HICAMP:

Architectural Support for Efficient Concurrency-Safe Shared Structured Data Access

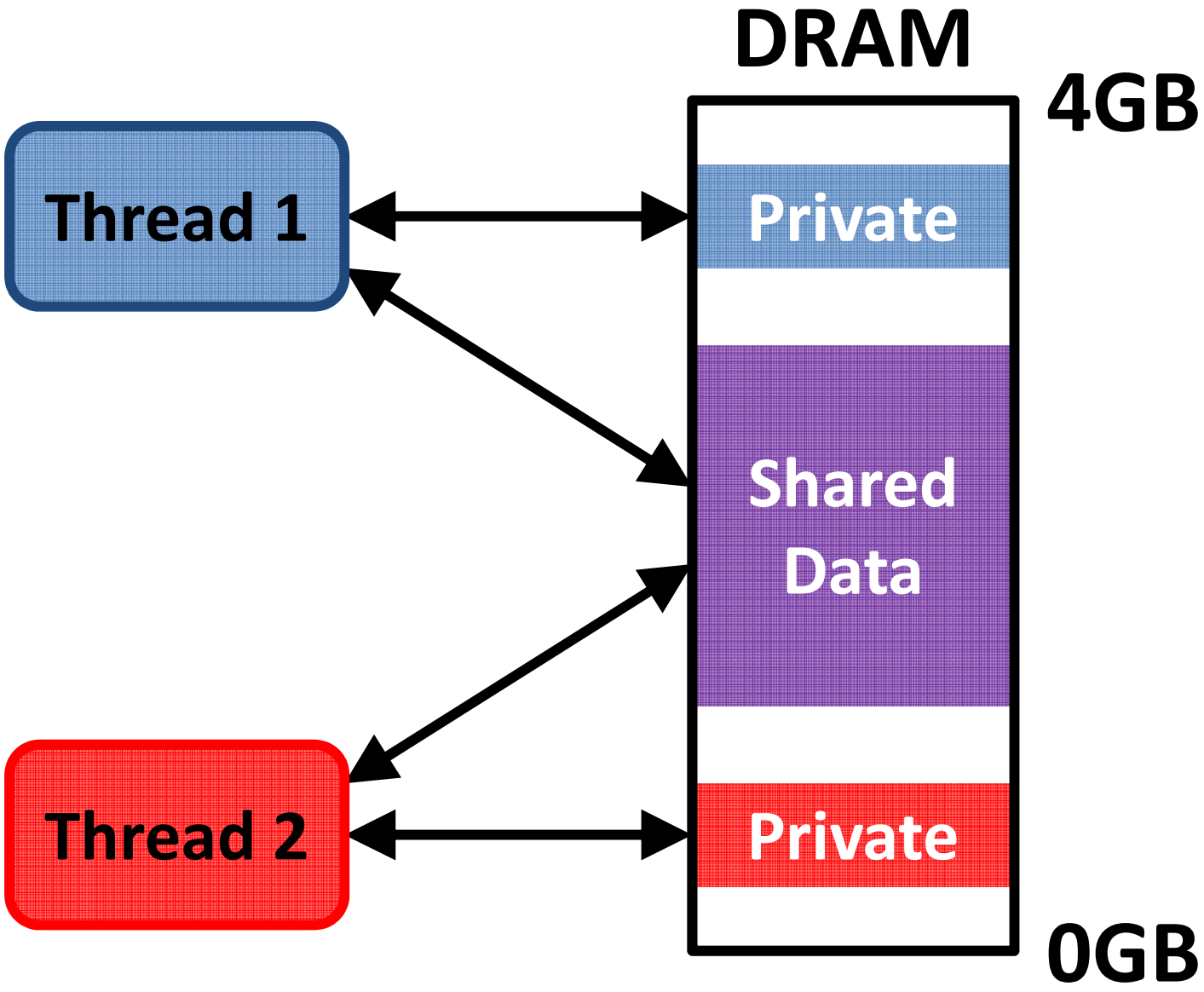
Cheriton et al., ASPLOS 2012

Yoongu Kim

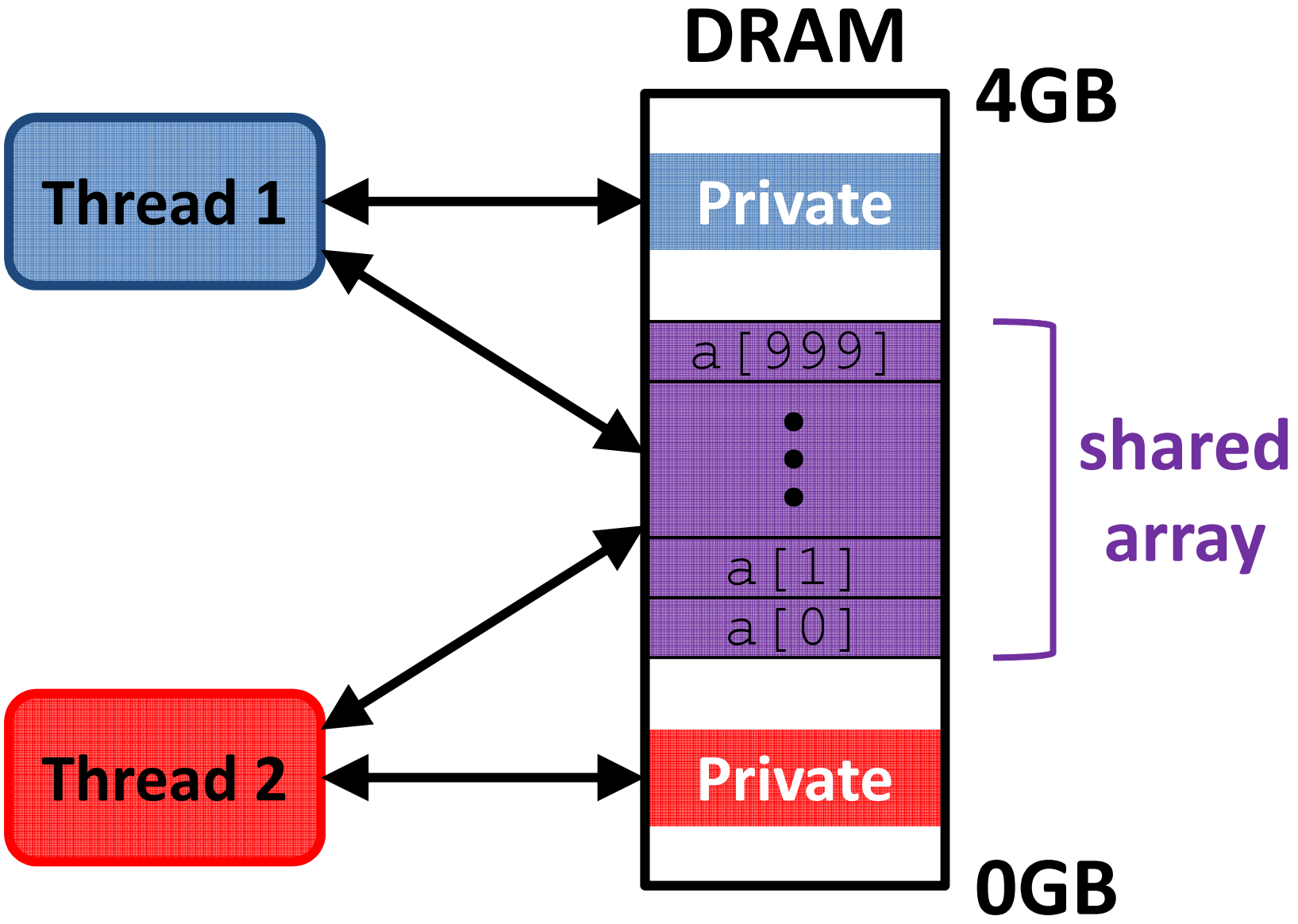
11/18/2013

INTRODUCTION

Intro: Shared Data



Intro: Shared Data

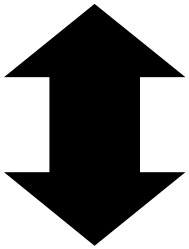


Problem: Concurrent Accesses

Thread 1

```
for(i=0; i<1000; i++)
    sum = sum + a[i];
```

***Read* access to shared array**



CONFLICT!

***Write* access to shared array**

Thread 2

```
a[900] = -1;
```

Traditional Solutions are Expensive

Solution #1: Lock

- Only *one* thread can access shared data ...
- ... the thread that holds the lock

- But what if shared data is *very large*?
  - Example: Bank database
  - When an auditing thread accesses the bank database, all other threads would starve
    - No deposits/withdrawals for any customer

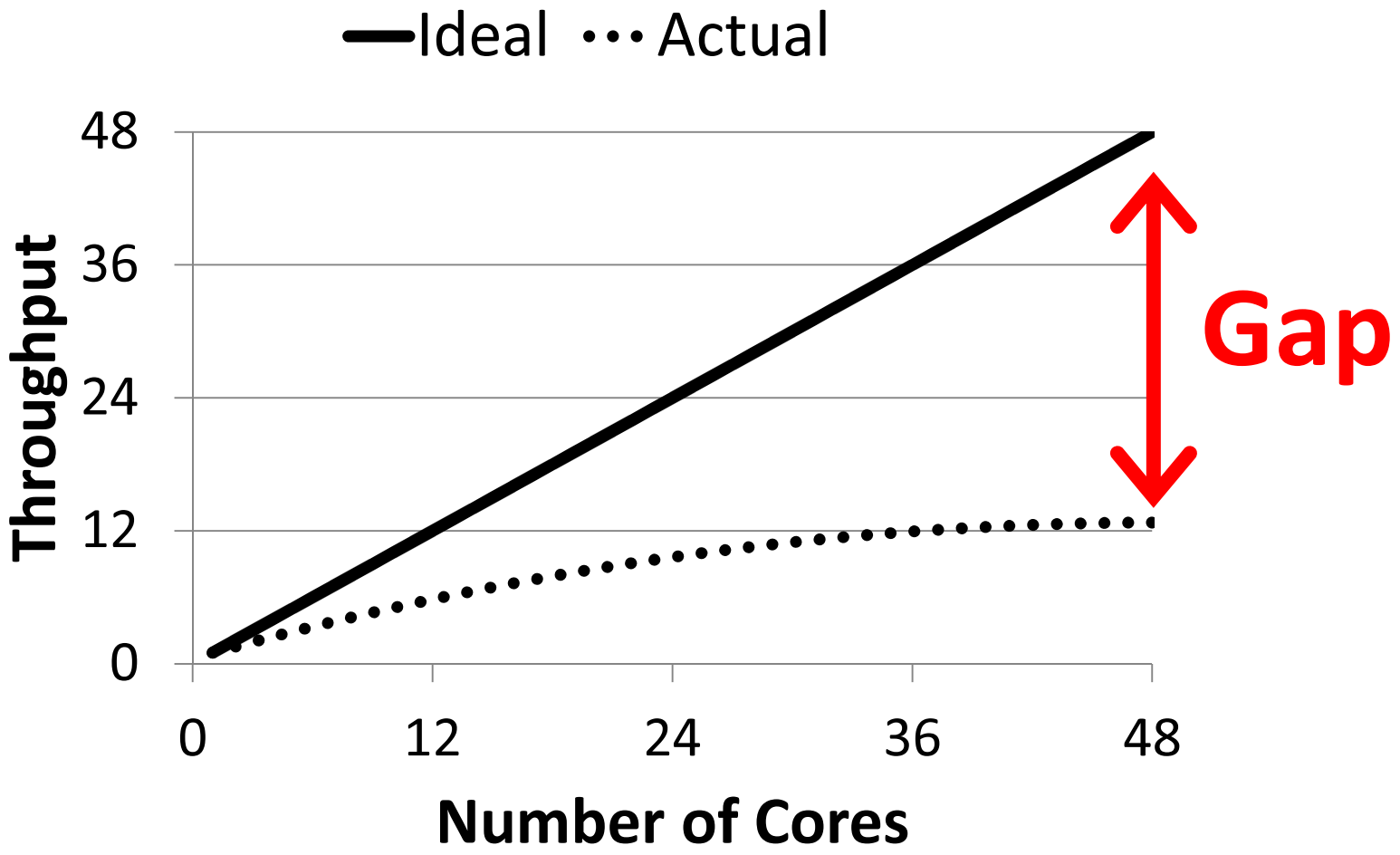
Traditional Solutions are Expensive

Solution #2: Transaction

- *Speculatively* allow multiple threads to access shared data in a concurrent manner
- If lucky ➔ no conflict
- If unlucky ➔ undo changes to shared data & retry

- But what if a transaction is *very long*?
  - 100% chance of being unlucky
  - Undoing/retrying a transaction is wasteful

Throughput vs. Number of Cores



Sharing is the root of all evil

Boyd-Wickizer et al., OSDI'10

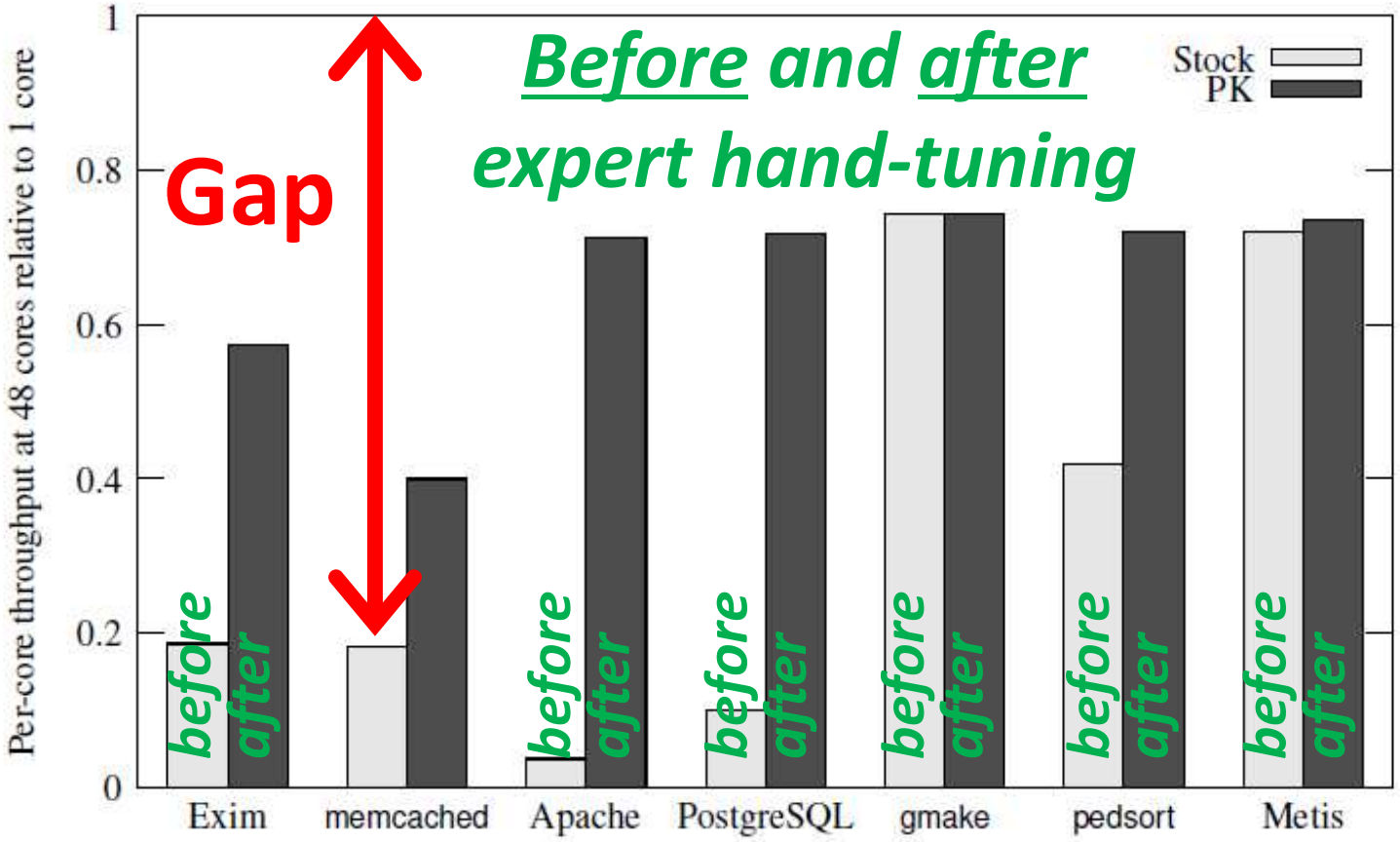
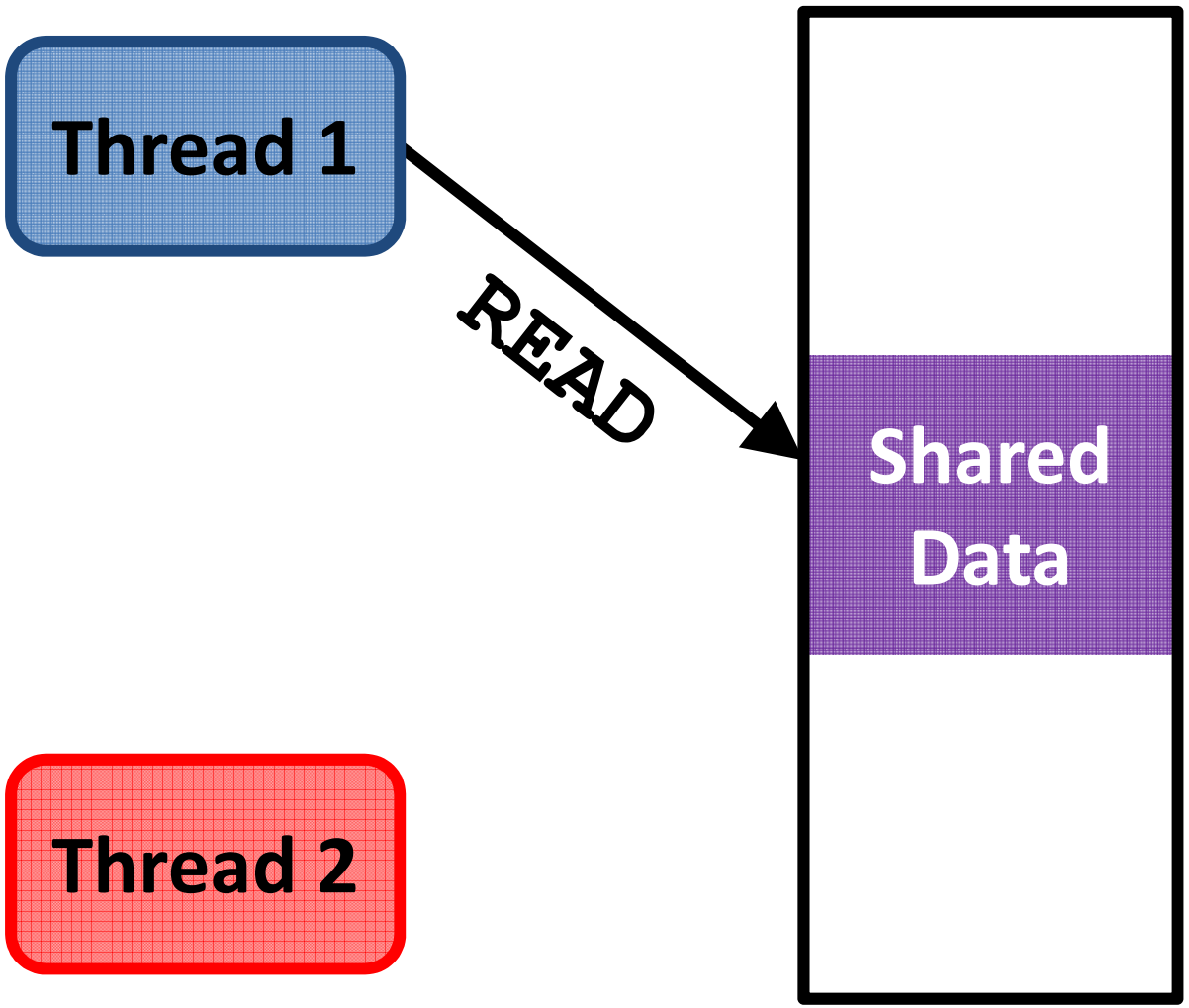
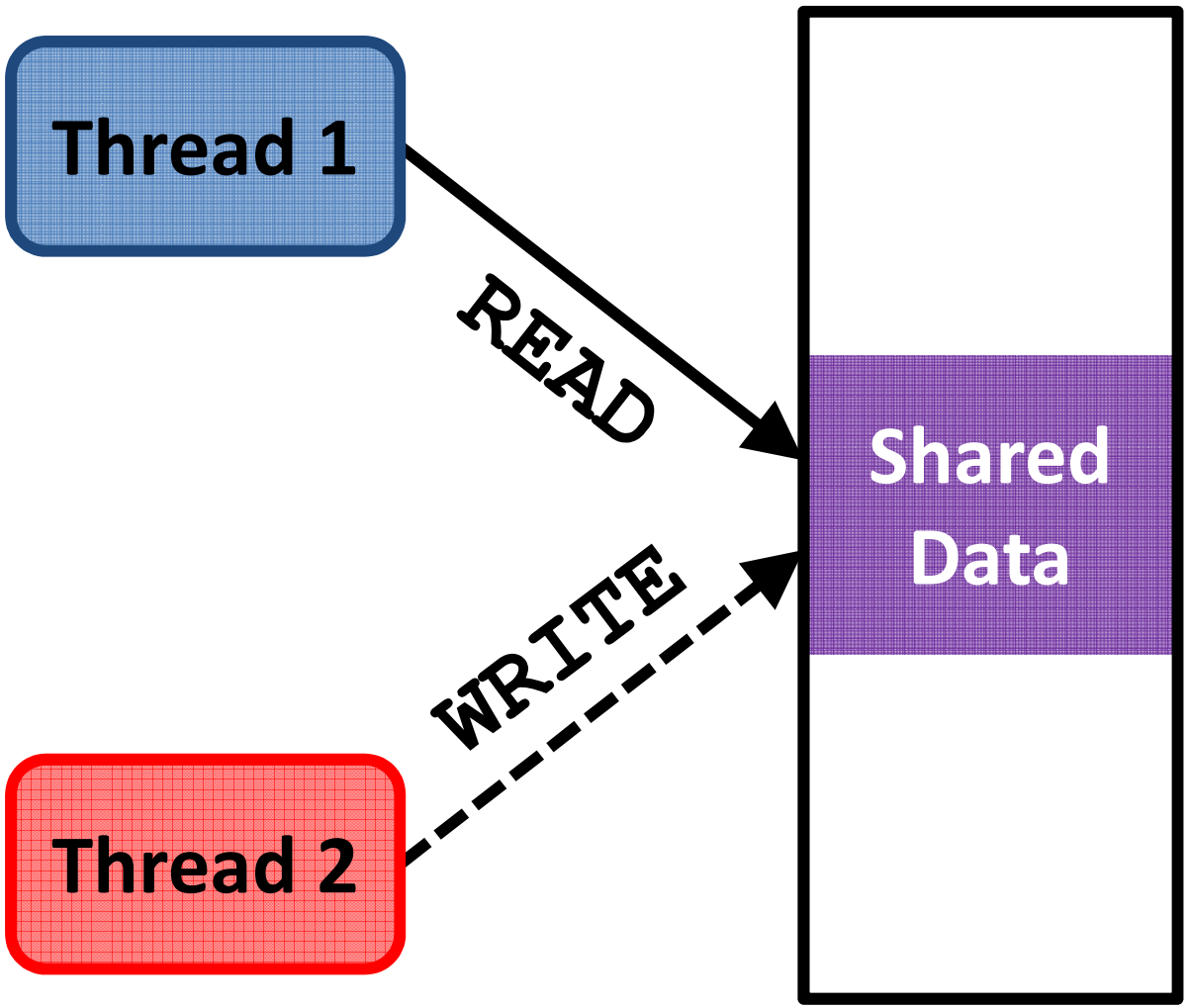


Figure 3: MOSBENCH results summary. Each bar shows the ratio of per-core throughput with 48 cores to throughput on one core, with 1.0 indicating perfect scalability. Each pair of bars corresponds to one application before and after our kernel and application modifications.

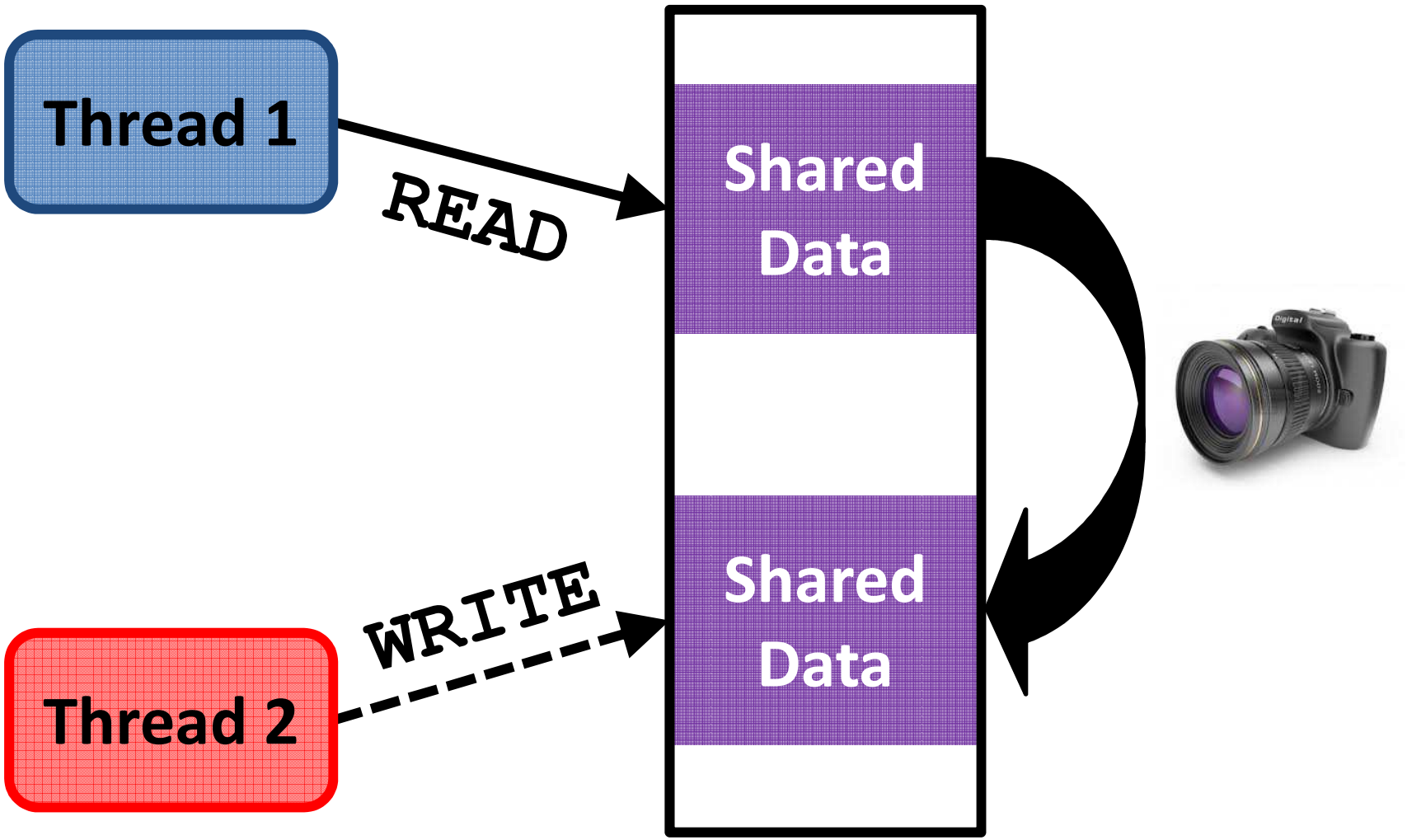
Alternative Solution: “Snapshotting”



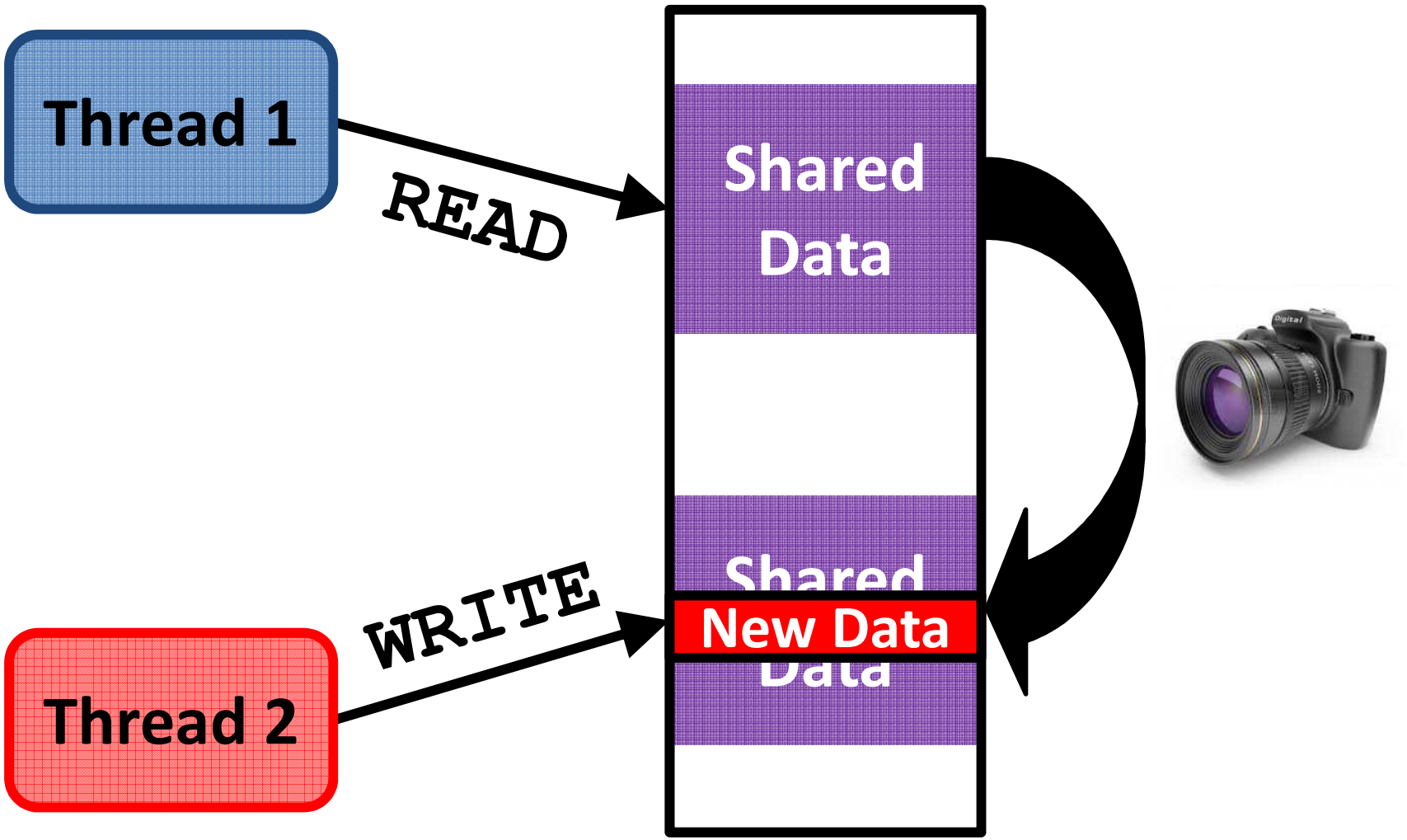
Alternative Solution: “Snapshotting”



Alternative Solution: “Snapshotting”



Alternative Solution: "Snapshotting"



Key Question

How to make memory "snapshots" cheap?

- Naïve approaches are very expensive
  1. Performance waste: copying data
  2. Capacity waste: duplicate data

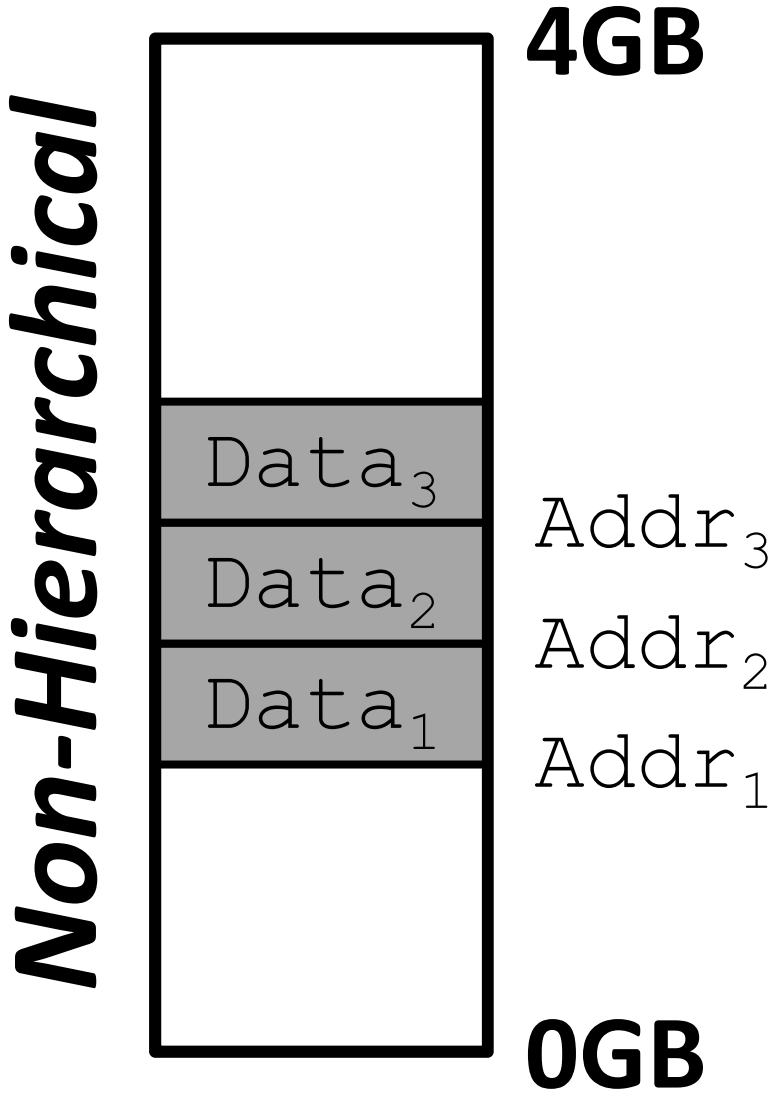
- A better approach: **HICAMP**
  - Provides hardware-support for "snapshots" while incurring only small overheads

HICAMP: THE BASICS

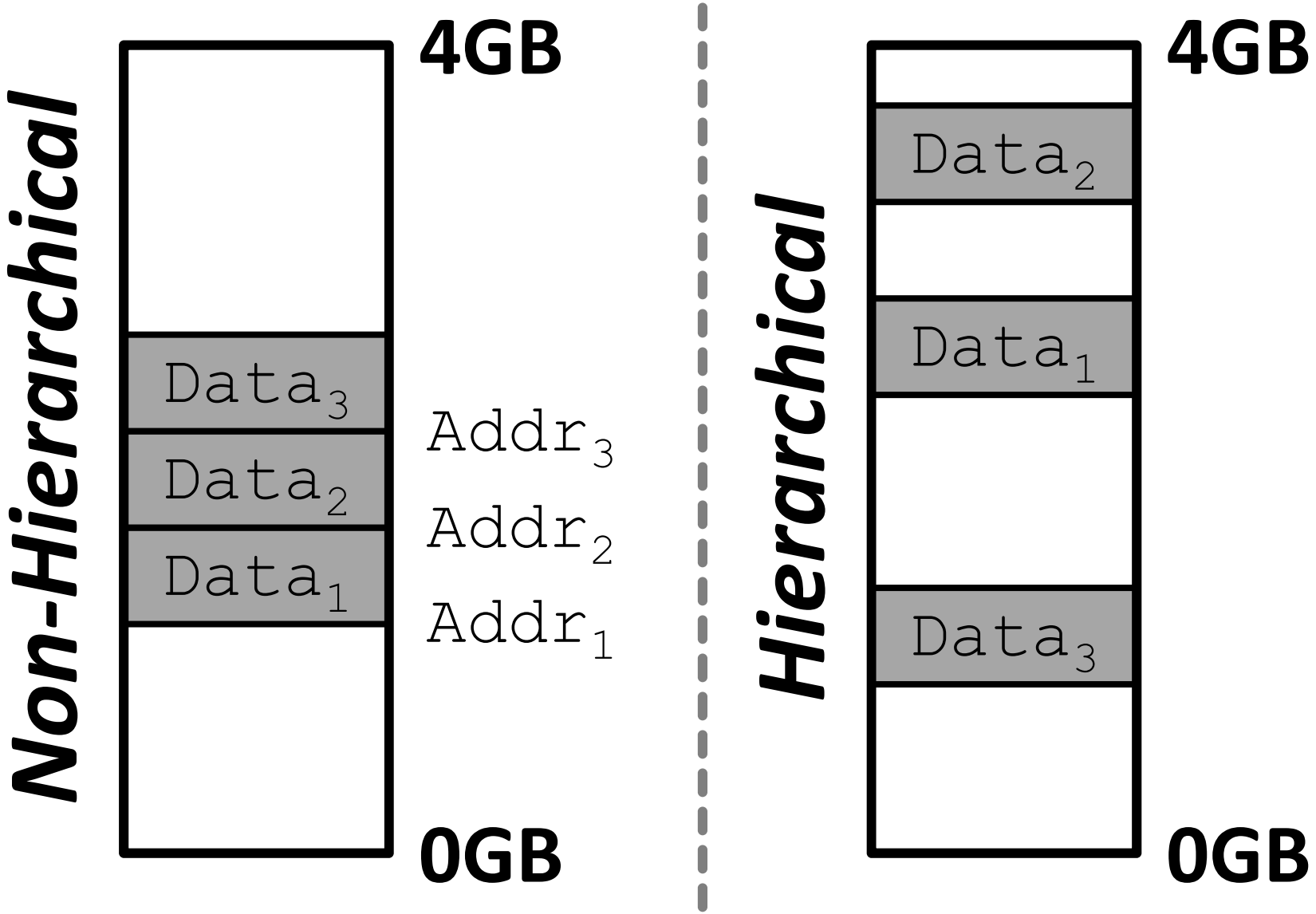
What is HICAMP?

1. **H**ierarchical
2. **I**mmutable
3. **C**ontent-**A**ddressable **M**emory
4. **P**rocessor

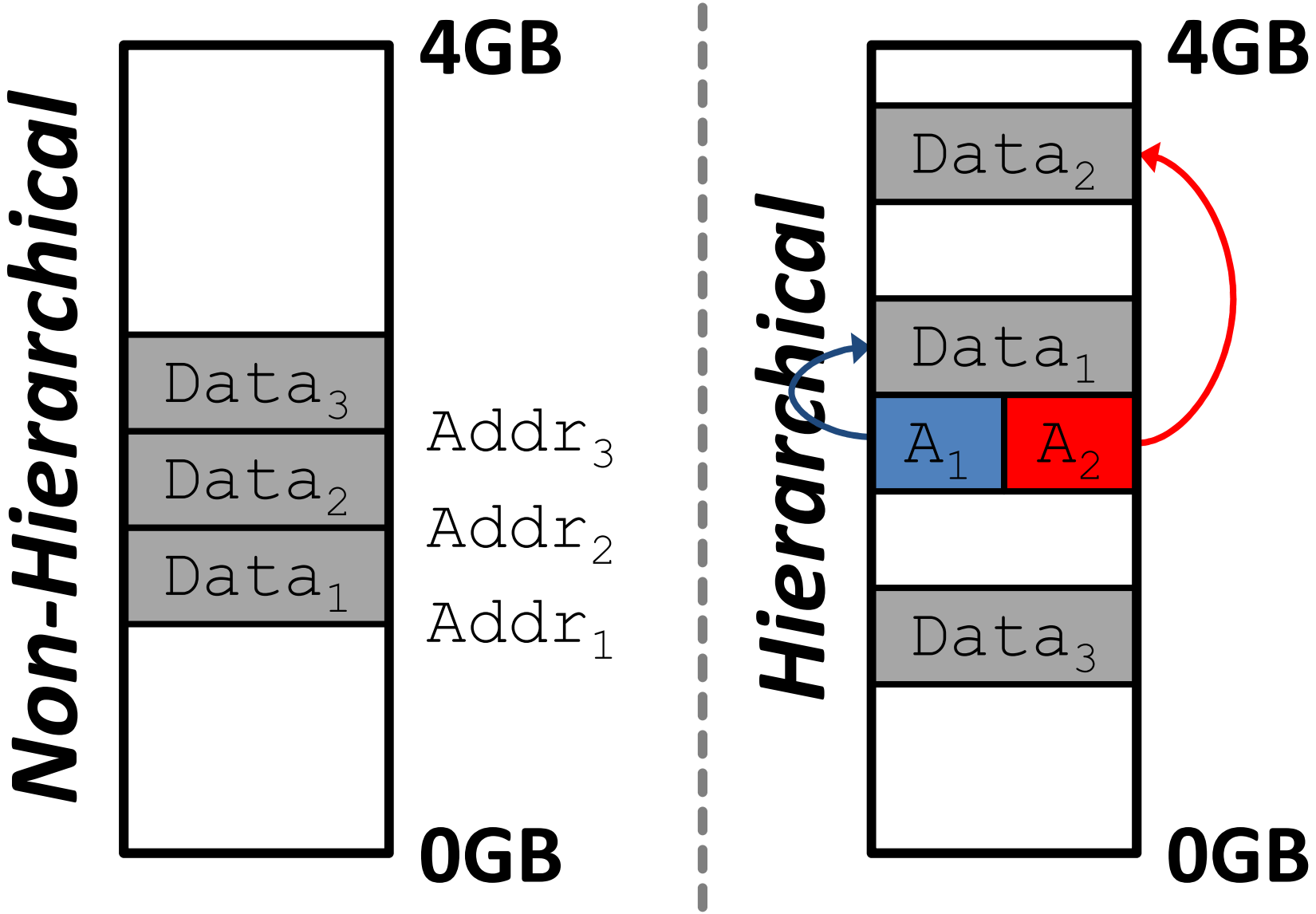
1. ‘H’ of HICAMP: “Hierarchical”



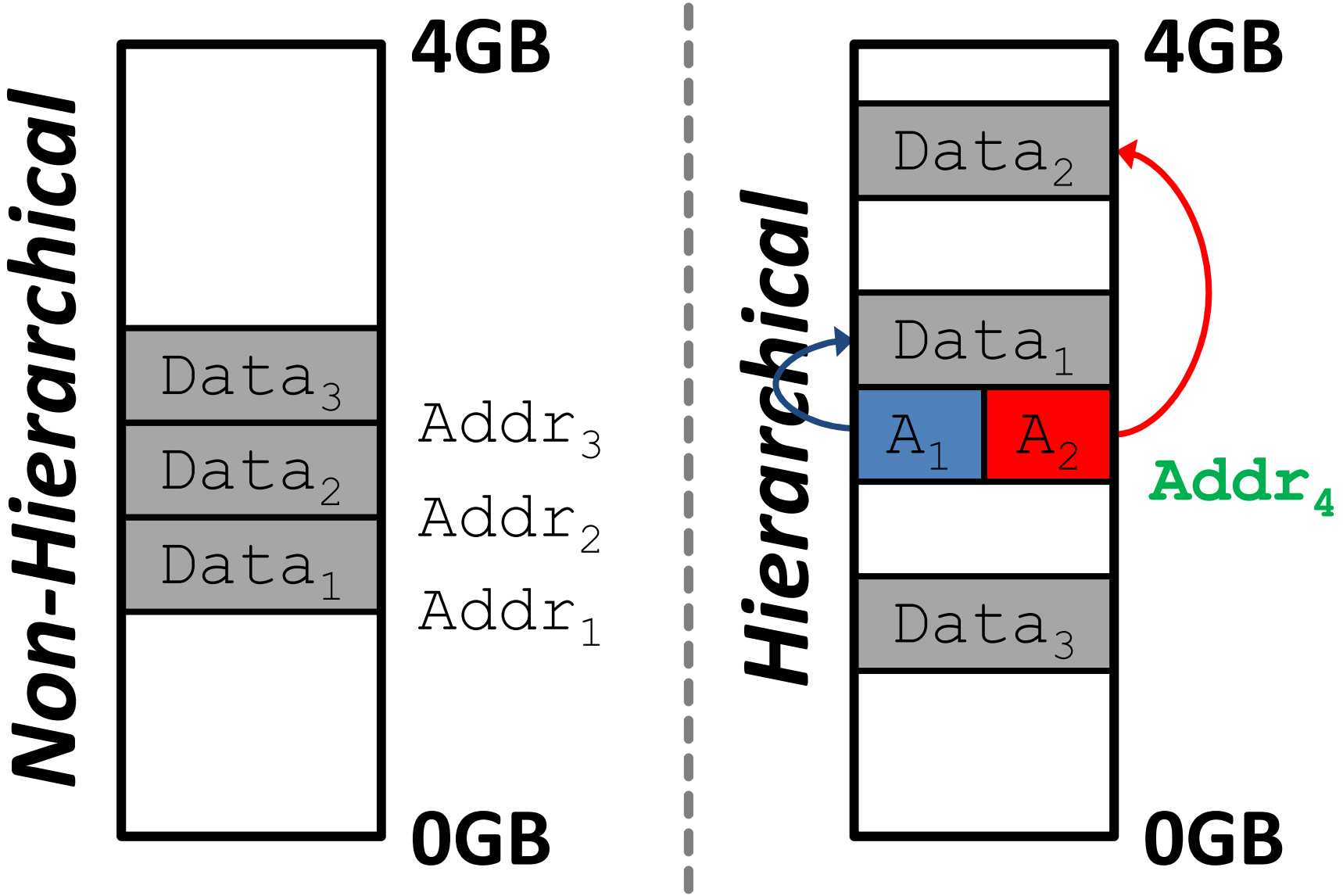
1. ‘H’ of HICAMP: “Hierarchical”



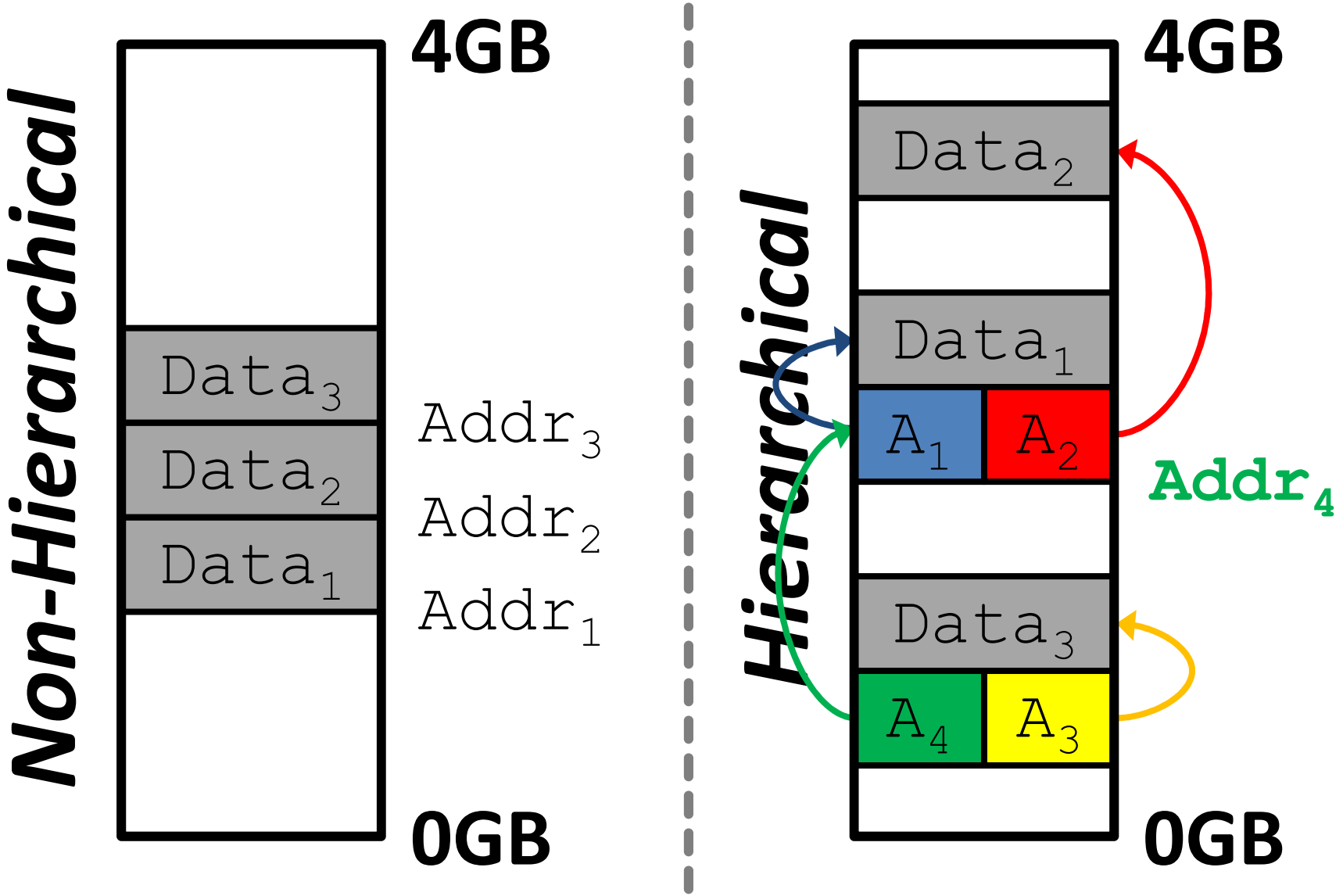
1. ‘H’ of HICAMP: “Hierarchical”



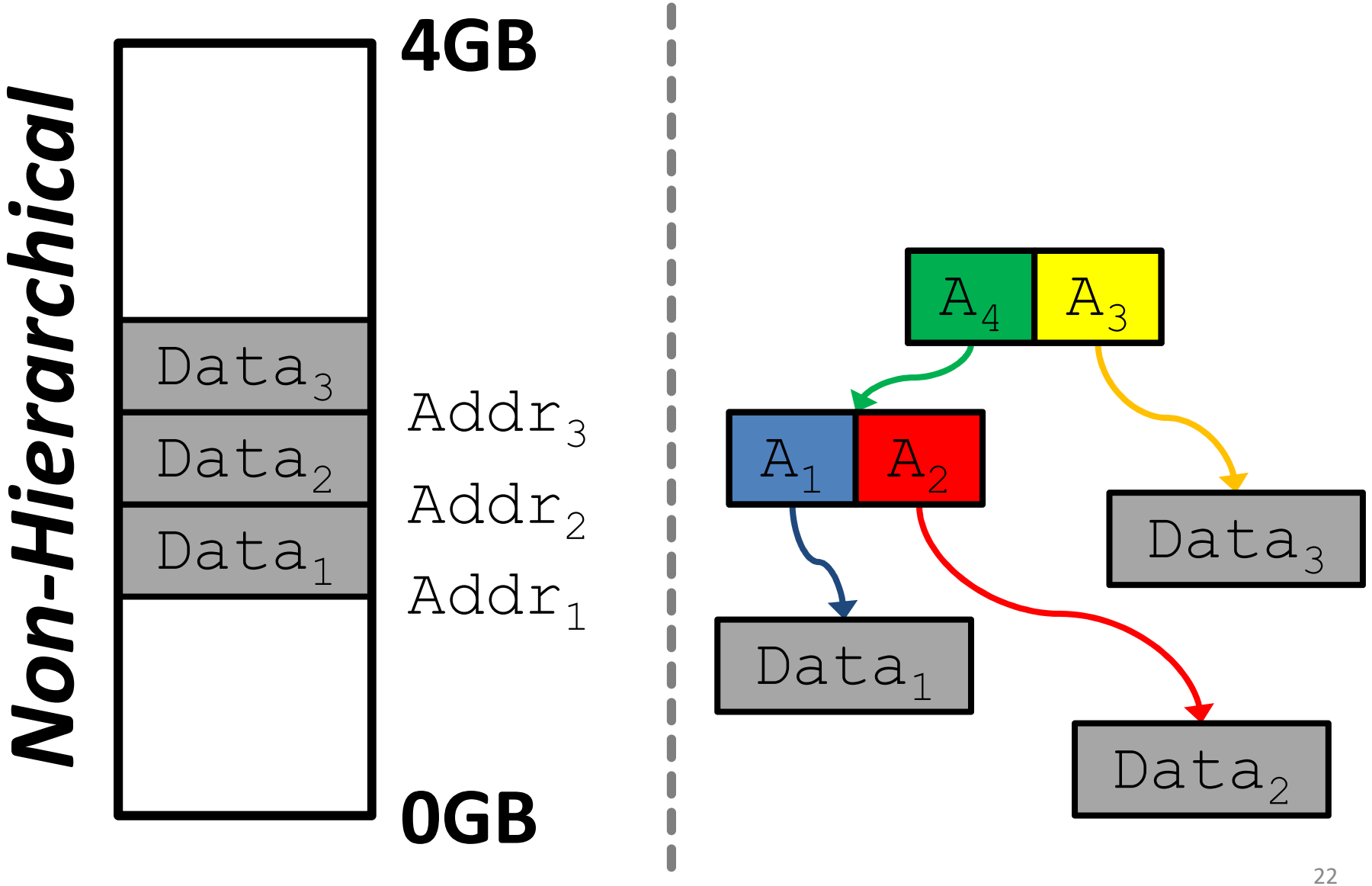
1. ‘H’ of HICAMP: “Hierarchical”



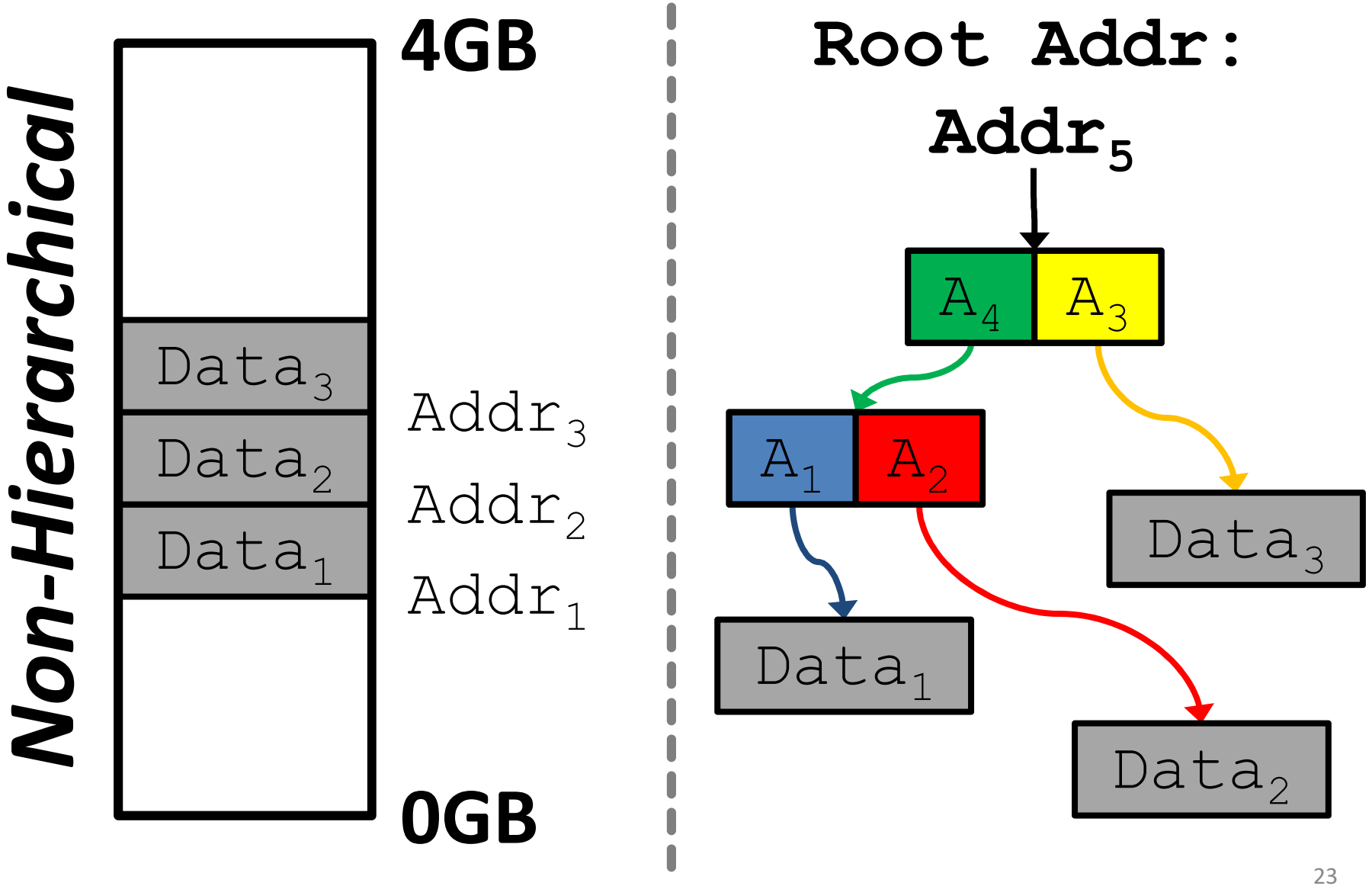
1. ‘H’ of HICAMP: “Hierarchical”



1. ‘H’ of HICAMP: “Hierarchical”



1. ‘H’ of HICAMP: “Hierarchical”

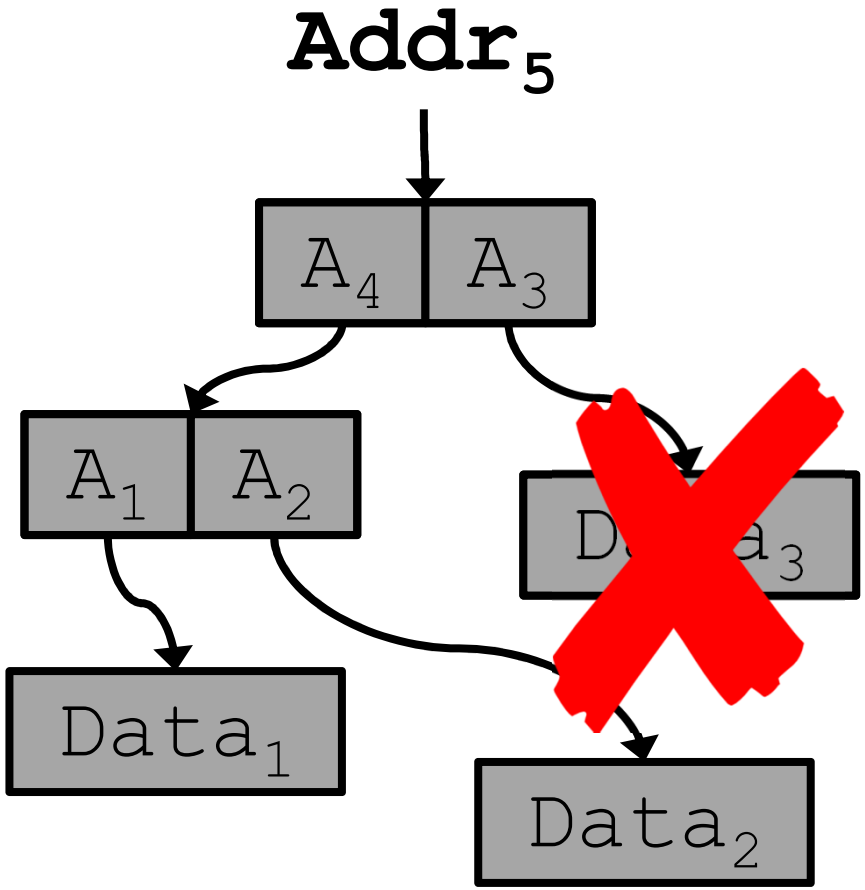


What is HICAMP?

1. **H**ierarchical
2. **I**mmutable
3. **C**ontent-**A**ddressable **M**emory
4. **P**rocessor

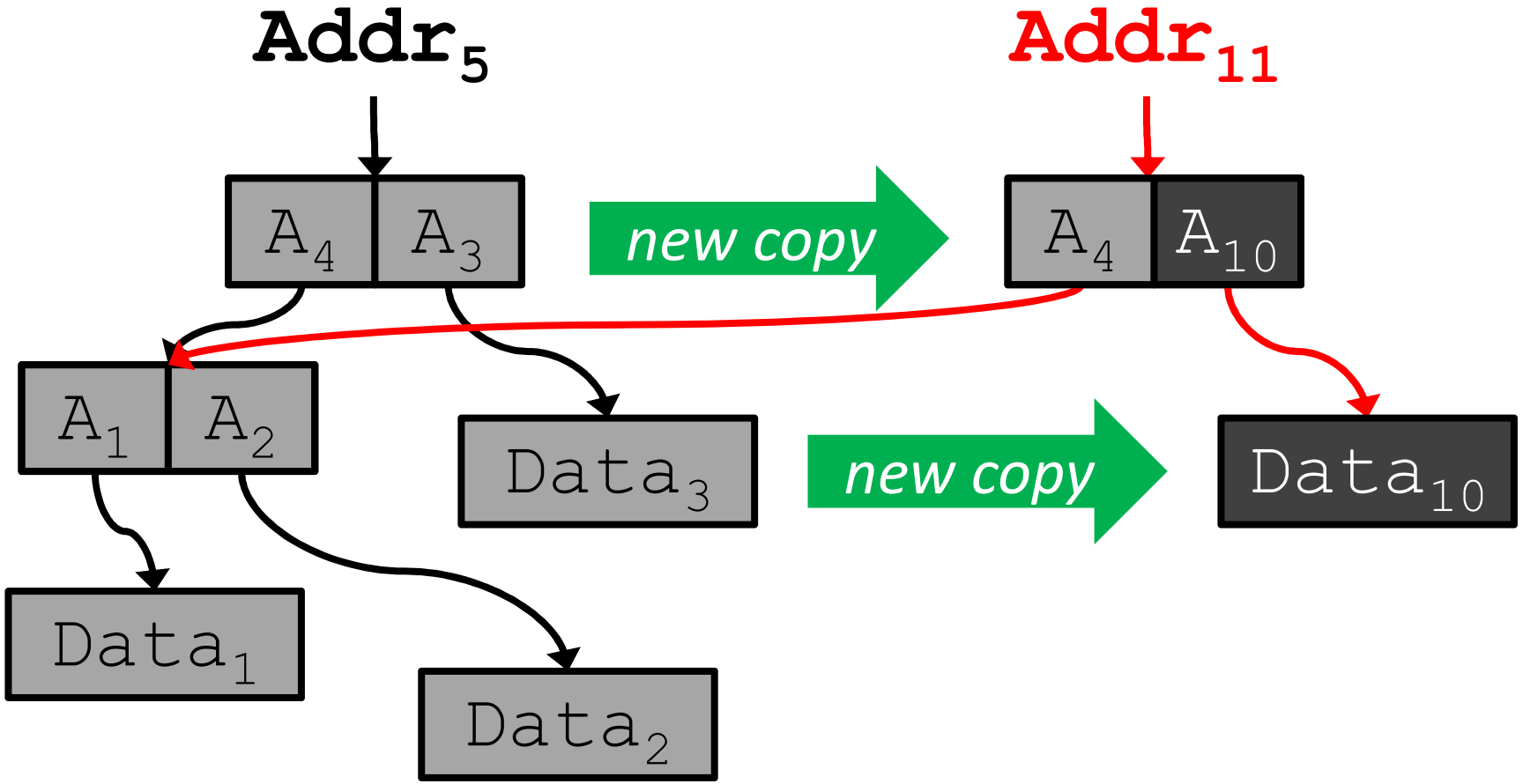
2. ‘I’ of HICAMP: “Immutable”

Overwriting of data is <u>not</u> allowed



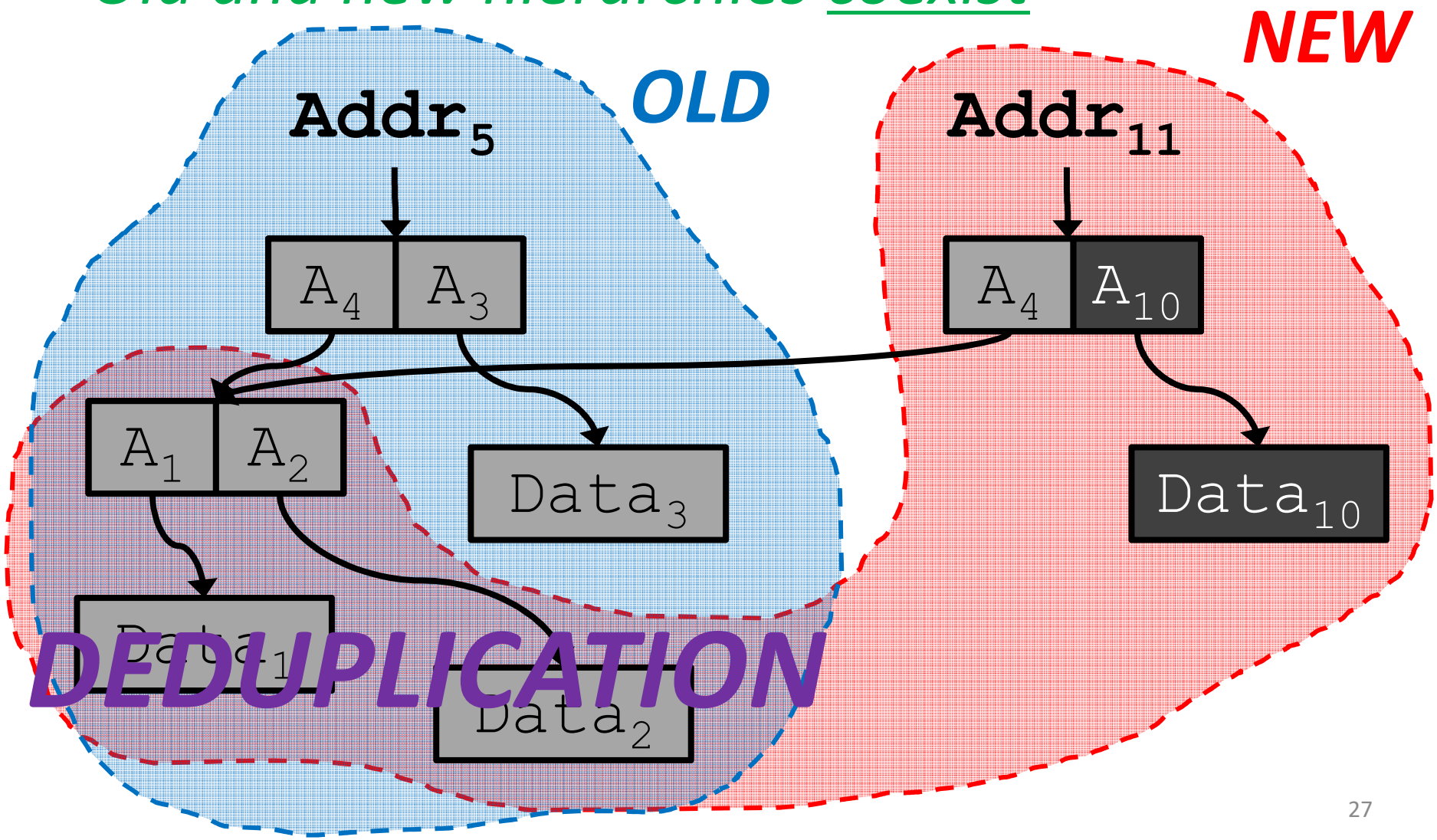
2. ‘I’ of HICAMP: “Immutable”

You must create a new hierarchy



2. ‘I’ of HICAMP: “Immutable”

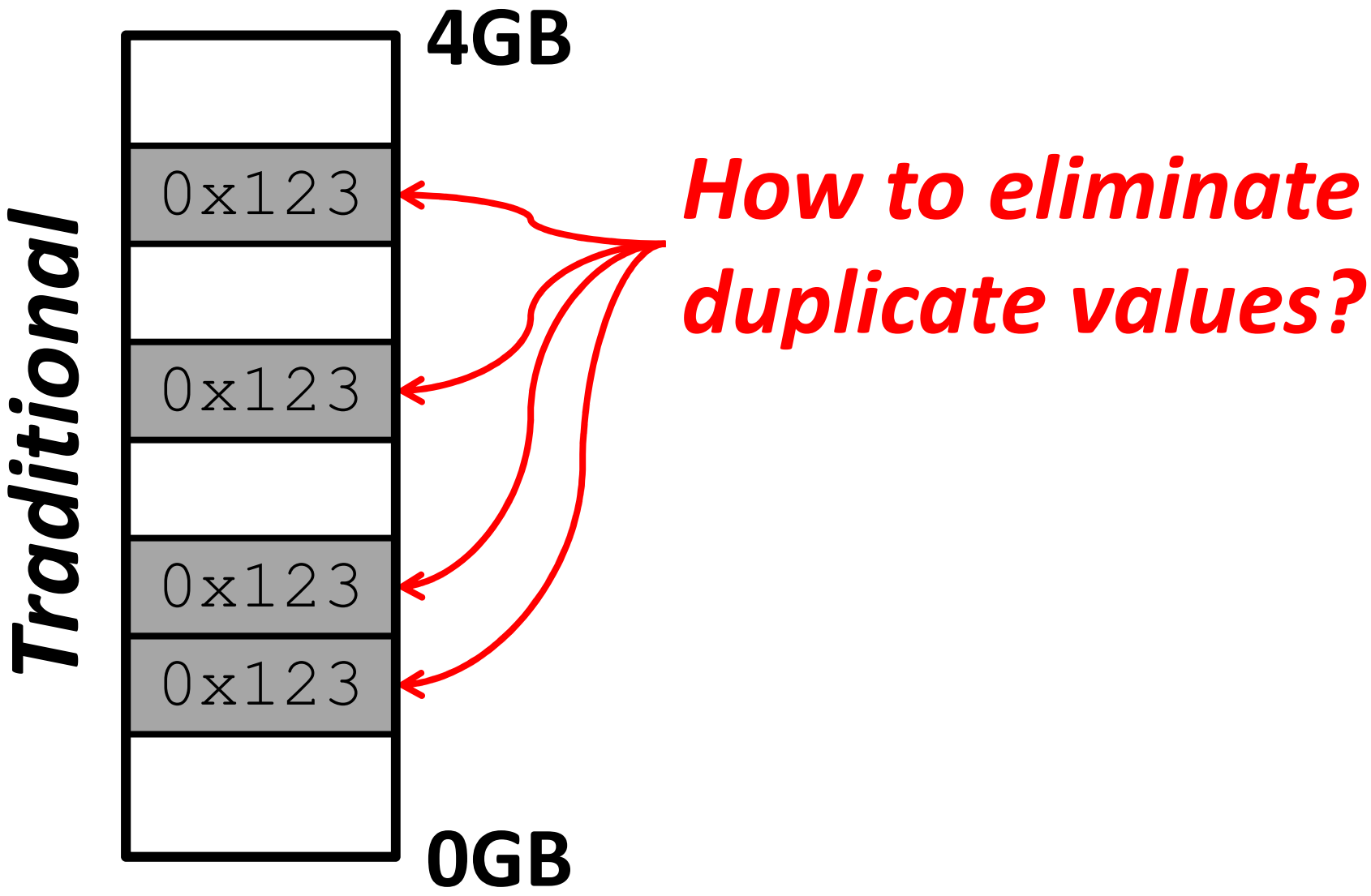
Old and new hierarchies coexist



What is HICAMP?

1. **H**ierarchical
2. **I**mmutable
3. **C**ontent-**A**ddressable **M**emory
4. **P**rocessor

3. “CAM” of HICAMP



3. “CAM” of HICAMP

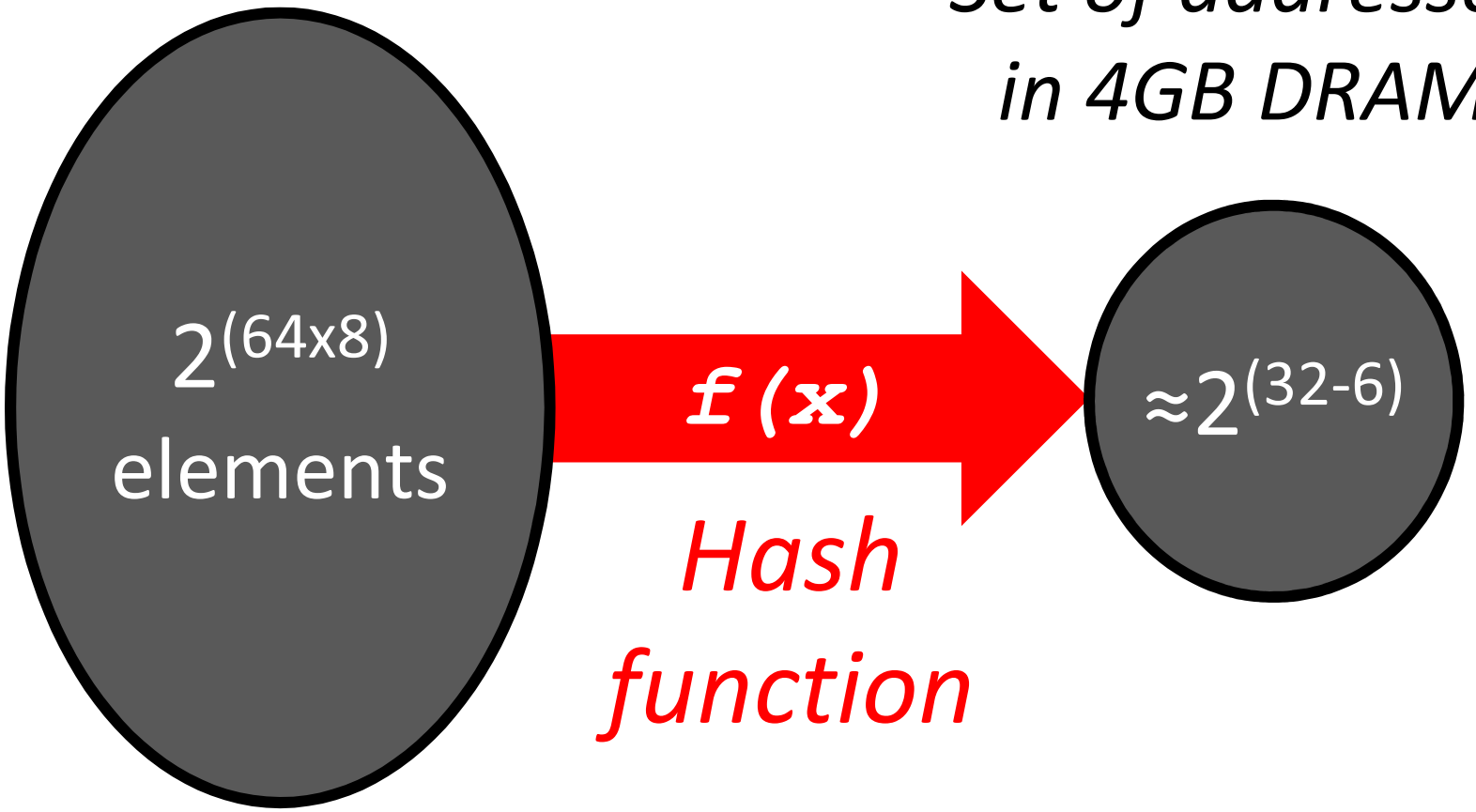
- **Q:** Why do duplicates exist?
- **A:** Because you can store the same value *<u>anywhere</u>* you want

For a particular value, let’s restrict the addresses it can have

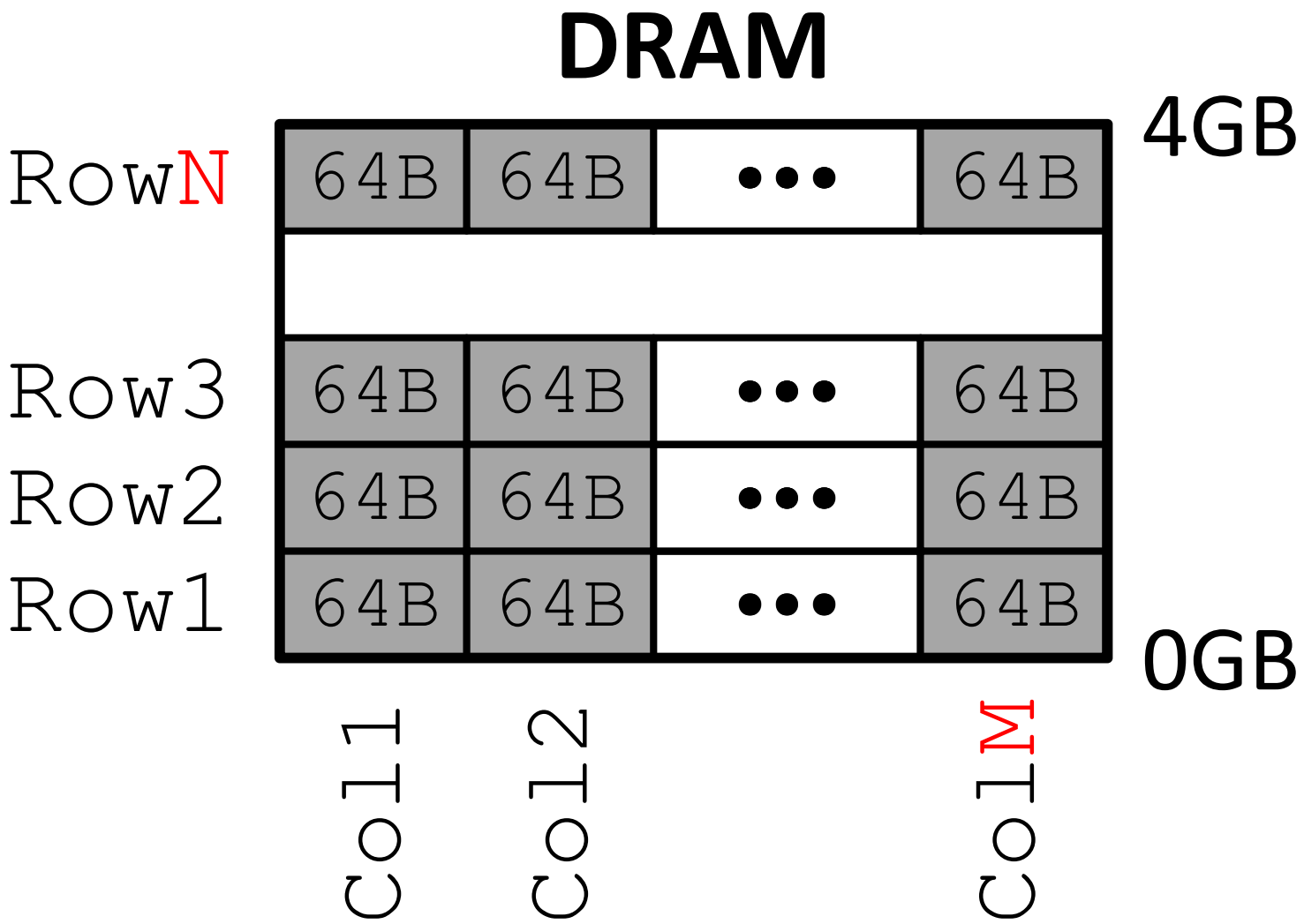
3. “CAM” of HICAMP

Set of 64-byte values

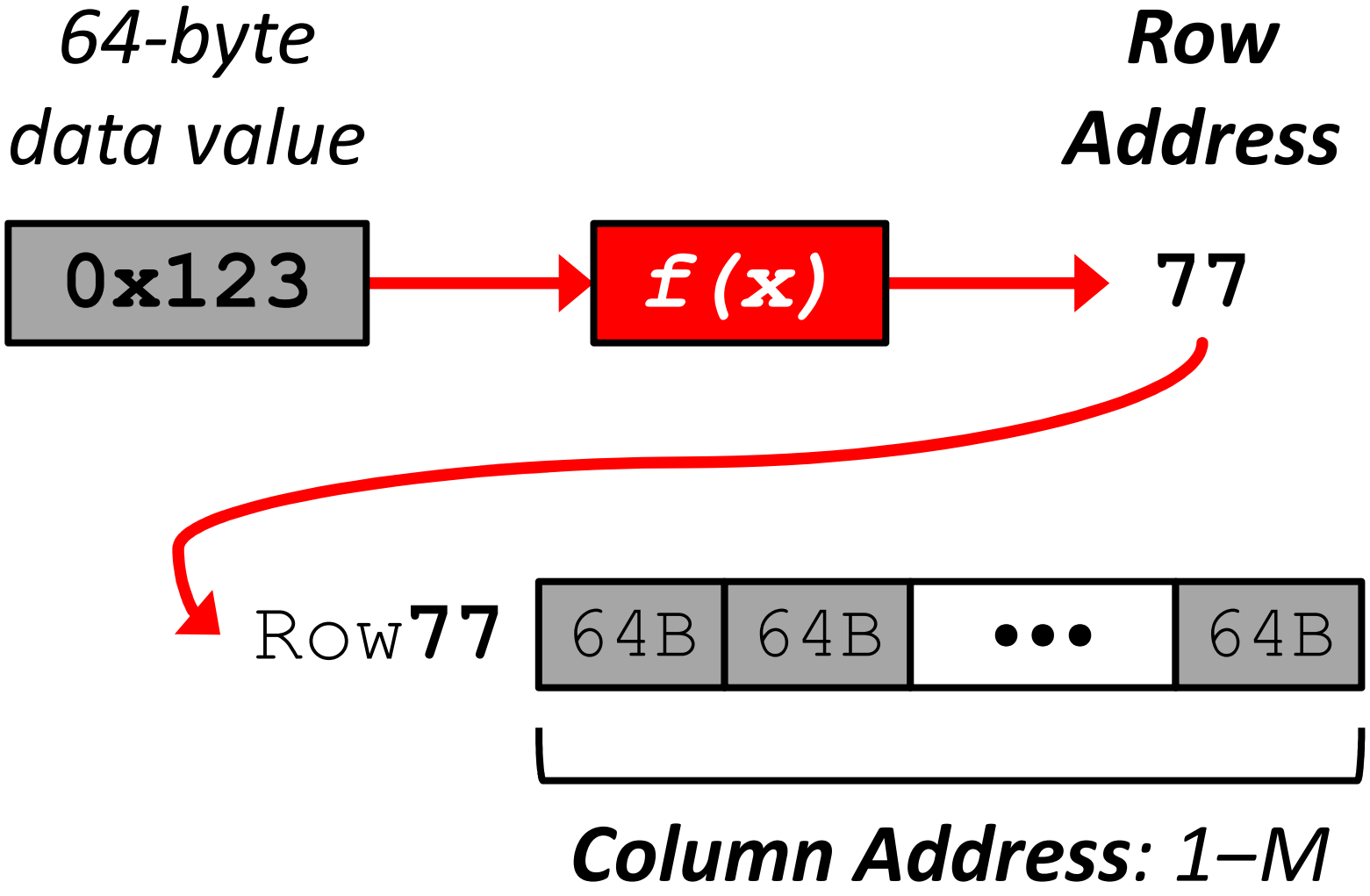
Set of addresses in 4GB DRAM



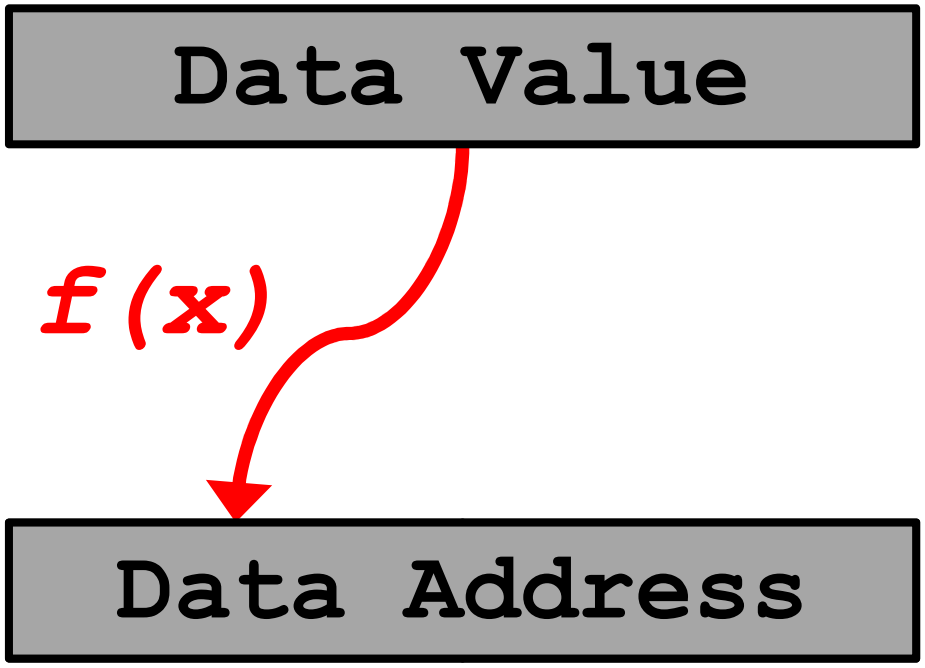
3. “CAM” of HICAMP



3. “CAM” of HICAMP



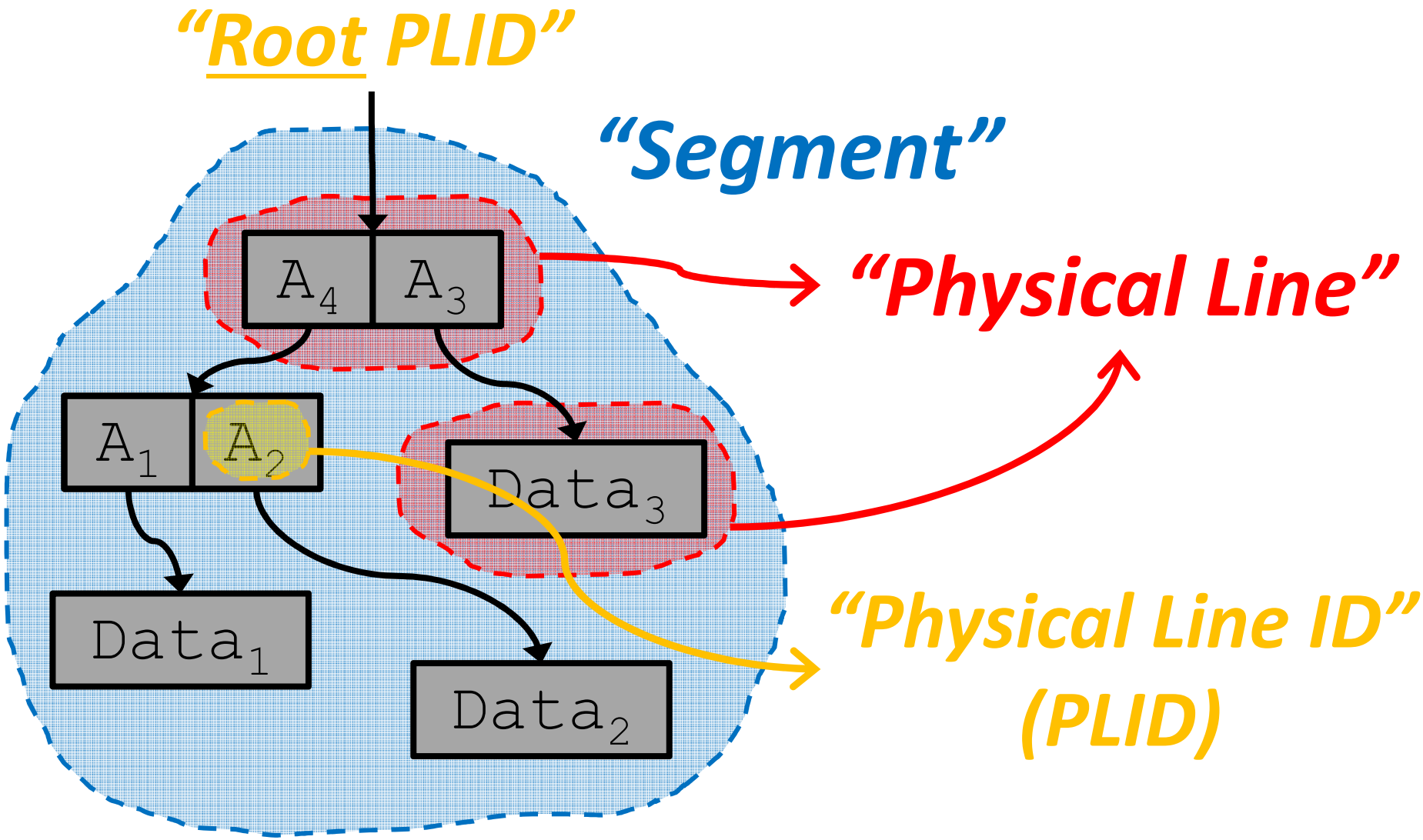
3. “CAM” of HICAMP



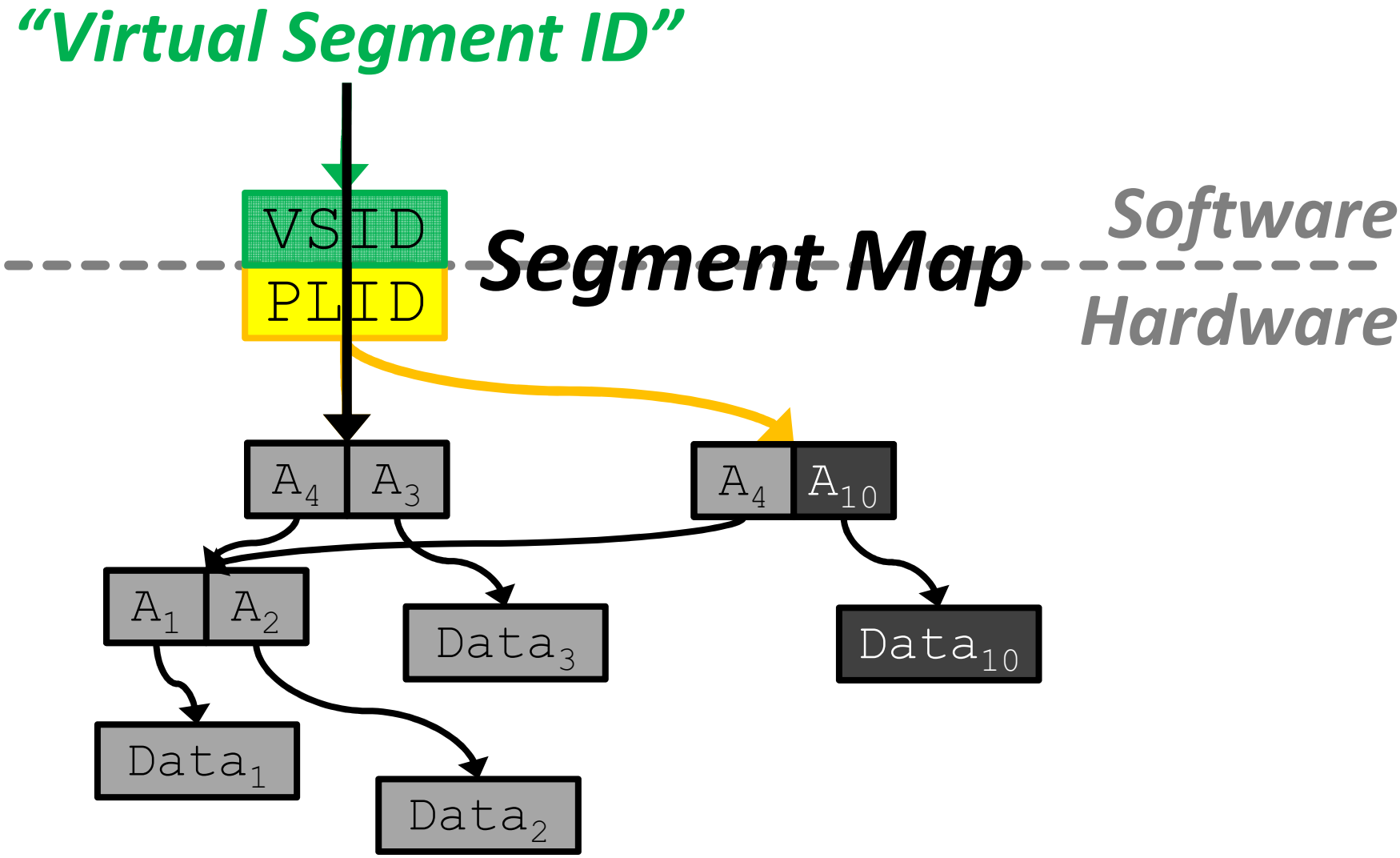
fixed ***flexible:*** *to reduce hash conflicts*

PROGRAMMING MODEL

Terminology



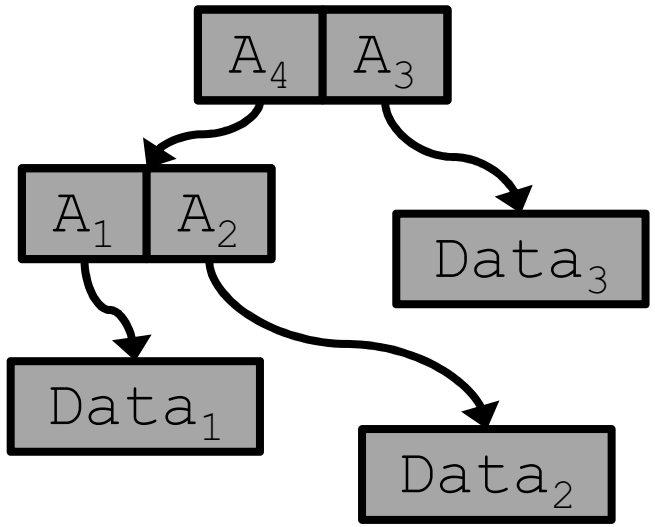
Virtual-to-Physical Translation



Example Program

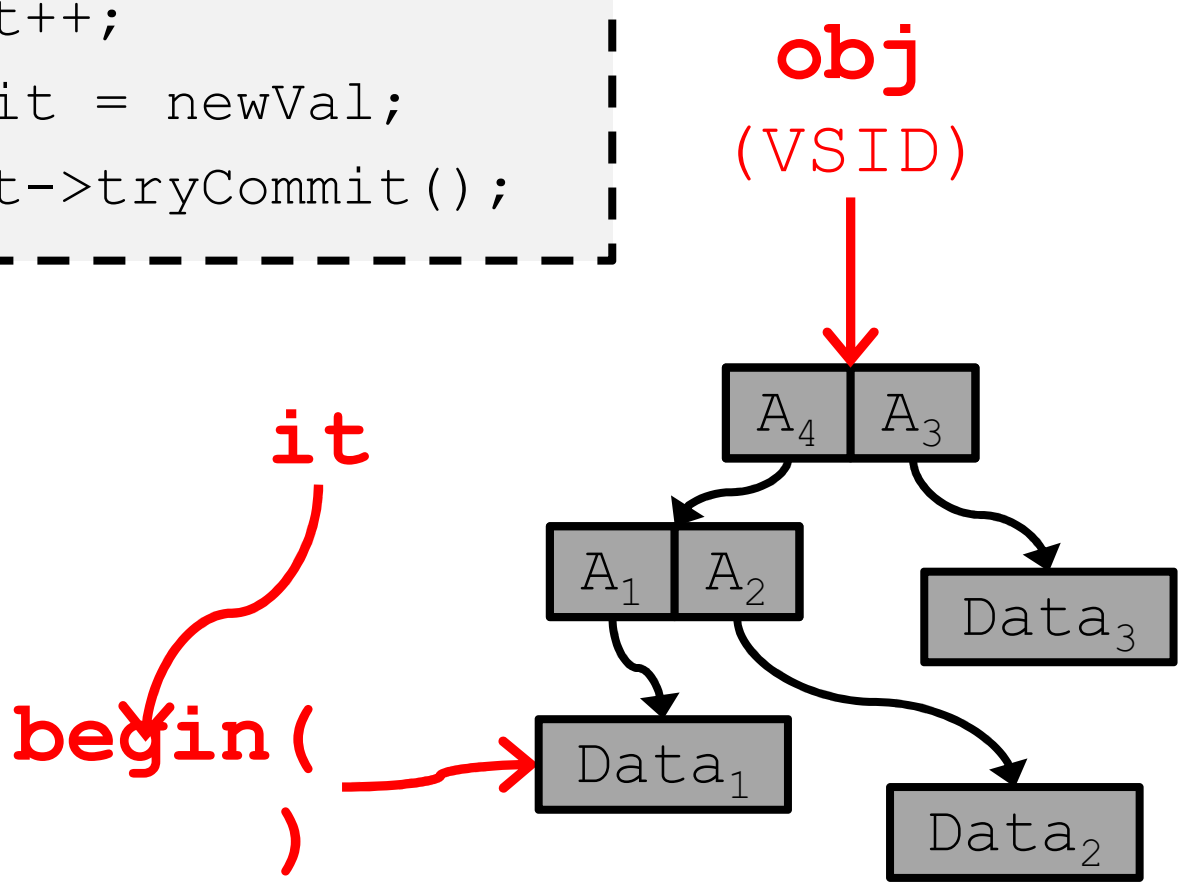
```
1: it = obj.begin();
2: it++;
3: it++;
4: *it = newVal;
5: it->tryCommit();
```

```
/* it = iterator */
```



Example Program

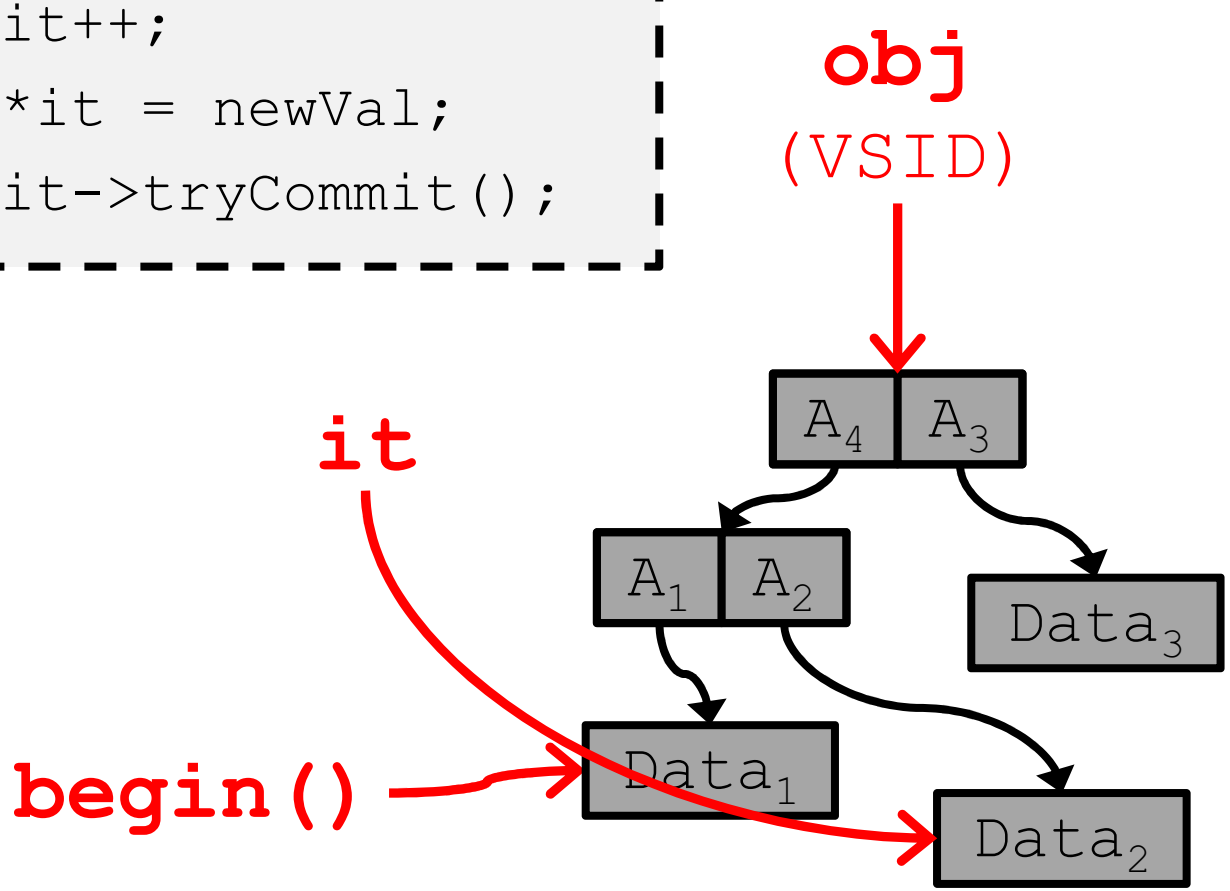
```
1: it = obj.begin();   /* it = iterator */
2: it++;
3: it++;
4: *it = newVal;
5: it->tryCommit();
```



Example Program

```
1: it = obj.begin();
2: it++;
3: it++;
4: *it = newVal;
5: it->tryCommit();
```

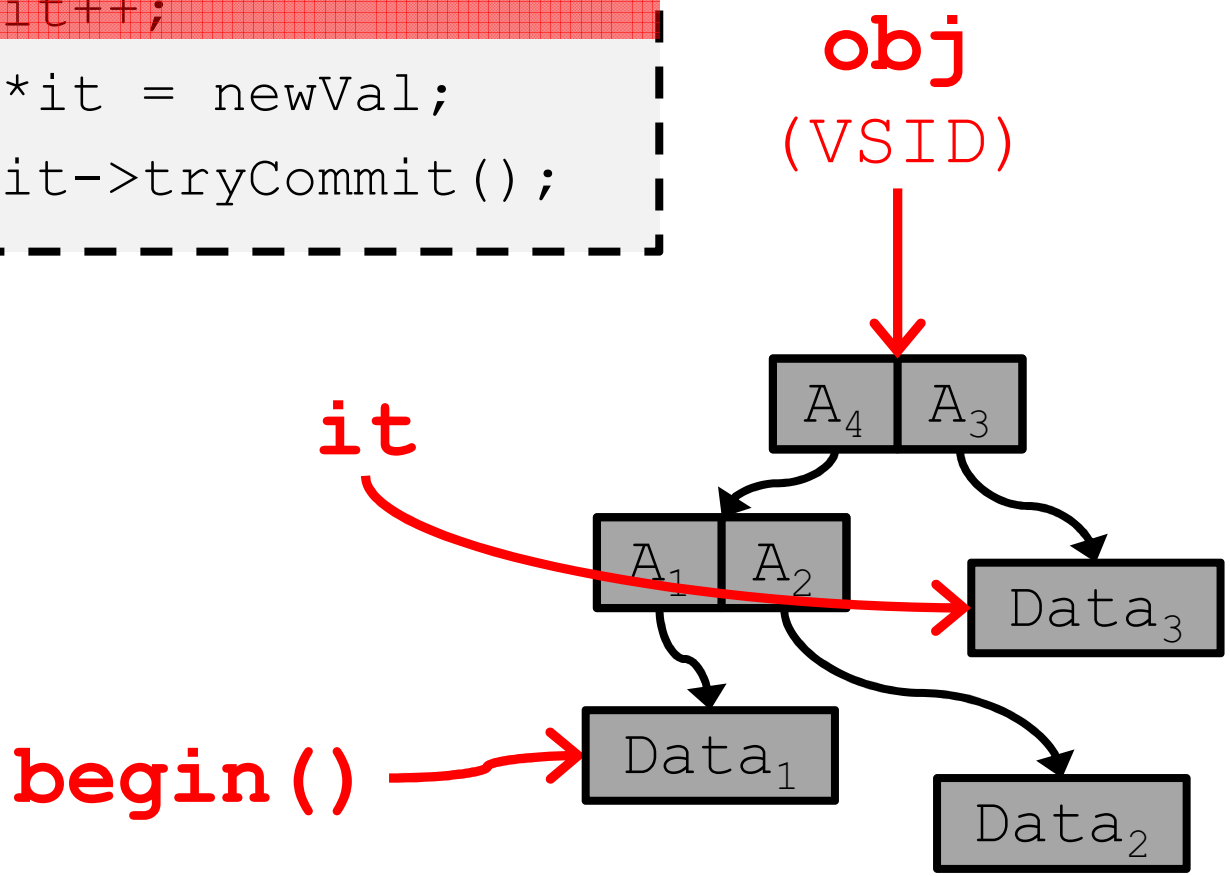
```
/* it = iterator */
```



Example Program

```
1: it = obj.begin();
2: it++;
3: it++;
4: *it = newVal;
5: it->tryCommit();
```

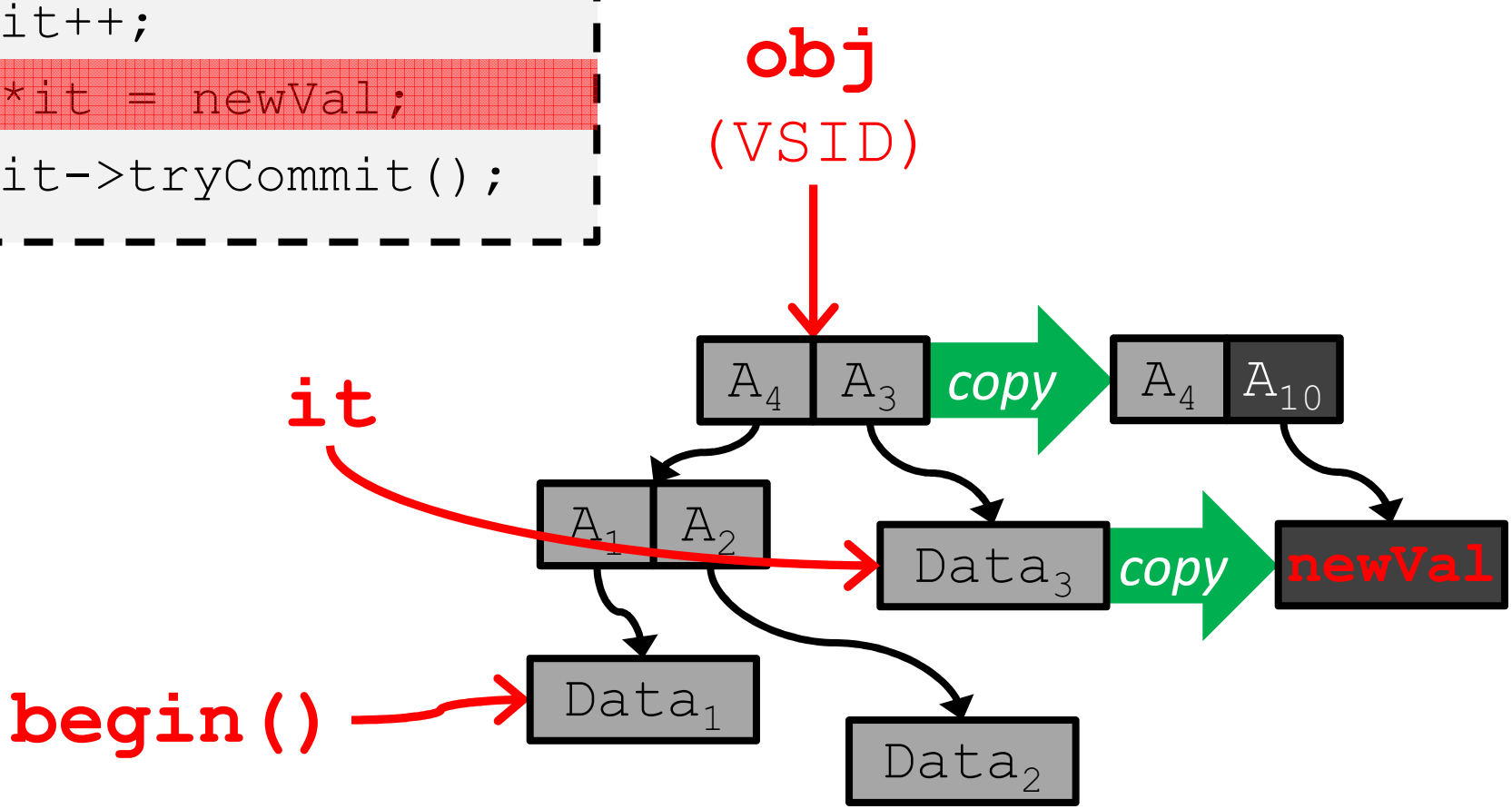
```
/* it = iterator */
```



Example Program

```
1: it = obj.begin();
2: it++;
3: it++;
4: *it = newVal;
5: it->tryCommit();
```

```
/* it = iterator */
```



Example Program

```
1: it = obj.begin();
2: it++;
3: it++;
4: it = newVal;
5: it->tryCommit();
```

```
/* it = iterator */
```