Cache – Part 2

Enhancements and Replacement Policies

Philippos Papaphilippou

Split Caches

- Splitting the cache fixes the storage space for data and instructions
- Fixed space is a disadvantage,
- but in that way there is a new opportunity to optimize each cache for its function
- Adjusting associativity, capacity and block size
- Bandwidth gets double
- Overcomes the need for stalls when some instruction misses occur (structural hazards)

Write Allocate

- On CPUs with **write allocate** a block is allocated to cache as if there was a read miss

  This scheme is usually present in write-back caches to eliminate latency for subsequent changes

- On simpler CPUs with **no-write allocate** writes are made exclusively into main memory

  This scheme is usually present in write-through caches because write allocate then is unnecessary

Write Allocate

- On write allocate caches the locally updated blocks to be replaced are called **victim** blocks
- There is **dirty bit** on each cache block to indicate whether the corresponding block in RAM is “clean” or “dirty”
- When there is the need to replace the locally updated blocks, they are transferred to the **victim buffer**
- When the victim buffer gets full, cache stalls

Types of misses

- We can sort all misses into the following categories in order to help better cache designs
- **Compulsory** misses occur when blocks are fetched for the first time (e.g. during boot up)
- **Capacity** misses occur when cache's size is not sufficient for a task, leading to retrieving the same blocks continuously
- **Conflict** misses occur when multiple needed blocks are mapping to the same set but the degree of associativity is not sufficient
- **Coherency** misses occur in multiprocessors

Reducing Miss Rates

- For Compulsory misses we can't do anything but to hide them by increasing block size (spatial localities help prefetch useful data)

  Trade-off: increase in miss penalty

- For Capacity misses, cache size could be increased

  Trade-off: increase in hit time, cost, power

- For Conflict misses, degree of Associativity could be increased

  Trade-off: increase in hit time

Considerations

- It's important to recognize that **lower miss rates isn't the only contributor to performance**
- Average memory access time formula can be used

  AMAT = Time for a hit + Miss rate * Miss penalty
- Even then we don't know the exact **execution time** because there is CPU execution time aside from AMAT
- For out-of-order processors **overlapped latency** should be taken to consideration also
- Stalls for I/O device RAM accesses (OS interactions)
- **Cache-obvious algorithms** prevent **thrashing**

Replacement Policies

- **Replacement policies** need to be applied when replacing a block in associative cache because there is a choice of n elements to be removed
- The tasks are **victim selection** and **insertion policy**
- Common replacement policies include

  Random - Replace a Random block on a miss

  FIFO - First in - First out

  LRU - replace Least Recently Used block
- We must see the complexity of implementation apart from improvements in miss rates

Random

Original animation can be found in the “xi” svn shared repository or at http://www.cs.ucy.ac.cy/~ppapap01/

Random (Cache Replacement Policy)

5-way set:

Pseudo-random number: 2

Philippos Papaphilippou

LRU (Least Recently Used)

- On a hit, the priority of the block changes to the most recent
- On a miss, the least important block in the "queue" is replaced with the fetched block in the beginning of the queue (**MRU insertion**)
- Using LRU in an associative cache gives high hit rates because it gives a high opportunity to have a hit for a block in the recent future (max. after n - 1 sequential misses)
- **FIFO** is like LRU but it does not rearrange priority on a hit

Least Recently Used (LRU)

Original animation can be found in the “xi” svn shared repository or at http://www.cs.ucy.ac.cy/~ppapap01/

LRU (Least Recently Used)

Miss Example

5-way set:

Priority

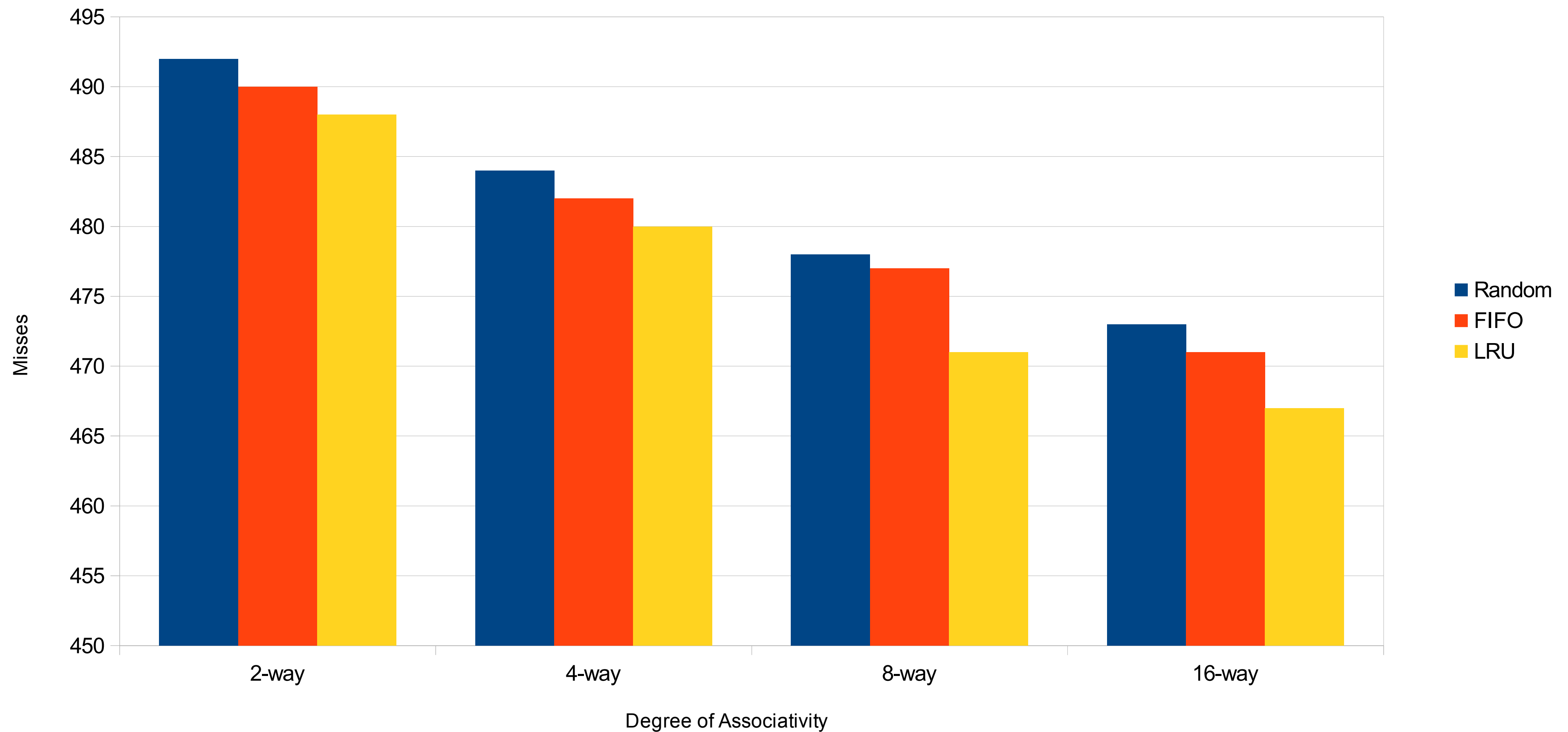
Philippos Papaphilippou

Pseudo-LRU

- In addition to cost, complex designs may introduce an increase in latency (e.g. for searching)
- Generally for caches with degree of Associativity greater than 4 use **Pseudo-LRU** to replace LRU
- A common implementation of Pseudo-LRU is **Bit-PLRU** (and Tree-PLRU)

  Each block has the LRU bit to indicate its status. At first all bits are zeros. When there is a hit on a block, its LRU bit becomes 1. When all become 1s they become 0s. On a miss the left-most block with 0 is replaced

Cache Simulator Runs



More Replacement Policies

- **LRU Insertion Policy** (**LIP**) - On misses blocks are inserted in the LRU position. On hits blocks go to the MRU position (like LRU Replacement Policy)
- **Bimodal Insertion Policy** (**BIP**) - On misses blocks are inserted mostly in the LRU position and with low probability in the MRU position. On hits blocks go to the MRU position (like LRU Replacement Policy)
- **Dynamic Insertion Policy** (**DIP**) - Uses **Set Dueling** (a few dedicated sets that use LRU and DIP policies) to behaviorally select replacement policy for the rest of the cache by counting the best number of misses.

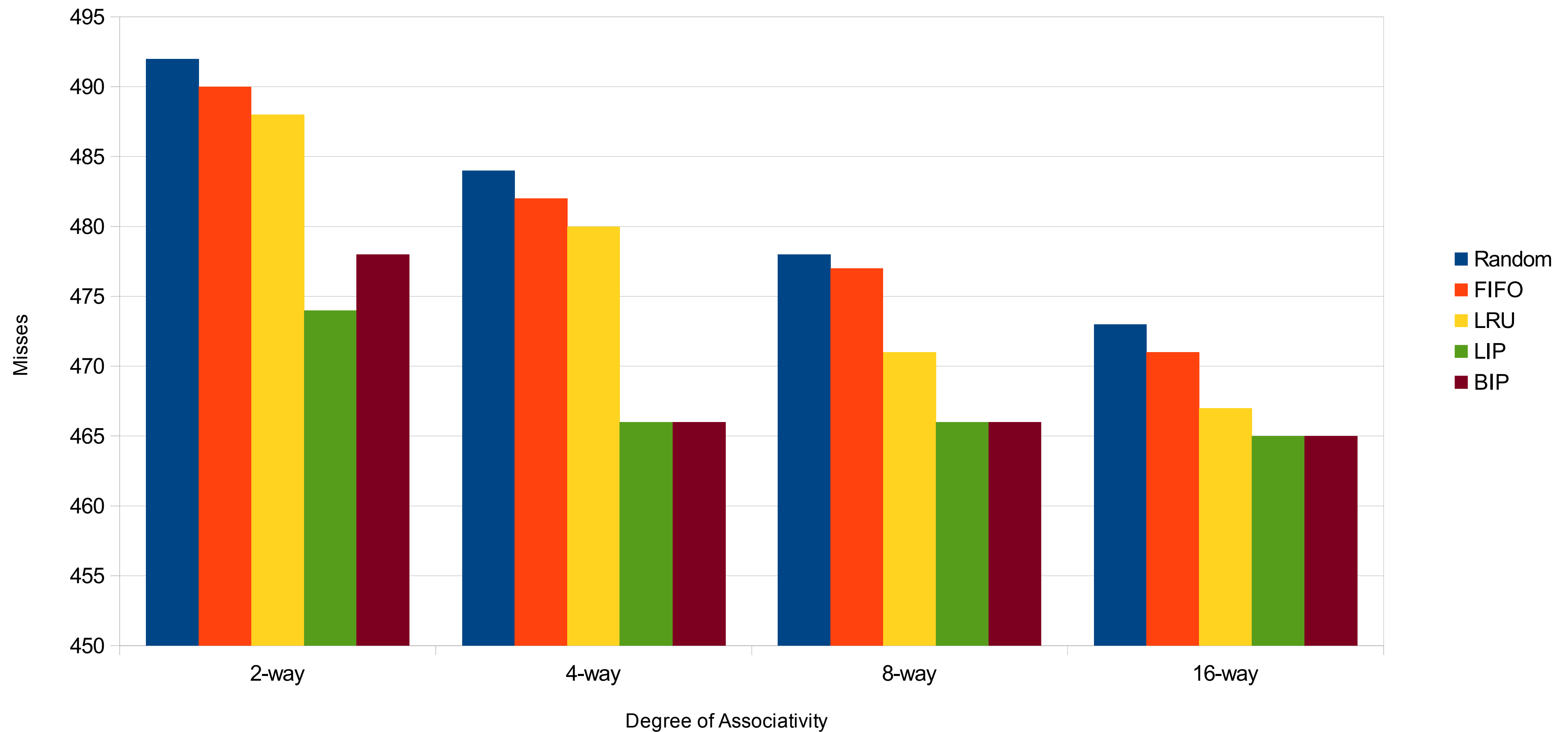
Performance

- LRU is good for workloads with a cyclic access pattern but may produce thrashing if the task has greater working set
- LIP prevents thrashing for memory-intensive tasks, but its not optimal for LRU-friendly workloads
- BIP can respond to changes in working set having also the thrashing protection of LIP.
  It works probabilistically using the **bimodal throttle parameter (ε)**. Using small ε to sporadicly work like LRU

For ε = 1 have LIP
For ε = 0 have LRU

```
If (rand()%1000 < epsilon*1000)
      mode = LRU;
else
      mode = LIP;
```

Cache Simulator Runs



a Python Script

Input:

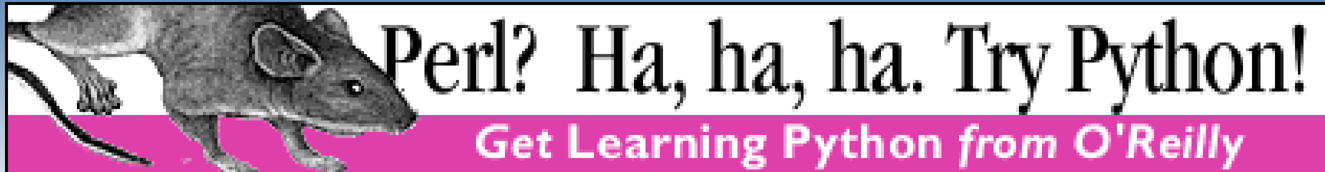
```
address 4608
address 582272
address 1717248
address 4608
address 1377856
address 1835840
address 1377920
address 1378880
address 582336
address 1372672
address 1372736
address 1717248
address 1757696
address 1372992
address 1372800
address 1373376
address 1373568
Address 1373440
...
```

Script:

```
#! /usr/bin/env python3.3
# (Python3 location)
print('Bounds for Cache with 4 sets:')
f=open('input.dat')
elements =[[],[],[],[]]; # Nof sets = 4
for line in f:
        tag=int(line.strip('adres \n'))>>6 # offset = 6
        a=tag&3
        tag >>= 2 # index for sets = 2
        if tag not in elements[a]:
            elements[a].append(tag)
total=0
for x in range(0,4):
    print('Unique elements for set',x,'=',len(elements[x]))
    total += len(elements[x])
print('-> Minimum sum of misses  =',total)
```

Output:

```
Bounds for Cache with 4 sets:
Unique elements for set 0 = 122
Unique elements for set 1 = 112
Unique elements for set 2 = 114
Unique elements for set 3 = 116
-> Minimum sum of misses  = 464
```



The End

Sources

- Computer Architecture: A Quantitative Approach John L. Hennessy, David A. Patterson
- Computer Organization and Design 4th Edition David A. Patterson, John L. Hennessy
- Adaptive Insertion Policies for High-Performance Caching Moinuddin K. Qureshi, Yale N. Patt, Aamer Jaleel, Simon C. Steely Jr., Joel Emer (ISCA 2007)