomic {
 R.value = 0xC0F;
 R.valid = 1;
}
```

⇒

```
begin_transaction();
stm_store(&R.value, 0xC0F);
stm_store(&R.valid, 1);
commit_transaction();
```

- Runtime supports ACID transactions
  - Based on TinySTM

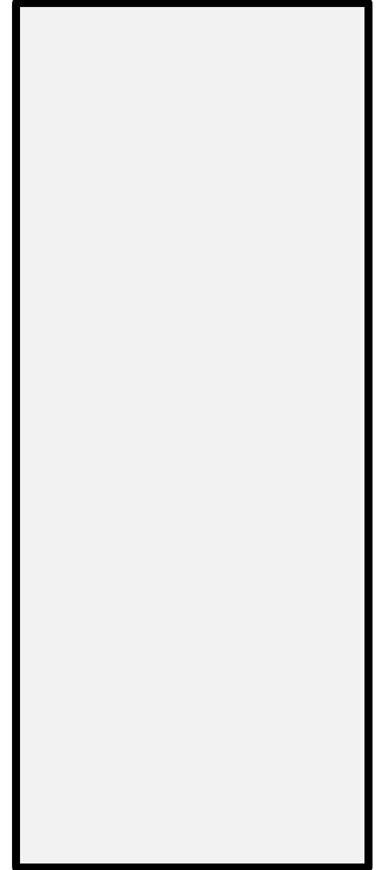
Hash Table Example

```
pstatic htRoot = NULL;
```

htRoot

0000000000

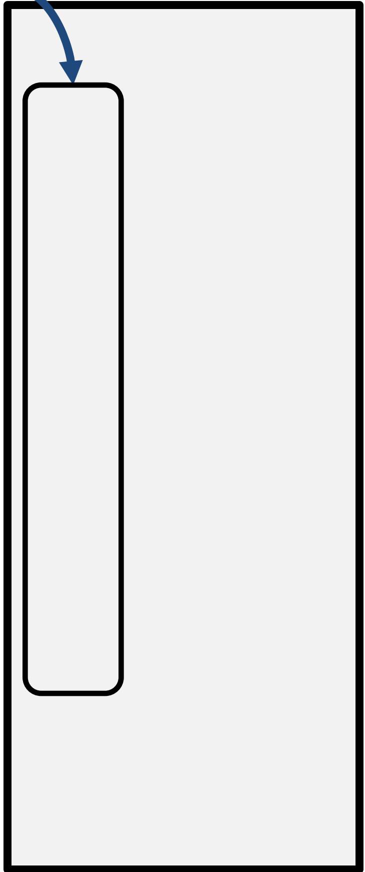
Static



Persistent Heap

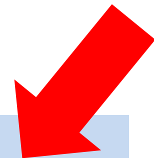
Hash Table Example

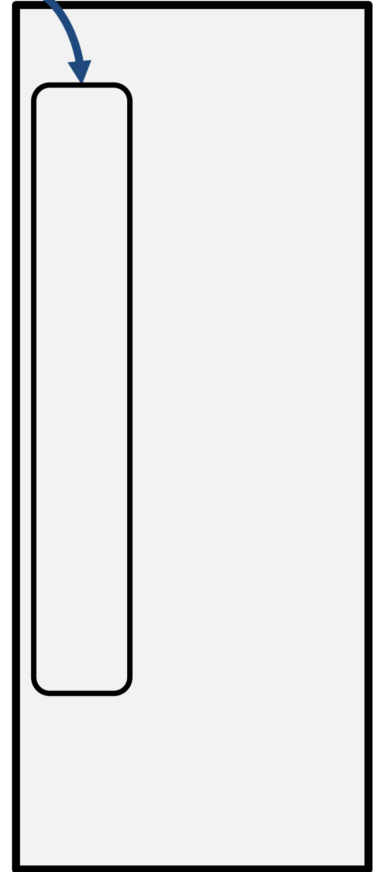
```
pstatic htRoot = NULL;

main() {
   if (!htRoot)
     pmalloc(N*sizeof(*bucket), &htRoot);
}
```



Hash Table Example

```
pstatic htRoot = NULL;

main() {
   if (!htRoot)
      pmalloc(N*sizeof(*bucket), &htRoot);
}
```

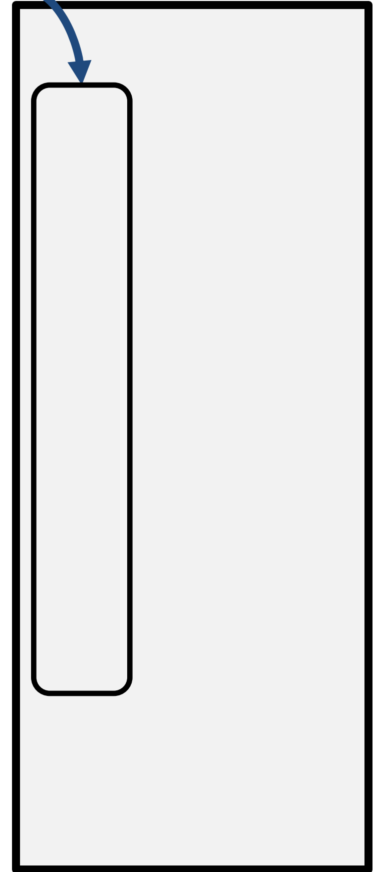




Hash Table Example

```
pstatic htRoot = NULL;

main() {
   if (!htRoot)
     pmalloc(N*sizeof(*bucket), &htRoot);
}


update_hash(key, value) {
  atomic {
    pmalloc(sizeof(*bucket), &bucket));
    bucket->key = key;
    bucket->value = value;
    insert(hash, bucket);
  }
}
```

Static

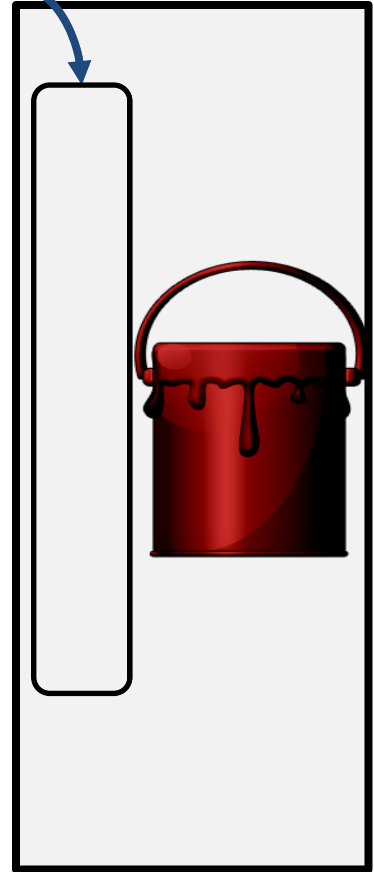


Persistent Heap

Hash Table Example

```
pstatic htRoot = NULL;

main() {
   if (!htRoot)
     pmalloc(N*sizeof(*bucket), &htRoot);
}



update_hash(key, value) {
  atomic {
    pmalloc(sizeof(*bucket), &bucket));
    bucket->key = key;
    bucket->value = value;
    insert(hash, bucket);
  }
}
```

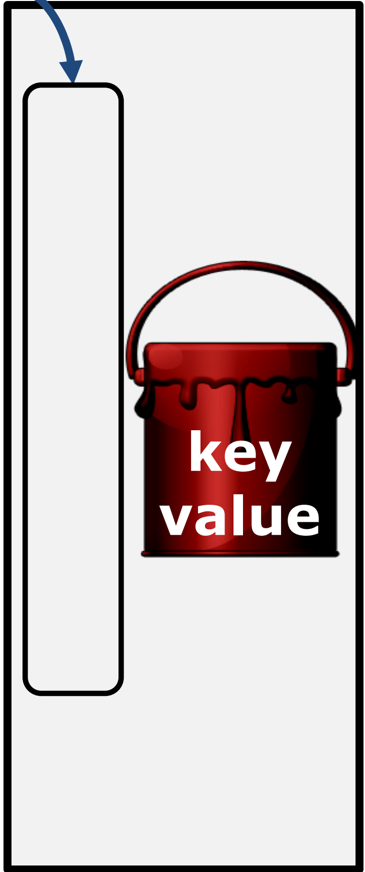
Static



Persistent Heap

Hash Table Example

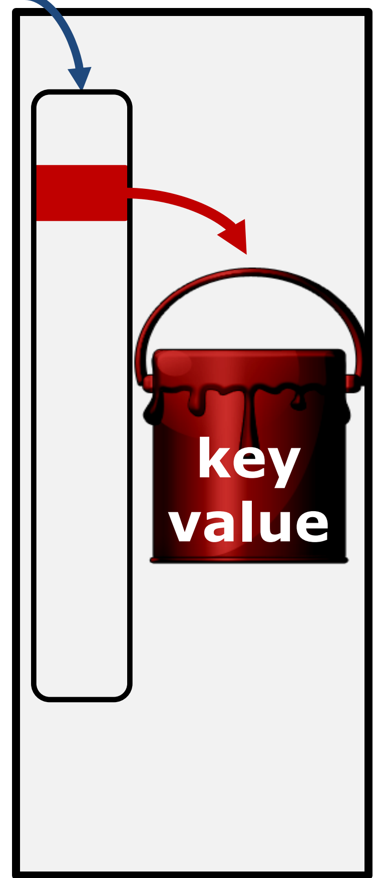
```
pstatic htRoot = NULL;

main() {
  if (!htRoot)
    pmalloc(N*sizeof(*bucket), &htRoot);
}


update_hash(key, value) {
  atomic {
    pmalloc(sizeof(*bucket), &bucket));
    bucket->key = key;
    bucket->value = value;
    insert(hash, bucket);
  }
}
```



Hash Table Example

```
pstatic htRoot = NULL;

main() {
  if (!htRoot)
    pmalloc(N*sizeof(*bucket), &htRoot);
}


update_hash(key, value) {
  atomic {
    pmalloc(sizeof(*bucket), &bucket));
    bucket->key = key;
    bucket->value = value;
    insert(hash, bucket);
  }
}
```

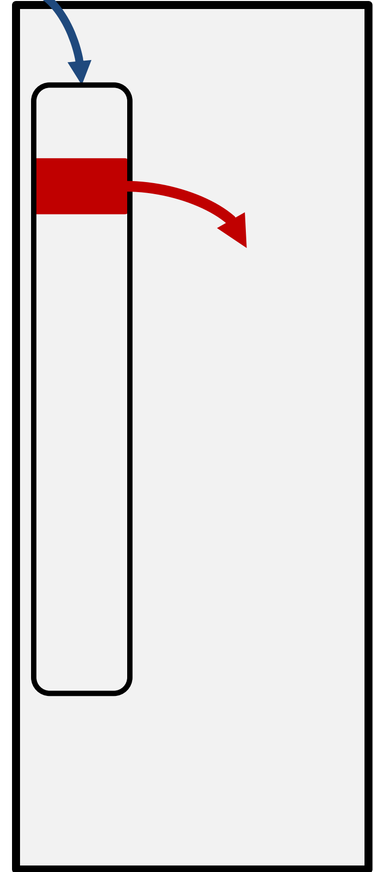
Static



Persistent Heap

Hash Table Example

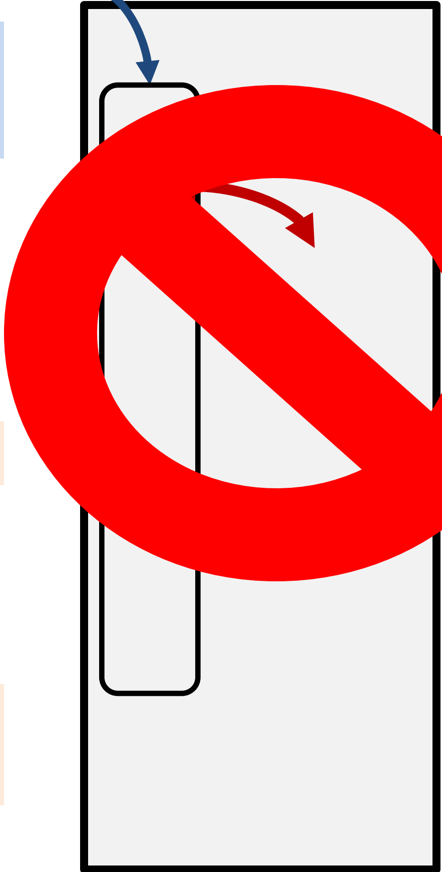
```
pstatic htRoot = NULL;

main() {
  if (!htRoot)
    pmalloc(N*sizeof(*bucket), &htRoot);
}


update_hash(key, value) {
  atomic {
    pmalloc(sizeof(*bucket), &bucket));
    bucket->key = key;
    bucket->value = value;
    insert(hash, bucket);
  }
}
```



Hash Table Example

```
pstatic htRoot = NULL;

main() {
  if (!htRoot)
    pmalloc(N*sizeof(*bucket), &htRoot);
}


update_hash(key, value) {
  atomic {
    pmalloc(sizeof(*bucket), &bucket));
    bucket->key = key;
    bucket->value = value;
    insert(hash, bucket);
  }
}
```

Static



Heap

Outline

- Motivation
- Persistent Memory
  - Persistent regions
  - Consistent updates
- Durable Memory Transactions
- **Evaluation**

Evaluation

- Is it easy to use?
  - Applications
- Is it fast?
  - Microbenchmarks
  - Applications

Applications

- TokyoCabinet: B-tree based key-value store
  - Original: syncs B-tree to a memory-mapped file
  - Modified: keeps B-tree in persistent memory
- OpenLDAP: Lightweight Directory
  - Original: stores dir-entries in Berkeley DB
  - Modified: keeps dir-entries in persistent memory

Performance Evaluation

- Platform
  - Intel Core 2 2.5GHz (4 cores)
  - x86-64 Linux 2.6.33
  - Intel STM C/C++ compiler
- Performance Model
  - Writes: DRAM + extra fixed delay (150ns)
  - Reads: DRAM
- Configurations
  - PCM-disk: ext2fs + RAM-disk
  - Mnemosyne: Persistent memory

Microbenchmark – Hash Table

Berkeley DB (PCM-disk) **vs Mnemosyne**

Microbenchmark – Hash Table

Berkeley DB (PCM-disk) **vs Mnemosyne**

Microbenchmark – Hash Table

Berkeley DB (PCM-disk) **vs Mnemosyne**

BDB-1 thread
BDB-2 threads
BDB-4 threads
MNE-1 thread
MNE-2 threads
MNE-4 threads

Latency (µs)

200
150
100
50
0

Update Throughput
(10^3 ops/s)

1200
800
400
0

8
256
2048

value size (bytes)

Microbenchmark – Hash Table

Berkeley DB (PCM-disk) vs Mnemosyne

Application Performance

- Write-mostly workloads
  - TokyoCabinet: 1024-byte ins/del queries
  - OpenLDAP: template-based update queries

Conclusion: Mnemosyne

- Exposes SCM directly to programmers as persistent memory
- Relies on conventional CPU primitives for ordering
- Enables low-latency, consistent, in-place updates via durable memory transactions